Прерид к лабораторной 1: токены и специальные токены

После этого прерида вы сможете объяснить, что такое токен; загрузить готовый токенизатор; получить токены и их идентификаторы; работать со специальными токенами, парами текстов, padding и масками.

Материалы с пометкой «Для любознательных» необязательны: всё необходимое для лабораторной есть в основном тексте.

1. Что происходит после отправки сообщения

Вы пишете модели сообщение:

```
Объясни, почему небо голубое.
```

На экране это обычная строка текста. Но прежде чем сообщение попадёт в нейросеть, оно проходит промежуточное преобразование.

СТУДЕНТ: Зачем преобразовывать текст? Разве модель не может работать с ним напрямую?

МОДЕЛЬ: Нейросеть выполняет вычисления над числами, поэтому текст нужно представить в числовом виде.

Конечно, можно было бы сопоставить число каждому отдельному символу или байту - такие модели существуют. Однако изолированного символа обычно недостаточно, чтобы распознать слово или понятие: значение возникает из сочетания символов. Кроме того, посимвольное представление делает последовательность очень длинной, и модели приходится обрабатывать больше позиций.

Поэтому большинство современных языковых моделей используют промежуточный масштаб: текст делят на повторяющиеся фрагменты - токены. Затем каждому токену сопоставляют целочисленный идентификатор, или `token_id`.

Токен не обязан быть осмысленной частью слова с точки зрения лингвистики. Токенизатор ищет удобные для вычислений повторяющиеся фрагменты, а смысл модель строит из их сочетаний и контекста.

```
текст → токены → token_id → эмбеддинги → нейросеть
```

Условно это можно представить так:

```
«Я пишу код!» → [«Я», « пишу», « код», «!»] → [103, 812, 4501, 0]
```

Реальные границы токенов и номера определяет конкретный токенизатор.

> *Для любознательных. В видео 3Blue1Brown «Transformers, the tech behind LLMs» визуально показан дальнейший путь данных: от текстовых фрагментов и эмбеддингов до вероятностей следующего токена. Для видео доступна русская звуковая дорожка. Более подробное графическое объяснение архитектуры можно найти в материале Джея Аламмара The Illustrated Transformer.*

> *Для любознательных. В статье журнала «Код» «Вся база про токены в нейросетях: что это и как их эффективно использовать» понятие токена разобрано на наглядных русскоязычных примерах. Особенно полезны разделы о границах токенов, числовых идентификаторах и различиях между токенизацией русского и английского текста.*

Один токен не обязательно совпадает с одним словом. Токеном может оказаться:

- целое слово;
- часть слова;
- знак пунктуации;
- пробел вместе со следующим фрагментом;
- байт или последовательность байтов;
- специальный служебный элемент.

Например, WordPiece-токенизатор может представить слово так:

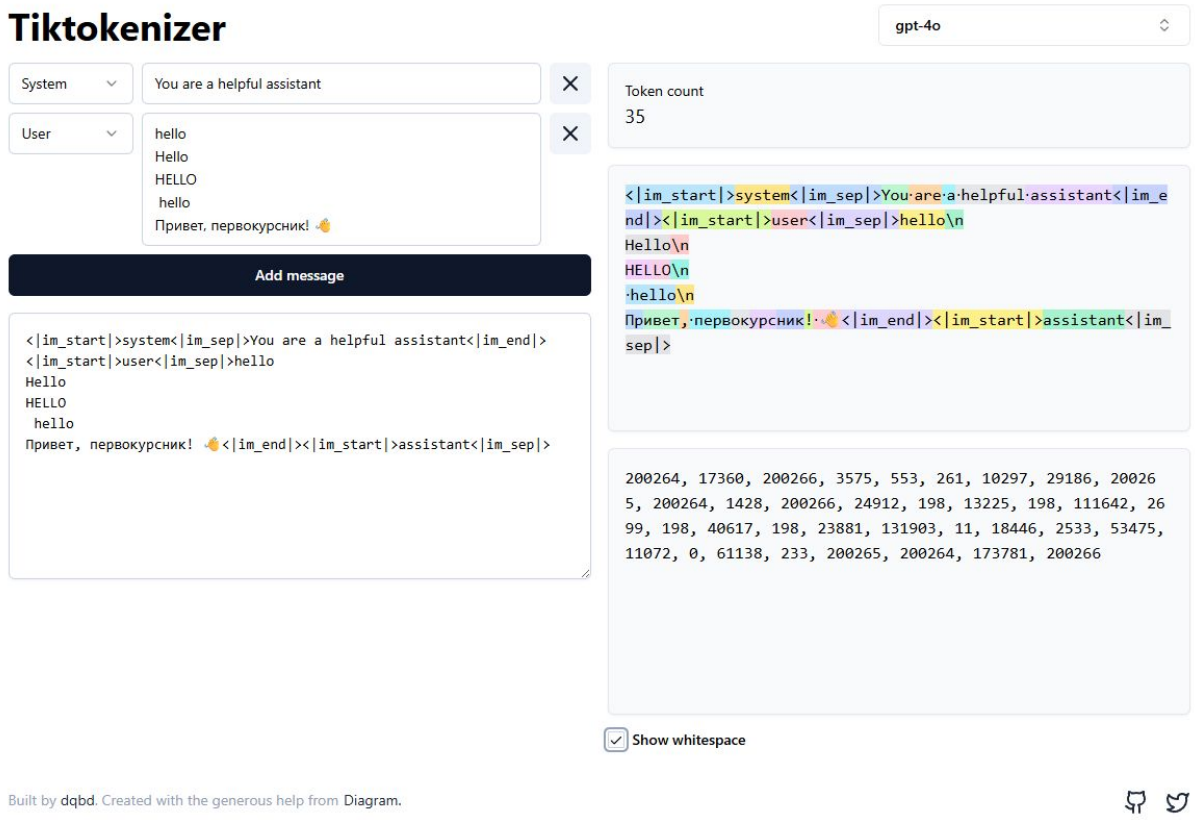
```
первокурсник → пер | ##вок | ##урс | ##ник
```

Префикс `##` означает, что фрагмент продолжает уже начатое слово. Символов `##` в исходном тексте не было.

Поэтому количество слов, символов и токенов - три разных величины. Подсчёт слов через `text.split()` не заменяет токенизацию.

Посмотрите на границы токенов

На сайте Tiktokenizer токены подсвечиваются разными цветами, а режим Show whitespace делает пробелы видимыми. На скриншоте почти одинаковые строки `hello`, `Hello`, `HELLO` и `hello` разбились по-разному. Кроме пользовательского текста в представлении видны системное сообщение, роли и служебные маркеры чата.



На скриншоте выбрана кодировка модели `gpt-4o`, поэтому получившееся число нельзя переносить на BERT или другую модель. Этот инструмент удобен для визуального исследования, а точный расчёт всегда выполняют токенизатором нужной модели.

Словарь из всех слов был бы огромным и всё равно не охватил бы фамилии, опечатки, новые термины, формы слов, код, ссылки и смешение языков. Подсловная токенизация собирает редкие слова из повторяющихся фрагментов. Это компромисс:

- крупные токены сокращают последовательность, но требуют большого словаря;
- мелкие токены уменьшают словарь, но удлиняют последовательность.

2. Почему токенизаторы дают разные результаты

Границы определяет токенизатор, подготовленный для конкретной модели. В нём зафиксированы словарь, нормализация текста, предварительное разбиение, алгоритм токенизации, постобработка и специальные токены.

Часто встречаются алгоритмы BPE, WordPiece и Unigram. SentencePiece - это система токенизации, в которой, в частности, могут использоваться Unigram или BPE. У разных токенизаторов отличаются словарь и правила, поэтому одна строка может превратиться в разное число токенов.

> *Для любознательных. В документации Hugging Face отдельно разобран весь конвейер токенизации: нормализация, предварительное разбиение, модель*

> *токенизации и постобработка. Исторические и технические подробности можно найти в статьях о применении BPE к подсловной токенизации и о SentencePiece.*

Для запуска модели используют совместимый с ней токенизатор. Нельзя взять токенизатор BERT, посчитать им запрос для другой модели и считать результат точным.

`token_id` - это индекс элемента словаря. Он указывает, какую строку таблицы эмбеддингов взять, но сам по себе не является «смыслом» токена. Близкие номера также не обязаны соответствовать близким значениям.

