Генерация текста на основе "Гарри Поттер и методы рационального мышления"

Описание проекта

Данный проект реализует различные подходы к **генерации текста** с использованием современных методов обработки естественного языка. В качестве обучающих данных используется фанфик "Гарри Поттер и методы рационального мышления" - произведение, сочетающее магический мир с научным подходом к мышлению.

Основные задачи:

- **Сбор и предобработка данных**: Извлечение текста с веб-сайта и очистка
- **Различные типы токенизации**: Посимвольная, пословная, BPE
- **Архитектуры нейронных сетей**: RNN, LSTM, Bidirectional LSTM, Transformer
- **Генерация текста**: Создание новых текстов в стиле оригинала

Изучаемые модели:

1. **Simple RNN** - базовая рекуррентная сеть
2. **LSTM** - долгосрочная память
3. **Bidirectional LSTM** - двунаправленная обработка
4. **GPT-style Transformer** - архитектура внимания

Начнем с загрузки и предобработки данных:

Сбор данных с веб-сайта

Загрузка текста из интернета

```
# Импорт необходимых библиотек для веб-скрапинга
import requests
from bs4 import BeautifulSoup
from tqdm import tqdm
import time

# Базовый URL для загрузки глав
base_url = "https://hpmor.ru/book/1/"
all_text = ""

print("Начинаем загрузку текста из интернета...")
```

```
print("Источник: https://hpmor.ru/book/1/")

# Загружаем первые 10 глав для демонстрации
for chapter in tqdm(range(1, 11), desc="Загрузка глав"):
    url = f"{base_url}{chapter}/"
    try:
        # Отправляем GET-запрос
        response = requests.get(url)
        response.raise_for_status()  # Проверяем успешность запроса

        # Парсим HTML-содержимое
        soup = BeautifulSoup(response.content, "html.parser")
        paragraphs = soup.find_all("p")

        # Ищем индекс абзаца со звездочками (разделитель между метаданными и те
        start_index = next(
            (i for i, p in enumerate(paragraphs) if p.get_text(strip=True) == "
            None
        )

        if start_index is None:
            print(f"Глава {chapter}: звездочки не найдены, пропускаем")
            continue

        # Берем только абзацы после звездочек и исключаем теги <em>
        filtered_paragraphs = [
            p.get_text(strip=True) for p in paragraphs[start_index + 1:]
            if not p.find("em")  # Исключаем курсивные примечания
        ]

        chapter_text = "\n".join(filtered_paragraphs)

        # Добавляем заголовок главы и текст
        all_text += f"\n\n=== Глава {chapter} ===\n\n"
        all_text += chapter_text

        # Вежливая задержка между запросами
        time.sleep(0.2)

    except Exception as e:
        print(f"Глава {chapter}: ошибка — {e}")

# Сохраняем результат в файл
with open("harry.txt", "w", encoding="utf-8") as f:
    f.write(all_text)

print("Загрузка завершена! Текст сохранен в файл: harry.txt")
print(f"Общий размер текста: {len(all_text)} символов")
```

```
Начинаем загрузку текста из интернета...
Источник: https://hpmor.ru/book/1/
Загрузка глав: 100%|██████████| 10/10 [00:13<00:00,  1.37s/it]Загрузка завершена
Общий размер текста: 202277 символов
```

Предобработка текста

Функции очистки и нормализации

```
# Импорт библиотек для обработки текста
import pandas as pd
import re

def remove_dialog_dashes(text: str) -> str:
    """
    Удаляет тире в начале строк (реплики диалога)

    Args:
        text (str): Исходный текст

    Returns:
        str: Текст без тире в начале строк
    """
    # Удаляем тире, дефисы и длинные тире в начале строк
    return re.sub(r"(?m)^[\s]*[—–-]\s*", "", text)

def keep_only_russian(text: str) -> str:
    """
    Оставляет только русские буквы и базовые знаки препинания

    Args:
        text (str): Исходный текст

    Returns:
        str: Очищенный текст в нижнем регистре
    """
    # Приводим к нижнему регистру
    text = text.lower()

    # Оставляем только русские буквы, пробелы и основные знаки препинания
    text = re.sub(r"[^а-яё .,!?;\n]", " ", text)

    # Убираем лишние пробелы
    text = re.sub(r"\s+", " ", text)

    return text.strip()

print("Функции предобработки текста определены:")
print("- remove_dialog_dashes(): удаление тире из диалогов")
print("- keep_only_russian(): нормализация русского текста")
```

```
Функции предобработки текста определены:
- remove_dialog_dashes(): удаление тире из диалогов
- keep_only_russian(): нормализация русского текста
```

```
# Загружаем сохраненный текст
with open("harry.txt", "r", encoding="utf-8") as f:
    all_text = f.read()

print(f"Загружен текст размером: {len(all_text)} символов")

# Разделяем главы по заголовкам
chapters_raw = re.split(r"\n+=+ Глава (\d+) =+\n+", all_text)

chapters = []
for i in range(1, len(chapters_raw), 2):
    chapter_number = int(chapters_raw[i])
    chapter_text = chapters_raw[i + 1].strip()

    # Применяем функции очистки
    chapter_text = remove_dialog_dashes(chapter_text)
    chapter_text = keep_only_russian(chapter_text)

    chapters.append({"chapter": chapter_number, "text": chapter_text})

# Создаем DataFrame для удобной работы с данными
df = pd.DataFrame(chapters)
df = df.sort_values("chapter").reset_index(drop=True)

print(f"Обработано глав: {len(df)}")
print("\nПримеры обработанных глав:")
print(df.head())

# Объединяем весь очищенный текст
full_cleaned_text = " ".join(df["text"].astype(str).tolist())
print(f"\nОбщий размер очищенного текста: {len(full_cleaned_text)} символов")
```

```
Загружен текст размером: 202277 символов
Обработано глав: 10

Примеры обработанных глав:
   chapter                                               text
0        1  все стены до последнего дюйма заняты книжными ...
1        2  давайте проясним ситуацию, сказал гарри, папа,...
2        3  господи боже! воскликнул бармен, уставившись н...
3        4  груды галлеонов. стройные ряды серебряных сикл...
4        5  скрытная лавка была маленьким причудливым неко...

Общий размер очищенного текста: 197645 символов
```

