Обучение нового токенизатора на основе старого

Ask a question | Open in Colab | Open Studio Lab

Если языковая модель не доступна на интересующем вас языке или ваш корпус сильно отличается от того, на котором обучалась языковая модель, вам, скорее всего, придется заново обучать модель с нуля, используя токенизатор, адаптированный к вашим данным. Для этого потребуется обучить новый токенизатор на вашем наборе данных. Но что именно это значит? Когда мы впервые рассматривали токенизаторы в Главе 2, мы увидели, что большинство моделей трансформеров используют *алгоритм токенизации по подсловам*. Чтобы определить, какие подслова представляют интерес и наиболее часто встречаются в корпусе, токенизатор должен внимательно изучить все тексты в корпусе - этот процесс мы называем *обучением*. Точные правила обучения зависят от типа используемого токенизатора, далее в этой главе мы рассмотрим три основных алгоритма.

⚠️ Обучение токенизатора - это не то же самое, что обучение модели! При обучении модели используется стохастический градиентный спуск, чтобы сделать потери немного меньше для каждого батча. Оно рандомизировано по своей природе (это означает, что вам нужно задать некоторое число seed, чтобы получить одинаковые результаты при повторном обучении). Обучение токенизатора - это статистический процесс, который пытается определить, какие подслова лучше всего выбрать для данного корпуса, а точные правила, используемые для их выбора, зависят от алгоритма токенизации. Это детерминированный процесс, то есть вы всегда получите одинаковые результаты при обучении одного и того же алгоритма на одном и том же корпусе.

Сбор корпуса слов

В 🤗 Transformers есть очень простой API, который можно использовать для обучения нового токенизатора с теми же характеристиками, что и у существующего: `AutoTokenizer.train_new_from_iterator()`. Чтобы увидеть это в действии, предположим, что мы хотим обучить GPT-2 с нуля, но на языке, отличном от английского. Нашей первой задачей будет собрать много данных на этом языке в обучающий корпус. Чтобы примеры были понятны всем, мы будем использовать не русский или китайский язык, а будем использовать специализированный английский: Python-код.

Библиотека 🤗 Datasets может помочь нам собрать корпус исходного кода Python. Мы воспользуемся обычной функцией `load_dataset()` для загрузки и кэширования набора данных CodeSearchNet. Этот набор данных был создан для конкурса CodeSearchNet challenge и содержит миллионы функций из библиотек с открытым исходным кодом с GitHub на нескольких языках программирования. Здесь мы загрузим Python-часть этого набора данных:

```
from datasets import load_dataset

# Загрузка может занять несколько минут, так что выпейте кофе или чай, пока ждете!
raw_datasets = load_dataset("code_search_net", "python")
```

Мы можем взглянуть на тренировочную часть датасета, чтобы узнать, к каким столбцам у нас есть доступ:

```
raw_datasets["train"]
```

```
Dataset({
    features: ['repository_name', 'func_path_in_repository', 'func_name', 'whole_func_string', 'language'
      'func_code_string', 'func_code_tokens', 'func_documentation_string', 'func_documentation_tokens', '
      'func_code_url'
    ],
    num_rows: 412178
})
```

Мы видим, что набор данных отделяет документацию от кода и предлагает токенизировать и то, и другое. Здесь мы просто используем колонку `whole_func_string` для обучения нашего токенизатора. Мы можем посмотреть пример одной из этих функций, обратившись к соответствующему индексу в части `train`:

```
print(raw_datasets["train"][123456]["whole_func_string"])
```

который должен вывести следующее:

```
def handle_simple_responses(
      self, timeout_ms=None, info_cb=DEFAULT_MESSAGE_CALLBACK):
    """Accepts normal responses from the device.

    Args:
      timeout_ms: Timeout in milliseconds to wait for each response.
      info_cb: Optional callback for text sent from the bootloader.

    Returns:
      OKAY packet's message.
    """
    return self._accept_responses('OKAY', info_cb, timeout_ms=timeout_ms)
```