3. Библиотека `tokenizers`

В лабораторной используется Python-библиотека Hugging Face Tokenizers. Примеры ниже проверены с последней стабильной версией ветки 0.x на дату подготовки материала:

```
python -m pip install "tokenizers==0.23.3"
```

На GitHub уже развивается предварительная версия 1.0, API которой частично отличается. Для воспроизводимости лабораторной используйте указанную стабильную версию, если преподаватель не сообщил другую.

Готовый токенизатор можно получить двумя способами.

Загрузка с Hugging Face Hub

`Tokenizer.from_pretrained(...)` принимает идентификатор репозитория, в котором есть совместимый `tokenizer.json`:

```
from tokenizers import Tokenizer

tokenizer = Tokenizer.from_pretrained("bert-base-multilingual-
cased")
```

При первом запуске файл скачивается, затем может браться из локального кэша. Если репозиторий закрыт, удалён, недоступен по сети или не содержит `tokenizer.json`, загрузка завершится ошибкой.

Загрузка локального файла

Если преподаватель выдал готовый файл, сеть не нужна:

```
from pathlib import Path
from tokenizers import Tokenizer

path = Path("tokenizers") / "bert_multilingual" / "tokenizer.json"

if not path.exists():
    raise FileNotFoundError(f"Нет файла: {path}")
```

```
tokenizer = Tokenizer.from_file(str(path))
```

В обоих случаях получается объект `Tokenizer`, с которым дальше работают одинаково. Не редактируйте `tokenizer.json`: словарь и правила токенизации должны оставаться согласованными с моделью.

4. Первое кодирование

Возьмём многоязычный BERT с WordPiece-токенизацией:

```
from tokenizers import Tokenizer

tokenizer = Tokenizer.from_pretrained("bert-base-multilingual-
cased")
text = "Привет, первокурсник! "
encoding = tokenizer.encode(text)

print(encoding.ids)
print(encoding.tokens)
print("Число токенов:", len(encoding.ids))
print("Декодирование:", tokenizer.decode(encoding.ids))

for token_id, token in zip(encoding.ids, encoding.tokens):
    print(token_id, repr(token))
```

Для `tokenizers==0.23.3` результат имеет следующий вид:

```
[101, 14337, 41102, 117, 61381, 72578, 77072, 11718, 106, 100, 102]
['[CLS]', 'При', '##вет', ',', 'пер', '##вок', '##урс', '##ник',
'!', '[UNK]', '[SEP]']
Число токенов: 11
Декодирование: Привет, первокурсник!
```

Здесь:

- `encoding.ids` - идентификаторы токенов;
- `encoding.tokens` - строковое представление тех же токенов;
- `[CLS]` и `[SEP]` добавлены постпроцессором токенизатора;
- эмодзи отсутствует в словаре и превратился в `[UNK]`;
- `decode` по умолчанию пропускает специальные токены, поэтому `[CLS]`, `[UNK]` и `[SEP]` не попали в восстановленную строку.

Декодирование не всегда восстанавливает исходную строку посимвольно. Нормализация, неизвестные фрагменты и правила работы с пробелами могут сделать преобразование необратимым.

Небольшой эксперимент с границами токенов

Изменим только регистр, пробел или пунктуацию:

```
examples = ["hello", "Hello", "HELLO", " hello", "hello!"]

for example in examples:
    encoding = tokenizer.encode(example, add_special_tokens=False)
    print(repr(example), len(encoding.ids), encoding.tokens)
```

Для `bert-base-multilingual-cased` результат будет таким:

```
'hello'  2  ['hell', '##o']
'Hello'  1  ['Hello']
'HELLO'  4  ['H', '##EL', '##L', '##O']
' hello' 2  ['hell', '##o']
'hello!' 3  ['hell', '##o', '!']
```

Регистр и пунктуация изменили результат, а начальный пробел у этого токенизатора - нет: предварительное разбиение BERT отделяет текст по пробельным границам. У другой модели пробел может войти в состав токена, как было видно в Tiktokenizer. Даже наблюдение о пробелах нельзя без проверки переносить между токенизаторами.

Русский и английский: какой текст «дороже»?