Часть 1: Simple RNN

Посимвольная токенизация

```
# Импорт библиотек для глубокого обучения
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

print("Настраиваем Simple RNN для посимвольной генерации текста...")

# 1. Подготовка алфавита
chars = sorted(set(full_cleaned_text))
char2idx = {c: i for i, c in enumerate(chars)}
idx2char = {i: c for c, i in char2idx.items()}
vocab_size = len(chars)

print(f"Размер словаря символов: {vocab_size}")
print(f"Первые 20 символов: {chars[:20]}")

# 2. Создание обучающих данных
seq_length = 40  # Длина входной последовательности
step = 3         # Шаг скользящего окна
input_seqs = []
target_chars = []

print("Создаем обучающие последовательности...")

for i in range(0, len(full_cleaned_text) - seq_length, step):
    input_seq = full_cleaned_text[i:i + seq_length]
    target_char = full_cleaned_text[i + seq_length]
    input_seqs.append([char2idx[c] for c in input_seq])
    target_chars.append(char2idx[target_char])

print(f"Создано обучающих последовательностей: {len(input_seqs)}")

# 3. One-hot encoding
X = tf.keras.utils.to_categorical(input_seqs, num_classes=vocab_size)
y = tf.keras.utils.to_categorical(target_chars, num_classes=vocab_size)

print(f"Размер входных данных: {X.shape}")
print(f"Размер целевых данных: {y.shape}")

# 4. Создание модели Simple RNN
model = Sequential([
    SimpleRNN(128, input_shape=(seq_length, vocab_size)),
    Dense(vocab_size, activation='softmax')
])

print("Архитектура модели:")
model.summary()

# Компиляция модели
model.compile(loss='categorical_crossentropy', optimizer='adam')
print("Модель скомпилирована и готова к обучению!")
```

```
Настраиваем Simple RNN для посимвольной генерации текста...
Размер словаря символов: 39
Первые 20 символов: [' ', '!', ',', '.', ';', '?', 'а', 'б', 'в', 'г', 'д', 'е',
Создаем обучающие последовательности...
Создано обучающих последовательностей: 65869
Размер входных данных: (65869, 40, 39)
Размер целевых данных: (65869, 39)
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWar
  super().__init__(**kwargs)
Архитектура модели:
```

Model: "sequential"

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 128)	21,504
dense (Dense)	(None, 39)	5,031

Total params: 26,535 (103.65 KB)
Trainable params: 26,535 (103.65 KB)
Non-trainable params: 0 (0.00 B)

```
Модель скомпилирована и готова к обучению!
```

```
# Обучение модели Simple RNN
print("Начинаем обучение Simple RNN...")
print("Это может занять некоторое время...")

# Обучение модели
model.fit(X, y, batch_size=128, epochs=200)

print("Обучение завершено!")
```

```
Начинаем обучение Simple RNN...
Это может занять некоторое время...
Epoch 1/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 13s 15ms/step - loss: 3.0089
Epoch 2/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.4733
Epoch 3/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 12s 19ms/step - loss: 2.3618
Epoch 4/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 2.2876
Epoch 5/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.2365
Epoch 6/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.2004
Epoch 7/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.1642
Epoch 8/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 2.1329
Epoch 9/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0986
Epoch 10/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0695
Epoch 11/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0457
```

```
Epoch 12/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0183
Epoch 13/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.9927
Epoch 14/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.9733
Epoch 15/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.9445
Epoch 16/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.9163
Epoch 17/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8946
Epoch 18/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8760
Epoch 19/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 1.8580
Epoch 20/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8397
Epoch 21/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 1.8335
Epoch 22/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8037
Epoch 23/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 1.8011
Epoch 24/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 1.7797
Epoch 25/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 1.7821
Epoch 26/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.7603
Epoch 27/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 1.7500
Epoch 28/200
515/515 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 1.7338
```

```
def generate_text(seed_text, gen_length=300):
    """
    Функция для генерации текста с помощью обученной модели

    Args:
        seed_text (str): Начальный текст для генерации
        gen_length (int): Длина генерируемого текста

    Returns:
        str: Сгенерированный текст
    """
    generated = seed_text

    for _ in range(gen_length):
        # Подготавливаем входную последовательность
        input_seq = [char2idx.get(c, 0) for c in generated[-seq_length:]]
        input_seq = tf.keras.utils.to_categorical(input_seq, num_classes=vocab_
        input_seq = np.expand_dims(input_seq, axis=0)  # Добавляем размерность

        # Получаем предсказание следующего символа
        prediction = model.predict(input_seq, verbose=0)[0]

        # Выбираем следующий символ на основе вероятностей
```

```
            next_index = np.random.choice(range(vocab_size), p=prediction)
            next_char = idx2char[next_index]

            generated += next_char

        return generated

    # Тестируем генерацию текста
    print("Генерируем текст на основе слова 'язык':")
    generated_text1 = generate_text("язык")
    print(generated_text1)

    print("\nГенерируем текст на основе слова 'зелье':")
    generated_text2 = generate_text("зелье")
    print(generated_text2)
```

```
Генерируем текст на основе слова 'язык':
языкая песпетенное падирмания овретевелшей честв. котово мести. гарри она дрого

Генерируем текст на основе слова 'зелье':
зельену, и они не заметиленный деть гонагичествое правиль я подаждай. я течее де
```

