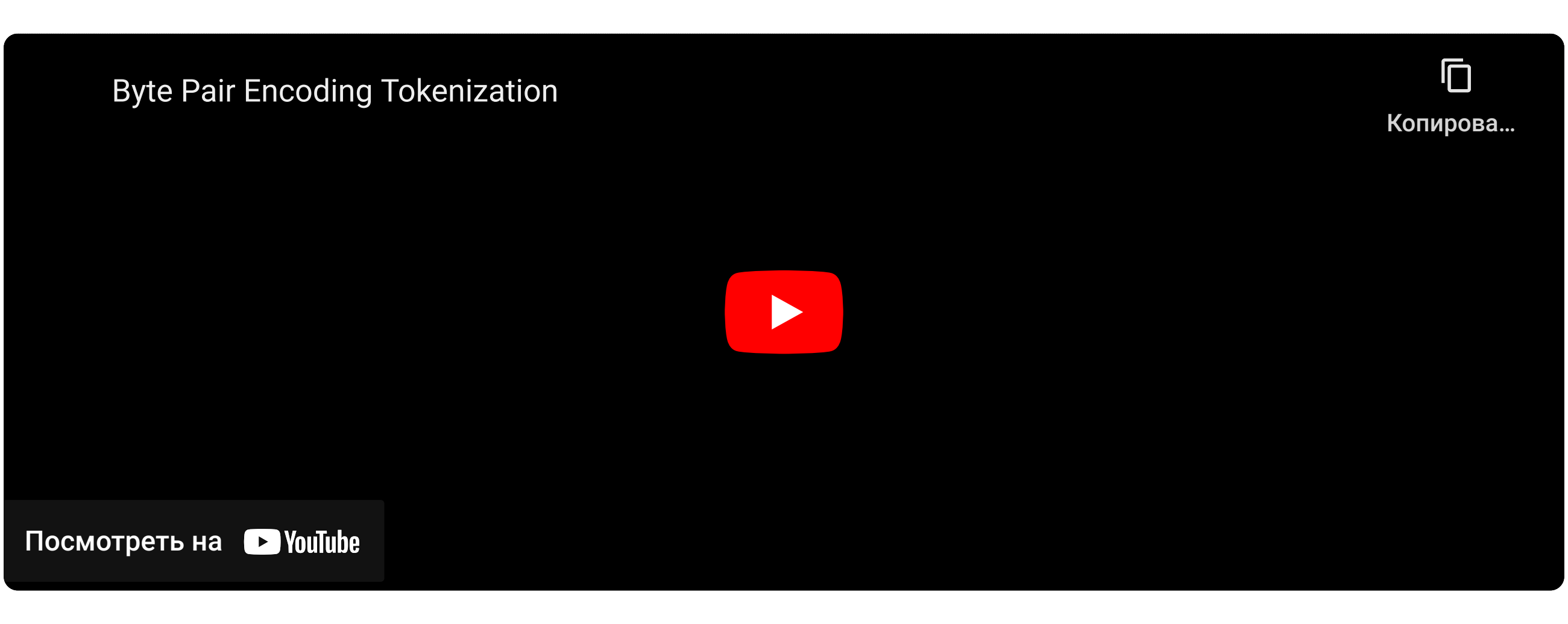
Токенизация Byte-Pair Encoding

Ask a question Open in Colab Open Studio Lab

Byte-Pair Encoding (BPE) изначально была разработана как алгоритм для сжатия текстов, а затем использовалась OpenAI для токенизации при предварительном обучении модели GPT. Она используется во многих моделях трансформеров, включая GPT, GPT-2, RoBERTa, BART и DeBERTa.



💡 В этом разделе подробно рассматривается BPE, вплоть до демонстрации полной реализации. Вы можете пропустить этот раздел, если вам нужен только общий обзор алгоритма токенизации.