Первое, что нам нужно сделать, это преобразовать набор данных в *итератор* списков текстов — например, список списков текстов. Использование списков текстов позволит нашему токенизатору работать быстрее (обучение на батчах текстов вместо обработки отдельных текстов по одному), и это должен быть итератор, если мы хотим избежать необходимости держать в памяти все сразу. Если ваш корпус огромен, вы захотите воспользоваться тем, что 🤗 Datasets не загружает все в RAM, а хранит элементы набора данных на диске.

Следующее действие создаст список списков по 1 000 текстов в каждом, но загрузит все в память:

```
# Не раскоментируйте следующую строку кода, если только ваш набор данных не маленький!
# training_corpus = [raw_datasets["train"][i: i + 1000]["whole_func_string"] for i in range(0, len(raw_da
```

Используя генератор Python, мы можем не загружать ничего в память Python до тех пор, пока это действительно не понадобится. Чтобы создать такой генератор, нужно просто заменить скобки на круглые скобки:

```
training_corpus = (
    raw_datasets["train"][i : i + 1000]["whole_func_string"]
    for i in range(0, len(raw_datasets["train"]), 1000)
)
```

Эта строка кода не получает никаких элементов из набора данных; она просто создает объект, который можно использовать в цикле Python `for`. Тексты будут загружаться только тогда, когда они вам нужны (то есть когда вы находитесь на том шаге цикла `for`, где они требуются), и за один раз будет загружено только 1 000 текстов. Таким образом, вы не исчерпаете всю память, даже если обрабатываете огромный набор данных.

Проблема с объектом-генератором заключается в том, что он может быть использован только один раз. Поэтому вместо того, чтобы выдать нам список первых 10 цифр дважды:

```
gen = (i for i in range(10))
print(list(gen))
print(list(gen))
```

мы получаем его один раз, а затем пустой список:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[]
```

Поэтому мы определяем функцию, которая возвращает генератор:

```
def get_training_corpus():
    return (
        raw_datasets["train"][i : i + 1000]["whole_func_string"]
        for i in range(0, len(raw_datasets["train"]), 1000)
    )


training_corpus = get_training_corpus()
```

Вы также можете определить свой генератор внутри цикла `for`, используя оператор `yield`:

```
def get_training_corpus():
    dataset = raw_datasets["train"]
    for start_idx in range(0, len(dataset), 1000):
        samples = dataset[start_idx : start_idx + 1000]
        yield samples["whole_func_string"]
```

который выдает точно такой же генератор, как и предыдущий, но позволяет использовать более сложную логику, чем при работе с list comprehension.

Обучение нового токенизатора

Теперь, когда у нас есть корпус в виде итератора батчей текстов, мы готовы обучить новый токенизатор. Для этого нам сначала нужно загрузить токенизатор, который мы хотим использовать в паре с нашей моделью (здесь GPT-2):

```
from transformers import AutoTokenizer

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

Несмотря на то, что мы собираемся обучить новый токенизатор, это хорошая идея сделать это, не начиная все с нуля. Таким образом, нам не придется ничего уточнять об алгоритме токенизации или специальных токенах, которые мы хотим использовать; наш новый токенизатор будет точно таким же, как GPT-2, и единственное, что изменится, - это словарный запас, который будет определен в результате обучения на нашем корпусе.