Пословная токенизация

```
# Настройка Simple RNN для пословной генерации
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

print("Настраиваем Simple RNN для пословной генерации...")

# 1. Подготовка: токенизация текста на слова
tokenizer = Tokenizer()
tokenizer.fit_on_texts([full_cleaned_text])

word_index = tokenizer.word_index          # Слово → индекс
index_word = {v: k for k, v in word_index.items()}  # Индекс → слово
total_words = len(word_index) + 1          # Добавляем 1 для padding

print(f"Всего уникальных слов: {total_words}")
print(f"Первые 10 слов в словаре: {list(word_index.keys())[:10]}")

# 2. Создание обучающих последовательностей (n-граммы)
input_sequences = []
words = full_cleaned_text.split()

# Скользящее окно для генерации n-грамм
window_size = 5
for i in range(1, len(words)):
    seq = words[max(0, i - window_size):i + 1]
    encoded = tokenizer.texts_to_sequences([" ".join(seq)])[0]
    if len(encoded) >= 2:
```

```
        input_sequences.append(encoded)

print(f"Количество обучающих последовательностей: {len(input_sequences)}")

# 3. Подготовка X и y
max_seq_len = max(len(seq) for seq in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_seq_len)

X_words = input_sequences[:, :-1]  # все слова, кроме последнего
y_words = tf.keras.utils.to_categorical(input_sequences[:, -1], num_classes=tot

print(f"Размер входных данных: {X_words.shape}")
print(f"Размер целевых данных: {y_words.shape}")

# 4. Построение модели Simple RNN для слов
model_word = Sequential([
    tf.keras.layers.Embedding(input_dim=total_words, output_dim=64, input_lengt
    SimpleRNN(128),
    Dense(total_words, activation='softmax')
])

model_word.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[

print("Архитектура модели для слов:")
model_word.summary()
```

```
Настраиваем Simple RNN для пословной генерации...
Всего уникальных слов: 8179
Первые 10 слов в словаре: ['и', 'в', 'не', 'что', 'гарри', 'я', 'на', 'с', 'но',
Количество обучающих последовательностей: 30340
Размер входных данных: (30340, 7)
Размер целевых данных: (30340, 8179)
Архитектура модели для слов:
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: U
  warnings.warn(
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
simple_rnn_1 (SimpleRNN)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

```
Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)
```

Часть 2: LSTM (Long Short-Term Memory)

Однонаправленная LSTM

Посимвольная токенизация с LSTM

```
# Импорт библиотек для LSTM
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.utils import to_categorical

print("Настраиваем LSTM для посимвольной генерации...")

# Подготовка данных (аналогично Simple RNN)
chars = sorted(set(full_cleaned_text))
char2idx = {c: i for i, c in enumerate(chars)}
idx2char = {i: c for c, i in char2idx.items()}
vocab_size = len(chars)

print(f"Размер словаря символов: {vocab_size}")

# Создание последовательностей
seq_length = 40
step = 3
sequences = []
next_chars = []

for i in range(0, len(full_cleaned_text) - seq_length, step):
    seq = full_cleaned_text[i:i+seq_length]
    target = full_cleaned_text[i+seq_length]
    sequences.append([char2idx[c] for c in seq])
    next_chars.append(char2idx[target])

# Преобразование в one-hot encoding
X = to_categorical(sequences, num_classes=vocab_size)
y = to_categorical(next_chars, num_classes=vocab_size)

print(f"Размер обучающих данных: {X.shape}")
print(f"Размер целевых данных: {y.shape}")

# Создание модели LSTM
model_lstm_1 = Sequential([
    LSTM(128, input_shape=(seq_length, vocab_size)),
    Dense(vocab_size, activation='softmax')
])

model_lstm_1.compile(loss='categorical_crossentropy', optimizer='adam')