Алгоритм обучения

Обучение BPE начинается с вычисления уникального набора слов, используемых в корпусе (после завершения этапов нормализации и предварительной токенизации), затем создается словарь, в который заносятся все символы, используемые для записи этих слов. В качестве очень простого примера предположим, что в нашем корпусе используются следующие пять слов:

```
"hug", "pug", "pun", "bun", "hugs"
```

Тогда базовым словарем будет `["b", "g", "h", "n", "p", "s", "u"]`. В реальном мире этот базовый словарь будет содержать, как минимум, все символы ASCII, а возможно, и некоторые символы Unicode. Если в примере, который вы обрабатываете, используется символ, которого нет в обучающем корпусе, этот символ будет преобразован в неизвестный токен. Это одна из причин, по которой многие модели NLP очень плохо анализируют контент с эмоджи, например.

> Токенизаторы GPT-2 и RoBERTa (которые довольно похожи) имеют умный способ решения этой проблемы: они рассматривают слова не как символы Unicode, а как байты. Таким образом, базовый словарь имеет небольшой размер (256), но все символы, которые вы можете придумать, все равно будут включены и не будут преобразованы в неизвестный токен. Этот трюк называется *byte-level BPE*.

После получения базового словаря мы добавляем новые токены, пока не достигнем желаемого объема словаря, обучаясь *слияниям*, которые представляют собой правила слияния двух элементов существующего словаря в новый. Таким образом, в начале эти слияния будут создавать токены с двумя символами, а затем, по мере обучения, более длинные подслова.

На любом шаге обучения токенизатора алгоритм BPE будет искать наиболее частую пару существующих токенов (под “парой” здесь понимаются два последовательных токена в слове). Эта наиболее часто встречающаяся пара и будет объединена, после чего все повторяется для следующего шага.

Возвращаясь к нашему предыдущему примеру, предположим, что слова имеют следующую частоту:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

значение `" hug"` встречалось в корпусе 10 раз, `"pug"` - 5 раз, `"pun"` - 12 раз, `"bun"` - 4 раза, и `"hugs"` - 5 раз. Мы начинаем обучение с разбиения каждого слова на части символов (те, которые формируют наш начальный словарь), чтобы мы могли рассматривать каждое слово как список токенов:

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

Затем мы посмотрим на пары. Пара `("h", "u")` присутствует в словах `"hug"` и `"hugs"`, всего 15 раз в корпусе. Однако это не самая частая пара: эта честь принадлежит `("u", "g")`, которая присутствует в словах `"hug"`, `"pug"` и `"hugs"`, в общей сложности 20 раз в словаре.

Таким образом, первое правило слияния, выученное токенизатором, - `("u", "g") -> "ug"`, что означает, что `"ug"` будет добавлено в словарь, и эта пара должна быть объединена во всех словах корпуса. В конце этого этапа словарь и корпус выглядят следующим образом:

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

Теперь у нас есть несколько пар, в результате которых получается токен длиннее двух символов: например, пара `("h", "ug")` (встречается в корпусе 15 раз). Самая частая пара на этом этапе - `("u", "n")`, однако она встречается в корпусе 16 раз, поэтому второе выученное правило слияния - `("u", "n") -> "un"`. Добавив это в словарь и объединив все существующие вхождения, мы получаем:

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)
```

Теперь наиболее частой парой является `("h", "ug")`, поэтому мы изучаем правило слияния `("h", "ug") -> "hug"`, что дает нам первый трехбуквенный токен. После слияния корпус выглядит следующим образом:

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

И продолжаем в том же духе, пока не достигнем желаемого размера словаря.

> ✏️ **Теперь ваша очередь!** Как вы думаете, каким будет следующее правило слияния?

Алгоритм токенизации

Токенизация следует за процессом обучения в том смысле, что новые входные данные подвергаются токенизации путем применения следующих шагов:

1. Нормализация
2. Предварительная токенизация
3. Разделение слов на отдельные символы
4. Применение правил слияния, изученных по порядку, к этим частям

Возьмем пример, который мы использовали во время обучения, с тремя выученными правилами слияния:

```
("u", "g") -> "ug"
("u", "n") -> "un"
("h", "ug") -> "hug"
```

Слово `"bug"` будет токенизировано как `["b", "ug"]`. Слово `"mug"`, однако, будет токенизировано как `["[UNK]", "ug"]`, поскольку буква `"m"` отсутствует в базовом словаре. Аналогично, слово `"thug" будет токенизировано как` [”[UNK]”, “hug”]`: буква` “t” отсутствует в базовом словаре, и применение правил слияния приводит сначала к слиянию `"u"` и `"g"`, а затем к слиянию `"h"` и `"ug"`.

> ✏️ **Теперь ваша очередь!** Как вы думаете, как будет токенизировано слово `'unhug'`?

Реализация BPE

Теперь давайте посмотрим на реализацию алгоритма BPE. Это не будет оптимизированная версия, которую вы сможете использовать на большом корпусе; мы просто хотим показать вам код, чтобы вы могли лучше понять алгоритм.

Для начала нам нужен корпус текста, поэтому давайте создадим простой корпус с несколькими предложениями:

```
corpus = [
    "This is the Hugging Face Course.",
    "This chapter is about tokenization.",
    "This section shows several tokenizer algorithms.",
    "Hopefully, you will be able to understand how they are trained and generate tokens.",
]
```

Далее нам нужно предварительно токенизировать корпус в слова. Поскольку мы воспроизводим токенизатор BPE (например, GPT-2), для предварительной токенизации мы будем использовать токенизатор `gpt2`:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
```

Затем мы вычисляем частоту каждого слова в корпусе, как и при предварительной токенизации:

```
from collections import defaultdict

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

print(word_freqs)
```

```
defaultdict(int, {'This': 3, 'Ġis': 2, 'Ġthe': 1, 'ĠHugging': 1, 'ĠFace': 1, 'ĠCourse': 1, '.': 4, 'Ġchap
    'Ġabout': 1, 'Ġtokenization': 1, 'Ġsection': 1, 'Ġshows': 1, 'Ġseveral': 1, 'Ġtokenizer': 1, 'Ġalgori
    'Hopefully': 1, ',': 1, 'Ġyou': 1, 'Ġwill': 1, 'Ġbe': 1, 'Ġable': 1, 'Ġto': 1, 'Ġunderstand': 1, 'Ġho
    'Ġthey': 1, 'Ġare': 1, 'Ġtrained': 1, 'Ġand': 1, 'Ġgenerate': 1, 'Ġtokens': 1})
```

Следующий шаг - составление базового словаря, состоящего из всех символов, используемых в корпусе:

```
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

print(alphabet)
```

```
[ ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p'
  't', 'u', 'v', 'w', 'y', 'z', 'Ġ']
```

Мы также добавляем специальные токены, используемые моделью, в начало этого словаря. В случае GPT-2 единственным специальным токеном является `"<|endoftext|>"`:

```
vocab = ["<|endoftext|>"] + alphabet.copy()
```

Теперь нам нужно разделить каждое слово на отдельные символы, чтобы можно было начать обучение:

```
splits = {word: [c for c in word] for word in word_freqs.keys()}
```

Теперь, когда мы готовы к обучению, давайте напишем функцию, которая вычисляет частоту каждой пары. Нам нужно будет использовать ее на каждом шаге обучения:

```
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs
```

Давайте посмотрим на часть этого словаря после первых разделений:

```
pair_freqs = compute_pair_freqs(splits)

for i, key in enumerate(pair_freqs.keys()):
    print(f"{key}: {pair_freqs[key]}")
    if i >= 5:
        break
```

```
('T', 'h'): 3
('h', 'i'): 3
('i', 's'): 5
('Ġ', 'i'): 2
('Ġ', 't'): 7
('t', 'h'): 3
```

Теперь, чтобы найти наиболее часто встречающуюся пару, нужно всего лишь сделать быстрый цикл:

```
best_pair = ""
max_freq = None

for pair, freq in pair_freqs.items():
    if max_freq is None or max_freq < freq:
        best_pair = pair
        max_freq = freq

print(best_pair, max_freq)
```

```
('Ġ', 't') 7
```

Итак, первое слияние, которое нужно выучить, это `('Ġ', 't') -> 'Ġt'`, и мы добавляем `'Ġt'` в словарь:

```
merges = {("Ġ", "t"): "Ġt"}
vocab.append("Ġt")
```