Сначала давайте посмотрим, как будет работать этот токенизатор с примером функции:

```
example = '''def add_numbers(a, b):
    """Add the two numbers `a` and `b`."""
    return a + b'''

tokens = old_tokenizer.tokenize(example)
tokens
```

```
['def', 'Ġadd', '_', 'n', 'umbers', '(', 'a', ',', 'Ġb', '):', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġ"""', 'Add', 'Ġthe',
 'Ġnumbers', 'Ġ`', 'a', '`', 'Ġand', 'Ġ`', 'b', '`', '."', '""', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġreturn', 'Ġa', 'Ġ+
```

Этот токенизатор имеет несколько специальных символов, таких как `Ġ` и `Ċ`, которые обозначают пробелы и новые строки, соответственно. Как мы видим, это не слишком эффективно: токенизатор возвращает отдельные токены для каждого пробела, в то время как он мог бы группировать уровни отступов (поскольку наборы из четырех или восьми пробелов будут очень часто встречаться в коде). Он также немного странно разделил имя функции, не ожидая увидеть слова с символом `_`.

Давайте обучим новый токенизатор и посмотрим, решит ли он эти проблемы. Для этого мы воспользуемся методом `train_new_from_iterator()`:

```
tokenizer = old_tokenizer.train_new_from_iterator(training_corpus, 52000)
```

Выполнение этой команды может занять много времени, если ваш корпус очень большой, но для данного набора данных с 1,6 ГБ текстов она работает молниеносно (1 минута 16 секунд на процессоре AMD Ryzen 9 3900X с 12 ядрами).

Обратите внимание, что `AutoTokenizer.train_new_from_iterator()` работает только в том случае, если используемый вами токенизатор является “быстрым” токенизатором. Как вы увидите в следующем разделе, библиотека 🤗 Transformers содержит два типа токенизаторов: одни написаны исключительно на Python, а другие (быстрые) опираются на библиотеку 🤗 Tokenizers, которая написана на языке программирования Rust. Python - это язык, который чаще всего используется для приложений data science и deep learning, но когда что-то нужно распараллелить для быстроты, это приходится писать на другом языке. Например, матричные умножения, которые лежат в основе вычислений модели, написаны на CUDA, оптимизированной библиотеке языка C для GPU.

Обучение совершенно нового токенизатора на чистом Python было бы мучительно медленным, поэтому мы разработали библиотеку 🤗 Tokenizers. Обратите внимание, что так же как вам не нужно было изучать язык CUDA, чтобы выполнить свою модель на батче входных данных на GPU, вам не понадобится изучать Rust, чтобы использовать быстрый токенизатор. Библиотека 🤗 Tokenizers предоставляет привязки к Python для многих методов, которые внутренне вызывают некоторые части кода на Rust; например, для распараллеливания обучения вашего нового токенизатора или, как мы видели в Главе 3, токенизации батча входных данных.

В большинстве моделей Transformer доступен быстрый токенизатор (есть некоторые исключения, о которых вы можете узнать здесь), а API `AutoTokenizer` всегда выбирает быстрый токенизатор, если он доступен. В следующем разделе мы рассмотрим некоторые другие особенности быстрых токенизаторов, которые будут очень полезны для таких задач, как классификация токенов и ответы на вопросы. Однако прежде чем погрузиться в эту тему, давайте попробуем наш новый токенизатор на предыдущем примере:

```
tokens = tokenizer.tokenize(example)
tokens
```

```
['def', 'Ġadd', '_', 'numbers', '(', 'a', ',', 'Ġb', '):', 'ĊĠĠĠ', 'Ġ"""', 'Add', 'Ġthe', 'Ġtwo', 'Ġnumbe
 'a', '`', 'Ġand', 'Ġ`', 'b', '`."""', 'ĊĠĠĠ', 'Ġreturn', 'Ġa', 'Ġ+', 'Ġb']
```

Здесь мы снова видим специальные символы `Ġ` и `Ċ`, обозначающие пробелы и новые строки, но мы также видим, что наш токенизатор выучил некоторые токены, которые очень специфичны для корпуса функций Python: например, есть токен `ĊĠĠĠ`, который обозначает отступ, и токен `Ġ"""`, который обозначает три кавычки, с которых начинается doc-строка. Токенизатор также правильно разделил имя функции на `_`. Это довольно компактное представление; для сравнения, использование токенизатора простого английского языка для того же примера даст нам более длинное предложение:

```
print(len(tokens))
print(len(old_tokenizer.tokenize(example)))
```

```
27
36
```