print("Архитектура однослойной LSTM:")
model_lstm_1.summary()
```

```
Настраиваем LSTM для посимвольной генерации...
Размер словаря символов: 39
Размер обучающих данных: (65869, 40, 39)
Размер целевых данных: (65869, 39)
Архитектура однослойной LSTM:
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 128)	86,016
dense_2 (Dense)	(None, 39)	5,031

```
Total params: 91,047 (355.65 KB)
Trainable params: 91,047 (355.65 KB)
Non-trainable params: 0 (0.00 B)
```

```
# Обучение однослойной LSTM
print("Начинаем обучение однослойной LSTM...")

model_lstm_1.fit(X, y, batch_size=128, epochs=50)

print("Обучение однослойной LSTM завершено!")
```

```
Начинаем обучение однослойной LSTM...
Epoch 1/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 3.0802
Epoch 2/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.5404
Epoch 3/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 2.4001
Epoch 4/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.3208
Epoch 5/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.2495
Epoch 6/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.1821
Epoch 7/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 2.1276
Epoch 8/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0768
Epoch 9/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 2.0308
Epoch 10/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.9885
Epoch 11/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.9450
Epoch 12/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.9141
Epoch 13/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8659
Epoch 14/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.8337
Epoch 15/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.7898
Epoch 16/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.7605
```

```
Epoch 17/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.7363
Epoch 18/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.6986
Epoch 19/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.6701
Epoch 20/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.6411
Epoch 21/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.6147
Epoch 22/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.5916
Epoch 23/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.5727
Epoch 24/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.5360
Epoch 25/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 1.5110
Epoch 26/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.4737
Epoch 27/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.4594
Epoch 28/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 1.4226
Epoch 29/50
```

```
def generate_text_char_lstm(model, seed_text="гарри", length=20, temperature=1.
    """
    Функция генерации текста с использованием LSTM

    Args:
        model: Обученная LSTM модель
        seed_text (str): Начальный текст
        length (int): Длина генерируемого текста
        temperature (float): Температура для сэмплирования

    Returns:
        str: Сгенерированный текст
    """
    generated = seed_text

    for _ in range(length):
        # Подготавливаем входную последовательность
        input_seq = [char2idx.get(c, 0) for c in generated[-seq_length:]]
        input_seq = tf.keras.utils.to_categorical(input_seq, num_classes=vocab_
        input_seq = np.expand_dims(input_seq, axis=0)

        # Получаем предсказания
        preds = model.predict(input_seq, verbose=0)[0]
        preds = np.asarray(preds).astype("float64")

        # Применяем температуру для контроля случайности
        preds = np.log(preds + 1e-8) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)

        # Выбираем следующий символ
```

```
            next_idx = np.random.choice(range(vocab_size), p=preds)
            next_char = idx2char[next_idx]
            generated += next_char

    return generated

# Тестируем генерацию с однослойной LSTM
print("Генерация текста с однослойной LSTM:")
print("Начальное слово: 'гермиона'")
generated_text = generate_text_char_lstm(model_lstm_1, "гермиона")
print(generated_text)

print("\nГенерация с другим начальным словом:")
print("Начальное слово: 'пожиратели'")
generated_text2 = generate_text_char_lstm(model_lstm_1, "пожиратели")
print(generated_text2)
```

```
Генерация текста с однослойной LSTM:
Начальное слово: 'гермиона'
гермиона урмух лискайком пер

Генерация с другим начальным словом:
Начальное слово: 'пожиратели'
пожирались мальнаго. гляна и
```

Многослойная LSTM

```
# Создание многослойной LSTM модели
print("Создаем многослойную LSTM модель...")

model_lstm_multi = Sequential([
    LSTM(128, return_sequences=True, input_shape=(seq_length, vocab_size)),
    LSTM(128),
    Dense(vocab_size, activation='softmax')
])

model_lstm_multi.compile(loss='categorical_crossentropy', optimizer='adam')

print("Архитектура многослойной LSTM:")
model_lstm_multi.summary()

# Обучение многослойной LSTM
print("Начинаем обучение многослойной LSTM...")
model_lstm_multi.fit(X, y, batch_size=128, epochs=50)

print("Обучение многослойной LSTM завершено!")

# Тестируем многослойную LSTM
print("\nГенерация текста с многослойной LSTM:")
print("Начальное слово: 'драко'")
generated_multi = generate_text_char_lstm(model_lstm_multi, "драко")
print(generated_multi)
```

```
print("\nГенерация с другим начальным словом:")
print("Начальное слово: 'поттер'")
generated_multi2 = generate_text_char_lstm(model_lstm_multi, "поттер")
print(generated_multi2)
```

```
Создаем многослойную LSTM модель...
Архитектура многослойной LSTM:
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 40, 128)	86,016
lstm_2 (LSTM)	(None, 128)	131,584
dense_3 (Dense)	(None, 39)	5,031

```
 Total params: 222,631 (869.65 KB)
 Trainable params: 222,631 (869.65 KB)
 Non-trainable params: 0 (0.00 B)
Начинаем обучение многослойной LSTM...
Epoch 1/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 3.1325
Epoch 2/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 2.5796
Epoch 3/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.4132
Epoch 4/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.3207
Epoch 5/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 2.2576
Epoch 6/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.1831
Epoch 7/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.1371
Epoch 8/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 2.1054
Epoch 9/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.0593
Epoch 10/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.0218
Epoch 11/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 1.9801
Epoch 12/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.9405
Epoch 13/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.9013
Epoch 14/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 1.8673
Epoch 15/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.8291
Epoch 16/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.8014
Epoch 17/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.7553
Epoch 18/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.7164
Epoch 19/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 1.6812
Epoch 20/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.6639
Epoch 21/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.6412
Epoch 22/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - loss: 1.8917
```

```
Epoch 23/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 1.7900
Epoch 24/50
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 1.6056
```

Пословная токенизация с LSTM

```
# Настройка LSTM для пословной генерации
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

print("Настраиваем LSTM для пословной генерации...")

# Токенизация слов
tokenizer = Tokenizer()
tokenizer.fit_on_texts([full_cleaned_text])
word_index = tokenizer.word_index
index_word = {v: k for k, v in word_index.items()}
total_words = len(word_index) + 1

print(f"Общий словарь слов: {total_words}")

# Создание последовательностей слов
input_sequences = []
words = full_cleaned_text.split()

for i in range(1, len(words)):
    ngram = words[max(0, i-5):i+1]
    encoded = tokenizer.texts_to_sequences([" ".join(ngram)])[0]
    if len(encoded) >= 2:
        input_sequences.append(encoded)

max_len = max(len(seq) for seq in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_len)

X_words = input_sequences[:, :-1]
y_words = to_categorical(input_sequences[:, -1], num_classes=total_words)

print(f"Размер обучающих данных: {X_words.shape}")
print(f"Размер целевых данных: {y_words.shape}")

# Создание однослойной LSTM для слов
model_words_1 = Sequential([
    tf.keras.layers.Embedding(total_words, 64, input_length=X_words.shape[1]),
    LSTM(128),
    Dense(total_words, activation='softmax')
])

model_words_1.compile(loss='categorical_crossentropy', optimizer='adam')

print("Архитектура однослойной LSTM для слов:")
model_words_1.summary()

# Обучение модели
print("Начинаем обучение однослойной LSTM для слов...")
model_words_1.fit(X_words, y_words, epochs=100, batch_size=128)
```