Сравним две инструкции одного назначения:

```
texts = [
    "Write a Python function that returns the sum of two numbers.",
    "Напиши функцию на Python, которая возвращает сумму двух чисел.
",
]

for text in texts:
    encoding = tokenizer.encode(text, add_special_tokens=False)
    print(len(encoding.ids), text)
```

Многоязычный BERT даёт 13 токенов для английского варианта и 19 для русского. `FacebookAI/xlm-roberta-base` даёт для тех же строк 14 и 17 токенов.

Разные языки действительно могут занимать разное количество токенов при передаче сопоставимого смысла. Во многих распространённых токенизаторах русский текст в среднем оказывается длиннее английского. Однако это статистическая тенденция, а не правило для каждой фразы.

Например, многоязычный BERT кодирует `Да.` и `Yes.` двумя токенами, а `Москва` и `Moscow` - одним. У XLM-R русский вариант `Спасибо.` занимает два токена, а английский `Thank you.` - три. Результат зависит от конкретного текста, алгоритма, словаря и данных, на которых обучали токенизатор.

Поэтому сравнивать языки корректно только на выбранном наборе примеров и для конкретного токенизатора. По одному примеру нельзя вывести постоянный коэффициент вроде «одно русское слово всегда равно трём токенам».

> *Для любознательных. В исследовании [Language Model Tokenizers Introduce Unfairness Between Languages] сравнивается средняя эффективность токенизации разных языков и обсуждается, как различия связаны со стоимостью и доступностью языковых моделей.*

5. Как правильно считать токены

Для подсчёта используется `len(encoding.ids)`, но сначала нужно решить, что именно считать. На практике встречаются несколько задач.

Задача 1. Посчитать токены исходного текста

Если нужны только токены, полученные из текста, отключите добавление служебных токенов:

```
text = "Сегодня первая лабораторная."
encoding = tokenizer.encode(text, add_special_tokens=False)

print(encoding.tokens)
print(len(encoding.ids))
```

Задача 2. Посчитать вход модели

Если нужно узнать длину последовательности после стандартной постобработки BERT, оставьте `add_special_tokens=True`. Это значение используется по умолчанию:

```
plain = tokenizer.encode(text, add_special_tokens=False)
model_input = tokenizer.encode(text, add_special_tokens=True)

print("Без специальных токенов:", len(plain.ids))
print("Со специальными токенами:", len(model_input.ids))
print("Добавлено:", len(model_input.ids) - len(plain.ids))
```

Для одной последовательности BERT обычно добавляет `[CLS]` и `[SEP]`, поэтому разница равна двум. Это не универсальное правило: постобработка другой модели может быть устроена иначе.

6. Специальные токены

Специальные токены передают структуру входа или правило обработки. Их названия, роли и наличие зависят от модели.

Токен	Типичная роль	Где часто встречается
`[CLS]`	Начало последовательности; его представление часто используют для классификации	BERT-подобные модели

Токен	Типичная роль	Где часто встречается
`[SEP]`	Разделитель частей входа и маркер конца части	BERT-подобные модели
`[PAD]`	Заполнитель до общей длины в пакете	Модели с пакетной обработкой
`[UNK]`	Неизвестный фрагмент	Некоторые подсловные токенизаторы
`[MASK]`	Позиция для задачи маскированного языкового моделирования	BERT-подобные модели
`<bos>`, `<eos>`	Начало и конец последовательности	Некоторые генеративные модели

`[CLS]` и `[SEP]` есть не у каждой модели. Наличие токена в словаре также не означает, что он добавляется автоматически: это определяют постпроцессор и аргумент `add_special_tokens`.

Идентификатор специального токена можно получить так:

```
for token in ["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"]:
    print(token, tokenizer.token_to_id(token))
```

Если элемента нет в словаре, `token_to_id` вернёт `None`.

Пара последовательностей

Некоторые модели принимают две связанные части, например вопрос и контекст:

```
pair = tokenizer.encode(
    "Курс начался",
    "Лабораторная уже доступна",
    add_special_tokens=True,
)

print(pair.tokens)
print(pair.special_tokens_mask)
```

Фактический результат:

```
['[CLS]', 'К', '##урс', 'начался', '[SEP]', 'Ла', '##бора', '##тор',
 '##ная', 'уже', 'доступ', '##на', '[SEP]']
[1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
```

Для этого BERT структура выглядит так:

```
[CLS] первая часть [SEP] вторая часть [SEP]
```

В `special_tokens_mask` единица отмечает специальный токен, а нуль - токен из исходного текста.