Давайте рассмотрим другой пример:

```
example = """class LinearLayer():
    def __init__(self, input_size, output_size):
        self.weight = torch.randn(input_size, output_size)
        self.bias = torch.zeros(output_size)

    def __call__(self, x):
        return x @ self.weights + self.bias
    """
tokenizer.tokenize(example)
```

```
['class', 'ĠLinear', 'Layer', '():', 'ĊĠĠĠ', 'Ġdef', 'Ġ__', 'init', '__(', 'self', ',', 'Ġinput', '_', 's
 'Ġoutput', '_', 'size', '):', 'ĊĠĠĠĠĠĠĠ', 'Ġself', '.', 'weight', 'Ġ=', 'Ġtorch', '.', 'randn', '(', 'ir
 'size', ',', 'Ġoutput', '_', 'size', ')', 'ĊĠĠĠĠĠĠĠ', 'Ġself', '.', 'bias', 'Ġ=', 'Ġtorch', '.', 'zeros'
 'output', '_', 'size', ')', 'ĊĊĠĠĠ', 'Ġdef', 'Ġ__', 'call', '__(', 'self', ',', 'Ġx', '):', 'ĊĠĠĠĠĠĠĠ',
 'Ġreturn', 'Ġx', 'Ġ@', 'Ġself', '.', 'weights', 'Ġ+', 'Ġself', '.', 'bias', 'ĊĠĠĠ']
```

В дополнение к токену, соответствующему отступу, здесь мы также видим токен для двойного отступа: `ĊĠĠĠĠĠĠĠ`. Специальные слова Python, такие как `class`, `init`, `call`, `self` и `return`, обрабатываются как один токен, и мы видим, что наряду с разделением на `_` и `.` токенизатор правильно разделяет даже имена с camel-case: `LinearLayer` обрабатывается как `["ĠLinear", "Layer"]`.

Сохранение токенизатора

Чтобы убедиться, что мы сможем использовать его позже, нам нужно сохранить наш новый токенизатор. Как и для моделей, это делается с помощью метода `save_pretrained()`:

```
tokenizer.save_pretrained("code-search-net-tokenizer")
```

В результате будет создана новая папка с именем *code-search-net-tokenizer*, в которой будут содержаться все файлы, необходимые токенизатору для загрузки. Если вы хотите поделиться этим токенизатором со своими коллегами и друзьями, вы можете загрузить его на Hub, войдя в свою учетную запись. Если вы работаете в блокноте, есть удобная функция, которая поможет вам в этом:

```
from huggingface_hub import notebook_login

notebook_login()
```

Появится виджет, в котором вы можете ввести свои учетные данные для входа в Hugging Face. Если вы работаете не в блокноте, просто введите следующую строку в терминале:

```
huggingface-cli login
```

После того как вы авторизовались, вы можете опубликовать свой токенизатор, выполнив следующую команду:

```
tokenizer.push_to_hub("code-search-net-tokenizer")
```

Это создаст новое хранилище в вашем пространстве имен с именем `code-search-net-tokenizer`, содержащее файл токенизатора. Затем вы можете загрузить токенизатор где угодно с помощью метода `from_pretrained()`:

```
# Замените "huggingface-course" ниже своим реальным пространством имен, чтобы использовать свой собствен
tokenizer = AutoTokenizer.from_pretrained("huggingface-course/code-search-net-tokenizer")
```

Теперь вы готовы обучить языковую модель с нуля и дообучить ее в соответствии с поставленной задачей! Мы займемся этим в Главе 7, но сначала в этой главе мы рассмотрим быстрые токенизаторы и подробно изучим, что происходит при вызове метода `train_new_from_iterator()`.

<> Update on GitHub