```
print("Обучение завершено!")
```

```
начальное слово: ˙поттер˙
поттереет из такую вы едиц
```

```
Настраиваем LSTM для пословной генерации...
Общий словарь слов: 8179
Размер обучающих данных: (30340, 7)
Размер целевых данных: (30340, 8179)
Архитектура однослойной LSTM для слов:
Model: "sequential_4"
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	0 (unbuilt)
lstm_3 (LSTM)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)

```
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Начинаем обучение однослойной LSTM для слов...
Epoch 1/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 8.2286
Epoch 2/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 7.4128
Epoch 3/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 7.2821
Epoch 4/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 7.1974
Epoch 5/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 7.1221
Epoch 6/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 7.0313
Epoch 7/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 6.8700
Epoch 8/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 6.7069
Epoch 9/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 6.5347
Epoch 10/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 6.3587
Epoch 11/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 6.1644
Epoch 12/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 5.9806
Epoch 13/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 5.8078
Epoch 14/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 5.6344
Epoch 15/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 5.4378
Epoch 16/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 5.2687
Epoch 17/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 5.1023
Epoch 18/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 4.9197
Epoch 19/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 4.7649
Epoch 20/100
```

```
def generate_text_word_lstm(model, seed_text, num_words=10, temperature=1.0):
    """
    Функция генерации текста по словам с использованием LSTM

    Args:
        model: Обученная LSTM модель
        seed_text (str): Начальный текст
        num_words (int): Количество слов для генерации
        temperature (float): Температура для сэмплирования

    Returns:
        str: Сгенерированный текст
    """
    for _ in range(num_words):
        # Токенизируем текущий текст
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=X_words.shape[1])

        # Получаем предсказания
        preds = model.predict(token_list, verbose=0)[0]
        preds = np.log(preds + 1e-8) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)

        # Выбираем следующее слово
        next_index = np.random.choice(range(total_words), p=preds)
        next_word = index_word.get(next_index, "")
        seed_text += " " + next_word

    return seed_text

# Тестируем генерацию слов
print("Генерация текста по словам с LSTM:")
print("Начальная фраза: 'волшебная палочка'")
generated_words = generate_text_word_lstm(model_words_1, "волшебная палочка")
print(generated_words)

print("\nГенерация с другой фразой:")
print("Начальная фраза: 'малфой'")
generated_words2 = generate_text_word_lstm(model_words_1, "малфой")
print(generated_words2)
```

```
Epoch 45/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 1.5376
Epoch 46/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 1.4707
Epoch 47/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 1.4140
Epoch 48/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 1.3486
Epoch 49/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 1.2966
Epoch 50/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 1.2250
Epoch 51/100
```

Генерация текста по словам с LSTM:
Начальная фраза: 'волшебная палочка'
волшебная палочка я не был вспомнить может быть развернулась ахава разговаривать

Генерация с другой фразой:
Начальная фраза: 'малфой'
малфой и ни мальчик кашлянул же его запнулась на случай мой

BPE (Byte Pair Encoding) токенизация

```
# Установка библиотеки tokenizers для BPE
import subprocess
import sys

try:
    from tokenizers import Tokenizer, models, trainers, pre_tokenizers
    print("Библиотека tokenizers уже установлена")
except ImportError:
    print("Устанавливаем библиотеку tokenizers...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "tokenizers"
    from tokenizers import Tokenizer, models, trainers, pre_tokenizers

from tensorflow.keras.utils import to_categorical

print("Настраиваем BPE токенизацию...")

# Создаем и обучаем BPE токенизатор
bpe_tokenizer = Tokenizer(models.BPE())
trainer = trainers.BpeTrainer(special_tokens=["<PAD>"])
bpe_tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()
bpe_tokenizer.train_from_iterator(full_cleaned_text.splitlines(), trainer)

print("BPE токенизатор обучен!")

# Создание последовательностей с BPE токенизацией
tokens = bpe_tokenizer.encode(full_cleaned_text).ids
seq_len = 40
X_bpe, y_bpe = [], []

for i in range(seq_len, len(tokens)):
    X_bpe.append(tokens[i-seq_len:i])
    y_bpe.append(tokens[i])

vocab_size_bpe = bpe_tokenizer.get_vocab_size()

X_bpe = np.array(X_bpe)
y_bpe = to_categorical(y_bpe, num_classes=vocab_size_bpe)

print(f"Размер словаря BPE: {vocab_size_bpe}")
print(f"Размер обучающих данных: {X_bpe.shape}")
print(f"Размер целевых данных: {y_bpe.shape}")
```

```
Epoch 75/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.3724
Epoch 76/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 0.3573
Epoch 77/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 0.3349
Epoch 78/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 0.3291
Epoch 79/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - loss: 0.3037
Epoch 80/100
238/238 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - loss: 0.2833
```

Библиотека tokenizers уже установлена
Настраиваем BPE токенизацию...
BPE токенизатор обучен!
Размер словаря BPE: 13265
Размер обучающих данных: (36487, 40)
Размер целевых данных: (36487, 13265)