7. Пакетная обработка, `[PAD]` и маска внимания

В одном пакете последовательности часто должны иметь одинаковую длину. Короткие последовательности дополняют токенами `[PAD]` до длины самой длинной.

```
pad_id = tokenizer.token_to_id("[PAD]")

if pad_id is None:
    raise ValueError("У токенизатора нет [PAD]")

tokenizer.enable_padding(
    pad_id=pad_id,
    pad_token="[PAD]",
)

batch = tokenizer.encode_batch(
    ["Короткая фраза", "Фраза заметно длиннее первой"]
)

for encoding in batch:
    print(encoding.tokens)
    print(encoding.attention_mask)
    print(encoding.special_tokens_mask)
```

Первая последовательность после дополнения:

```
['[CLS]', 'К', '##орот', '##кая', 'фр', '##аза', '[SEP]', '[PAD]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]']
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
```

Здесь:

- `attention_mask` содержит `1` на содержательных и структурных позициях и `0` на позициях заполнения;
- `special_tokens_mask` отмечает единицей `[CLS]`, `[SEP]` и `[PAD]`;
- сама по себе маска ничего не «игнорирует»: её нужно передать модели, которая использует маску внимания.

После включения padding он будет применяться и в последующих вызовах. Отключить его можно методом `tokenizer.no_padding()`.

8. `[UNK]` и обратимость

`[UNK]` не обязательно означает ошибку программы. Он показывает, что токенизатор не смог представить фрагмент элементами своего словаря. Разные неизвестные фрагменты превращаются в один идентификатор, поэтому исходные различия теряются.

Байтовые токенизаторы обычно могут представить произвольные байты и потому реже нуждаются в `[UNK]`. Но это свойство нужно проверять у конкретного токенизатора, а не переносить на все модели.

Метод `decode` преобразует идентификаторы обратно в текст:

```
decoded = tokenizer.decode(encoding.ids)
decoded_with_special = tokenizer.decode(
    encoding.ids,
    skip_special_tokens=False,
)
```

По умолчанию `skip_special_tokens=True`. Даже при `False` декодирование может не восстановить исходную строку посимвольно из-за нормализации или `[UNK]`.

9. Типичные ошибки

1. Считать токены разделением по пробелам. Пунктуация и части слов тоже могут стать токенами.
2. Использовать чужой токенизатор. Число токенов действительно только для выбранного токенизатора и его конфигурации.
3. Не указывать, учитываются ли специальные токены. Для воспроизводимого ответа фиксируйте `add_special_tokens=True` или `False`.
4. Считать `[PAD]` содержанием. Padding нужен для выравнивания пакета; его позиции отмечаются нулями в `attention_mask`.
5. Ожидать полного восстановления через `decode`. Токенизация может быть необратимой.

10. Что нужно уметь перед лабораторной

Вы должны уметь:

- объяснить, почему токен не равен слову;
- загрузить `Tokenizer` через `from_pretrained` или `from_file`;
- различать `encoding.tokens` и `encoding.ids`;
- посчитать длину через `len(encoding.ids)`;
- объяснить влияние `add_special_tokens` на результат;
- прочитать `attention_mask` и `special_tokens_mask`;
- выбрать токенизатор, совместимый с исследуемой моделью.

Мини-проверка

1. Почему два токенизатора могут дать разное число токенов для одной строки?
2. В чём разница между `token_id` и эмбеддингом?
3. Как посчитать только токены исходного текста, без `[CLS]` и `[SEP]`?
4. Почему у `[PAD]` в `attention_mask` стоит `0`, а в `special_tokens_mask` - `1`?
5. Верно ли, что `[CLS]` есть у любой языковой модели? Объясните.
6. Что неверно в коде `len(text.split())`, если он используется для подсчёта токенов?
7. Почему по одному примеру нельзя вывести универсальный коэффициент пересчёта между русским и английским текстом?