Чтобы продолжить, нам нужно применить это объединение в нашем экземпляре `splits` словаря. Давайте напишем для этого еще одну функцию:

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue

        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                split = split[:i] + [a + b] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

И мы можем посмотреть на результат первого слияния:

```
splits = merge_pair("Ġ", "t", splits)
print(splits["Ġtrained"])
```

```
['Ġt', 'r', 'a', 'i', 'n', 'e', 'd']
```

Теперь у нас есть все, что нужно, чтобы проитерироваться до тех пор, пока мы не выучим все слияния, которые нам нужны. Пусть размер словаря будет 50:

```
vocab_size = 50

while len(vocab) < vocab_size:
    pair_freqs = compute_pair_freqs(splits)
    best_pair = ""
    max_freq = None
    for pair, freq in pair_freqs.items():
        if max_freq is None or max_freq < freq:
            best_pair = pair
            max_freq = freq
    splits = merge_pair(*best_pair, splits)
    merges[best_pair] = best_pair[0] + best_pair[1]
    vocab.append(best_pair[0] + best_pair[1])
```

В результате мы выучили 19 правил слияния (исходный словарь имел размер 31 - 30 символов в алфавите плюс специальный токен):

```
print(merges)
```

```
{('Ġ', 't'): 'Ġt', ('i', 's'): 'is', ('e', 'r'): 'er', ('Ġ', 'a'): 'Ġa', ('Ġt', 'o'): 'Ġto', ('e', 'n'):
 ('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('Ġto', 'k'): 'Ġtok',
 ('Ġtok', 'en'): 'Ġtoken', ('n', 'd'): 'nd', ('Ġ', 'is'): 'Ġis', ('Ġt', 'h'): 'Ġth', ('Ġth', 'e'): 'Ġthe'
 ('i', 'n'): 'in', ('Ġa', 'b'): 'Ġab', ('Ġtoken', 'i'): 'Ġtokeni'}
```

А словарь состоит из специального токена, начального алфавита и всех результатов слияния:

```
print(vocab)
```

```
['<|endoftext|>', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm
 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'Ġ', 'Ġt', 'is', 'er', 'Ġa', 'Ġto', 'en', 'Th', 'This', 'ou
 'Ġtok', 'Ġtoken', 'nd', 'Ġis', 'Ġth', 'Ġthe', 'in', 'Ġab', 'Ġtokeni']
```

> 💡 Использование `train_new_from_iterator()` на том же корпусе не приведет к созданию точно такого же словаря. Это связано с тем, что при выборе наиболее частотной пары мы выбираем первую попавшуюся, в то время как библиотека 🤗 Tokenizers выбирает первую пару, основываясь на ее внутренних ID.

Чтобы токенизировать новый текст, мы предварительно токенизируем его, разбиваем на части, а затем применяем все изученные правила слияния:

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    splits = [[l for l in word] for word in pre_tokenized_text]
    for pair, merge in merges.items():
        for idx, split in enumerate(splits):
            i = 0
            while i < len(split) - 1:
                if split[i] == pair[0] and split[i + 1] == pair[1]:
                    split = split[:i] + [merge] + split[i + 2 :]
                else:
                    i += 1
            splits[idx] = split

    return sum(splits, [])
```

Мы можем попробовать это на любом тексте, состоящем из символов алфавита:

```
tokenize("This is not a token.")
```

```
['This', 'Ġis', 'Ġ', 'n', 'o', 't', 'Ġa', 'Ġtoken', '.']
```

> ⚠️ Наша реализация будет выбрасывать ошибку при наличии неизвестного символа, поскольку мы ничего не сделали для их обработки. На самом деле в GPT-2 нет неизвестного токена (невозможно получить неизвестный символ при использовании BPE на уровне байтов), но здесь это может произойти, поскольку мы не включили все возможные байты в начальный словарь. Этот аспект BPE выходит за рамки данного раздела, поэтому мы опустили подробности.

Вот и все об алгоритме BPE! Далее мы рассмотрим WordPiece.

<> Update on GitHub