Однослойная LSTM с BPE

```
# Создание однослойной LSTM модели с BPE
model_bpe_1 = Sequential([
    tf.keras.layers.Embedding(vocab_size_bpe, 64, input_length=seq_len),
    LSTM(128),
    Dense(vocab_size_bpe, activation='softmax')
])

model_bpe_1.compile(loss='categorical_crossentropy', optimizer='adam')

print("Архитектура однослойной LSTM с BPE:")
model_bpe_1.summary()

# Обучение модели
print("Начинаем обучение однослойной LSTM с BPE...")
model_bpe_1.fit(X_bpe, y_bpe, epochs=150, batch_size=128)

print("Обучение завершено!")

# Функция генерации с BPE
def generate_text_bpe_lstm(model, tokenizer_bpe, seed_text, gen_tokens=50, temp
    """
    Функция генерации текста с использованием BPE токенизации

    Args:
        model: Обученная модель
        tokenizer_bpe: BPE токенизатор
        seed_text (str): Начальный текст
        gen_tokens (int): Количество токенов для генерации
        temperature (float): Температура сэмплирования

    Returns:
        str: Сгенерированный текст
    """
    tokens = tokenizer_bpe.encode(seed_text).ids
    generated = tokens[:]

    for _ in range(gen_tokens):
        input_seq = generated[-seq_len:]
        input_seq = pad_sequences([input_seq], maxlen=seq_len)

        preds = model.predict(input_seq, verbose=0)[0]
        preds = np.log(preds + 1e-8) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)

        next_index = np.random.choice(range(vocab_size_bpe), p=preds)
        generated.append(next_index)

    decoded = tokenizer_bpe.decode(generated)
    return decoded

# Тестируем генерацию с BPE
print("Генерация текста с BPE токенизацией:")
print("Начальная фраза: 'заклинание против'")
generated_bpe = generate_text_bpe_lstm(model_bpe_1, bpe_tokenizer, "заклинание
```

```
print(generated_bpe)

print("\nГенерация с другой фразой:")
print("Начальная фраза: 'проклятые маглы'")
generated_bpe2 = generate_text_bpe_lstm(model_bpe_1, bpe_tokenizer, "проклятые
print(generated_bpe2)
```

```
Архитектура однослойной LSTM с BPE:
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	?	0 (unbuilt)
lstm_4 (LSTM)	?	0 (unbuilt)
dense_5 (Dense)	?	0 (unbuilt)

```
 Total params: 0 (0.00 B)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 0 (0.00 B)
Начинаем обучение однослойной LSTM с BPE...
Epoch 1/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - loss: 7.8886
Epoch 2/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 8s 22ms/step - loss: 6.7055
Epoch 3/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - loss: 6.4712
Epoch 4/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 6s 20ms/step - loss: 6.3003
Epoch 5/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 6.1378
Epoch 6/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 5s 16ms/step - loss: 5.9955
Epoch 7/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 7s 24ms/step - loss: 5.8899
Epoch 8/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - loss: 5.7438
Epoch 9/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 5.6149
Epoch 10/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 5.4954
Epoch 11/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 5.3711
Epoch 12/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 5.2643
Epoch 13/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - loss: 5.1454
Epoch 14/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 5.0087
Epoch 15/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 4.9093
Epoch 16/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 4.7802
Epoch 17/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 4.6569
Epoch 18/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 4.5816
Epoch 19/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 4.4441
Epoch 20/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 4.3297
Epoch 21/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 4.2124
Epoch 22/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 4.1000
Epoch 23/150
```

```
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 3.9800
Epoch 24/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 3.8653
Epoch 25/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 3.7510
Epoch 26/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 3.6696
Epoch 27/150
```

Часть 3: Двунаправленная LSTM

Посимвольная токенизация с Bidirectional LSTM

```
# Импорт библиотек для Bidirectional LSTM
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Bidirectional, Dense, Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical

print("Настраиваем Bidirectional LSTM для посимвольной генерации...")

# Подготовка данных (аналогично предыдущим моделям)
chars = sorted(set(full_cleaned_text))
char2idx = {c: i for i, c in enumerate(chars)}
idx2char = {i: c for c, i in char2idx.items()}
vocab_size = len(chars)

# Создание последовательностей
seq_length = 40
step = 3
input_seqs, target_chars = [], []

for i in range(0, len(full_cleaned_text) - seq_length, step):
    seq = full_cleaned_text[i:i + seq_length]
    target = full_cleaned_text[i + seq_length]
    input_seqs.append([char2idx[c] for c in seq])
    target_chars.append(char2idx[target])

X_char = tf.keras.utils.to_categorical(input_seqs, num_classes=vocab_size)
y_char = tf.keras.utils.to_categorical(target_chars, num_classes=vocab_size)

print(f"Размер обучающих данных: {X_char.shape}")
print(f"Размер целевых данных: {y_char.shape}")

# Создание модели Bidirectional LSTM
model_char = Sequential([
    Bidirectional(LSTM(128), input_shape=(seq_length, vocab_size)),
    Dense(vocab_size, activation='softmax')
])

model_char.compile(loss='categorical_crossentropy', optimizer='adam')

print("Архитектура Bidirectional LSTM:")
model_char.summary()

# Обучение модели
print("Начинаем обучение Bidirectional LSTM...")
```

```
model_char.fit(X_char, y_char, batch_size=128, epochs=50)

print("Обучение завершено!")

# Функция генерации для Bidirectional LSTM
def generate_char_text(model, seed_text="раppи", length=20, temperature=1.0):
    """
    Функция генерации текста с Bidirectional LSTM

    Args:
        model: Обученная модель
        seed_text (str): Начальный текст
        length (int): Длина генерируемого текста
        temperature (float): Температура сэмплирования

    Returns:
        str: Сгенерированный текст
    """
    generated = seed_text

    for _ in range(length):
        input_seq = [char2idx.get(c, 0) for c in generated[-seq_length:]]
        input_seq = tf.keras.utils.to_categorical(input_seq, num_classes=vocab_
        input_seq = np.expand_dims(input_seq, axis=0)

        preds = model.predict(input_seq, verbose=0)[0]
        preds = np.log(preds + 1e-8) / temperature
        preds = np.exp(preds) / np.sum(np.exp(preds))

        next_idx = np.random.choice(range(vocab_size), p=preds)
        next_char = idx2char[next_idx]
        generated += next_char

    return generated

# Тестируем генерацию
print("Генерация текста с Bidirectional LSTM:")
print("Начальное слово: 'заклинание'")
generated_char = generate_char_text(model_char, "заклинание", length=60)
print(generated_char)
```

```
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 1.2918
Epoch 77/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 1.2796
Epoch 78/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - loss: 1.2411
Epoch 79/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 1.2361
Epoch 80/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 1.2294
Epoch 81/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 1.1874
Epoch 82/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 1.1637
Epoch 83/150
286/286 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 1.1429
```

```
Настраиваем Bidirectional LSTM для посимвольной генерации...
Размер обучающих данных: (65869, 40, 39)
Размер целевых данных: (65869, 39)
Архитектура Bidirectional LSTM:
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/bidirectional.py:10
  super().__init__(**kwargs)
```

Model: "sequential_6"

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 256)	172,032
dense_6 (Dense)	(None, 39)	10,023

Total params: 182,055 (711.15 KB)

Trainable params: 182,055 (711.15 KB)

Non-trainable params: 0 (0.00 B)

```
Начинаем обучение Bidirectional LSTM...
Epoch 1/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 3.0902
Epoch 2/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.5454
Epoch 3/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.3861
Epoch 4/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 2.2891
Epoch 5/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.2190
Epoch 6/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 2.1429
Epoch 7/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.0844
Epoch 8/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 2.0300
Epoch 9/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 1.9808
Epoch 10/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.9353
Epoch 11/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.8955
Epoch 12/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.8209
Epoch 13/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.7797
Epoch 14/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 5s 9ms/step - loss: 1.7284
Epoch 15/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.6767
Epoch 16/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.6347
Epoch 17/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.5804
Epoch 18/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.5273
Epoch 19/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 6s 10ms/step - loss: 1.4784
Epoch 20/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.4397
Epoch 21/150
515/515 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - loss: 1.4002
```

Часть 4: GPT с нуля

Символьная GPT модель

```
# Импорт библиотек для Transformer архитектуры
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
import numpy as np

print("Создаем GPT-style модель с нуля...")

# Подготовка данных для символов
vocab = sorted(set(full_cleaned_text))
char2idx = {c: i for i, c in enumerate(vocab)}
idx2char = {i: c for c, i in char2idx.items()}
vocab_size = len(vocab)

print(f"Размер словаря символов: {vocab_size}")

# Преобразование в числовой вид
encoded = [char2idx[c] for c in full_cleaned_text]

# Создание обучающих пар
context_len = 64
X, y = [], []
for i in range(len(encoded) - context_len):
    X.append(encoded[i:i + context_len])
    y.append(encoded[i + context_len])

X = np.array(X)
y = to_categorical(y, num_classes=vocab_size)

print(f"Размер входных данных: {X.shape}")
print(f"Размер целевых данных: {y.shape}")

# Определение кастомных слоев для Transformer
from tensorflow.keras.layers import Layer, Dense, LayerNormalization
import tensorflow.keras.backend as K

class MaskedSelfAttention(Layer):
    """
    Слой маскированного self-attention для GPT архитектуры
    """
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.proj_q = Dense(embed_dim)
        self.proj_k = Dense(embed_dim)
```

```
        self.proj_v = Dense(embed_dim)
        self.out = Dense(embed_dim)

    def call(self, x):
        B, T, C = tf.shape(x)[0], tf.shape(x)[1], self.embed_dim
        q = self.proj_q(x)
        k = self.proj_k(x)
        v = self.proj_v(x)

        # Разделение на головы внимания
        q = tf.concat(tf.split(q, self.num_heads, axis=-1), axis=0)
        k = tf.concat(tf.split(k, self.num_heads, axis=-1), axis=0)
        v = tf.concat(tf.split(v, self.num_heads, axis=-1), axis=0)

        # Вычисление attention scores
        att = tf.matmul(q, k, transpose_b=True) / tf.math.sqrt(tf.cast(C // sel

        # Маскирование будущих позиций (causal mask)
        mask = tf.linalg.band_part(tf.ones((T, T)), -1, 0)
        att = tf.where(mask == 0, -1e10, att)

        att = tf.nn.softmax(att, axis=-1)
        out = tf.matmul(att, v)
        out = tf.concat(tf.split(out, self.num_heads, axis=0), axis=-1)
        return self.out(out)

class TransformerBlock(Layer):
    """
    Блок Transformer с self-attention и feed-forward сетью
    """
    def __init__(self, embed_dim, num_heads, ff_dim):
        super().__init__()
        self.att = MaskedSelfAttention(embed_dim, num_heads)
        self.norm1 = LayerNormalization()
        self.ff = tf.keras.Sequential([
            Dense(ff_dim, activation='relu'),
            Dense(embed_dim)
        ])
        self.norm2 = LayerNormalization()

    def call(self, x):
        # Self-attention с residual connection
        x = x + self.att(self.norm1(x))
        # Feed-forward с residual connection
        x = x + self.ff(self.norm2(x))
        return x

class PositionalEncoding(Layer):
    """
    Позиционное кодирование для Transformer
    """
    def __init__(self, max_len, embed_dim):
        super().__init__()
        pos = np.arange(max_len)[:, None]
        i = np.arange(embed_dim)[None, :]
```

```
        angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(embed_dim))
        angle_rads = pos * angle_rates
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        self.pos_encoding = tf.constant(angle_rads[np.newaxis, ...], dtype=tf.f

    def call(self, x):
        return x + self.pos_encoding[:, :tf.shape(x)[1], :]

print("Кастомные слои Transformer определены!")
```

```
Создаем GPT-style модель с нуля...
Размер словаря символов: 39
Размер входных данных: (197581, 64)
Размер целевых данных: (197581, 39)
Кастомные слои Transformer определены!
```

```
# Создание GPT модели
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding

# Параметры модели
embed_dim = 128
num_heads = 4
ff_dim = 256
num_layers = 4

print(f"Создаем GPT модель с параметрами:")
print(f"- Размерность эмбеддинга: {embed_dim}")
print(f"- Количество голов внимания: {num_heads}")
print(f"- Размерность feed-forward: {ff_dim}")
print(f"- Количество слоев: {num_layers}")

# Создание модели
inp = Input(shape=(context_len,))
x = Embedding(vocab_size, embed_dim)(inp)
x = PositionalEncoding(context_len, embed_dim)(x)

# Добавляем блоки Transformer
for i in range(num_layers):
    x = TransformerBlock(embed_dim, num_heads, ff_dim)(x)

# Выходной слой для предсказания следующего токена
out = Dense(vocab_size, activation="softmax")(x[:, -1, :])

gpt_model = Model(inputs=inp, outputs=out)
gpt_model.compile(loss="categorical_crossentropy", optimizer="adam")

print("Архитектура GPT модели:")
gpt_model.summary()

# Обучение модели
print("Начинаем обучение GPT модели...")
```

```
print("Внимание: Обучение может занять значительное время!")

gpt_model.fit(X, y, batch_size=128, epochs=10)

print("Обучение GPT модели завершено!")
```

```
Создаем GPT модель с параметрами:
- Размерность эмбеддинга: 128
- Количество голов внимания: 4
- Размерность feed-forward: 256
- Количество слоев: 4
Архитектура GPT модели:
```

Model: "functional_10"

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 64)	0
embedding_3 (Embedding)	(None, 64, 128)	4,992
positional_encoding (PositionalEncoding)	(None, 64, 128)	0
transformer_block (TransformerBlock)	(None, 64, 128)	132,480
transformer_block_1 (TransformerBlock)	(None, 64, 128)	132,480
transformer_block_2 (TransformerBlock)	(None, 64, 128)	132,480
transformer_block_3 (TransformerBlock)	(None, 64, 128)	132,480
get_item (GetItem)	(None, 128)	0
dense_31 (Dense)	(None, 39)	5,031

Total params: 539,943 (2.06 MB)
Trainable params: 539,943 (2.06 MB)
Non-trainable params: 0 (0.00 B)

```
Начинаем обучение GPT модели...
Внимание: Обучение может занять значительное время!
Epoch 1/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 52s 24ms/step - loss: 2.9251
Epoch 2/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 2.1664
Epoch 3/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.9115
Epoch 4/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.7795
Epoch 5/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.6979
Epoch 6/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 20ms/step - loss: 1.6402
Epoch 7/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.5875
Epoch 8/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.5468
Epoch 9/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 20ms/step - loss: 1.5110
Epoch 10/10
1544/1544 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - loss: 1.4819
Обучение GPT модели завершено!
```

Напишите программный код или сгенерируйте его с помощью искусственного интеллекта.

```
def generate_gpt(seed_text, length=300, temperature=1.0):
    """
    Функция генерации текста с помощью GPT модели

    Args:
        seed_text (str): Начальный текст
        length (int): Длина генерируемого текста
        temperature (float): Температура сэмплирования

    Returns:
        str: Сгенерированный текст
    """
    generated = seed_text
    context = [char2idx.get(c, 0) for c in seed_text][-context_len:]

    for _ in range(length):
        # Подготавливаем входную последовательность
        padded = pad_sequences([context], maxlen=context_len)
        preds = gpt_model.predict(padded, verbose=0)[0]

        # Безопасное масштабирование вероятностей
        preds = np.asarray(preds).astype("float64")
        preds = np.log(np.clip(preds, 1e-8, 1.0)) / temperature
        preds = np.exp(preds)
        preds = preds / np.sum(preds)

        # Выбираем следующий символ
        next_idx = np.random.choice(len(preds), p=preds)
        next_char = idx2char[next_idx]

        generated += next_char
        context.append(next_idx)
        context = context[-context_len:]  # Сохраняем только последние context_

    return generated

# Тестируем генерацию с GPT
print("Генерация текста с GPT моделью:")
print("Начальное слово: 'гарри'")
generated_gpt = generate_gpt("гарри ", 300, temperature=1.0)
print(generated_gpt)

print("\nГенерация с другим начальным словом:")
print("Начальное слово: 'гарри' (короткая генерация)")
generated_gpt_short = generate_gpt("гарри ", 40)
print(generated_gpt_short)
```

```
Генерация текста с GPT моделью:
Начальное слово: 'гарри'
гарри с этинируев их гермионё чекала макгонагалл бынеи под серьёзно хорошо её по
```

```
Генерация с другим начальным словом:
Начальное слово: 'гарри' (короткая генерация)
гарри мысего она родить говориру! называет ту,
```

Дообучение готовой GPT модели

```
# Установка библиотек для дообучения
try:
    from datasets import Dataset
    from transformers import GPT2Tokenizer, GPT2LMHeadModel, TrainingArguments,
    print("Библиотеки transformers и datasets уже установлены")
except ImportError:
    print("Устанавливаем необходимые библиотеки...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "transformer
    from datasets import Dataset
    from transformers import GPT2Tokenizer, GPT2LMHeadModel, TrainingArguments,

print("Настраиваем дообучение готовой GPT модели...")

# Создание датасета из наших данных
dataset = Dataset.from_pandas(df[["text"]])

print(f"Создан датасет с {len(dataset)} записями")

# Загрузка предобученного токенизатора для русского языка
tokenizer = GPT2Tokenizer.from_pretrained("sberbank-ai/rugpt3small_based_on_gpt
tokenizer.pad_token = tokenizer.eos_token
```