Предобработка данных

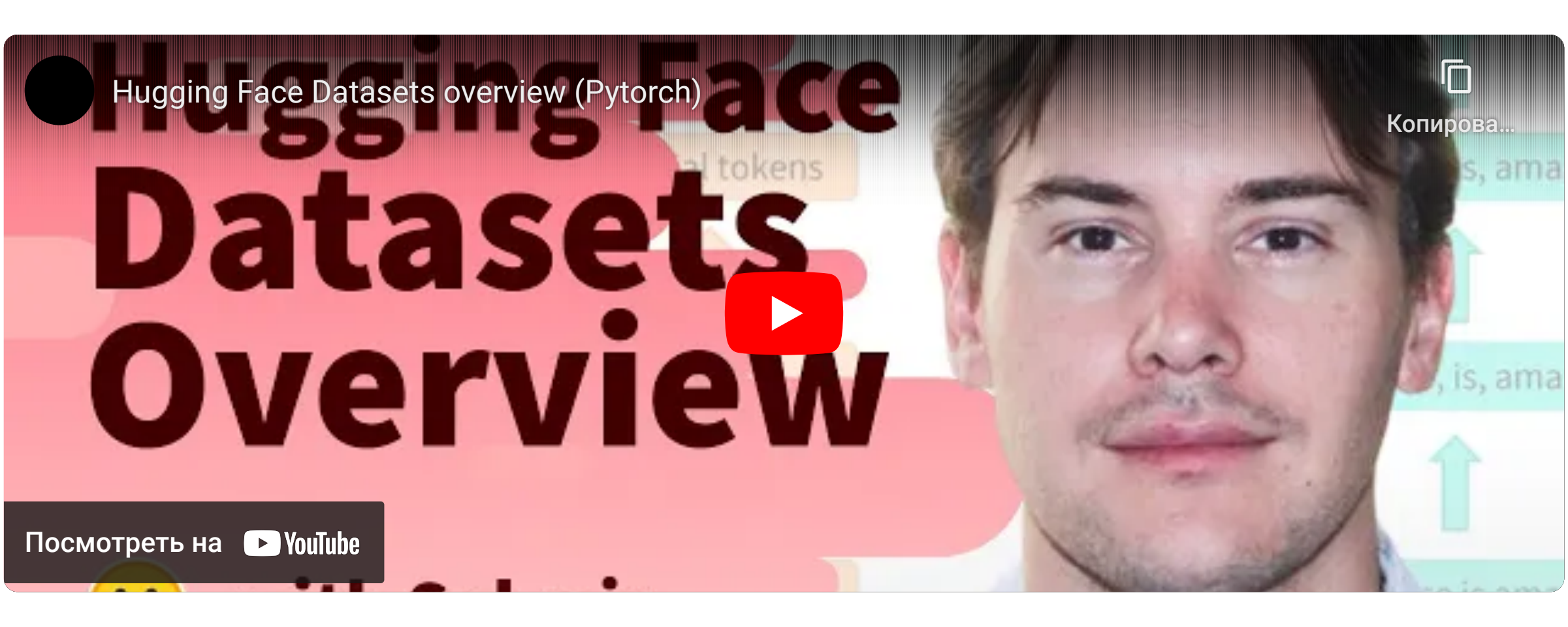
Продолжим с примером из предыдущей главы, вот как мы будем обучать классификатор последовательности на одном батче с помощью PyTorch:

```
import torch
from transformers import AdamW, AutoTokenizer, AutoModelForSequenceClassification

# Так же, как и в прошлый раз
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = [
    "I've been waiting for a HuggingFace course my whole life.",
    "This course is amazing!",
]
batch = tokenizer(sequences, padding=True, truncation=True, return_tensors="pt")

# Эта часть новая
batch["labels"] = torch.tensor([1, 1])

optimizer = AdamW(model.parameters())
loss = model(**batch).loss
loss.backward()
optimizer.step()
```

Обучение всего лишь на двух предложениях, конечно, не даст хорошего результата. Чтобы получить более качественные результаты, вам следует подготовить бо́льший датасет.

В данном разделе мы будем использовать в качестве примера MRPC (Microsoft Research Paraphrase Corpus) dataset, предложенный в статье авторами William B. Dolan и Chris Brockett. Датасет состоит из 5801 пар предложений с соответствующим им лейблом: является ли пара преложений парафразами или нет (т.е. идет ли речь в обоих предложениях об одном и том же). Мы выбрали именно этот датасет для этой главы потому что он небольшой: с ним легко экспериментировать в процессе обучения.

Загрузка датасета с Hub



Hub содержит не только модели, там также расположено множество датасетов на различных языках. Вы можете посмотреть на них тут, а также мы рекомендуем попровать загрузить новый датасет после того, как вы изучите текущий раздел (см. документацию здесь). Но сейчас вернемся к датасету MRPC! Это один из 10 датасетов из состава GLUE, который является тестом для производительности моделей машинного обучения в задачах классификации текста.

Библиотека 🤗 Datasets предоставляет возможность использовать очень простую команду для загрузки и кэширования датасета с Hub. Мы можем загрузить датасет следующим образом:

> ⚠️ **Предупреждение** Убедитесь, что `datasets` установлены, выполнив `pip install datasets`. Затем загрузите набор данных MRPC и выведите его, чтобы увидеть, что он содержит.

```
from datasets import load_dataset

raw_datasets = load_dataset("glue", "mrpc")
raw_datasets
```

```
DatasetDict({
    train: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
        num_rows: 3668
    })
    validation: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
        num_rows: 408
    })
    test: Dataset({
        features: ['sentence1', 'sentence2', 'label', 'idx'],
        num_rows: 1725
    })
})
```

Как можно заметить, мы получили объект типа `DatasetDict`, который содержит обучающую выборку, валидационную выборку и тестовую выборку. Каждая из них содержит несколько колонок (`sentence1`, `sentence2`, `label`, и `idx`) и переменную с числом строк (число элементов в каждой выборке): 3668 пар предложений в обучающей части, 408 в валидационной и 1725 в тестовой .

Эта команда загружает и кэширует датасет (по умолчанию в *~/.cache/huggingface/dataset*). Вспомним из главы 2, что вы можете изменить путь к кэшу изменив переменную окружения `HF_HOME`.

Мы можем получить доступ к предложениями в объекте `raw_datasets` путем индексирования, как в словаре:

```
raw_train_dataset = raw_datasets["train"]
raw_train_dataset[0]
```

```
{'idx': 0,
 'label': 1,
 'sentence1': 'Amrozi accused his brother , whom he called " the witness " , of deliberately distorting h
 'sentence2': 'Referring to him as only " the witness " , Amrozi accused his brother of deliberately dist
```

Можно увидеть, что лейблы уже являются целыми числами (integer), их обрабатывать не нужно. Чтобы сопоставить индекс класса с его названием, можно распечатать значение переменной `features` у `raw_train_dataset`:

```
raw_train_dataset.features
```

```
{'sentence1': Value(dtype='string', id=None),
 'sentence2': Value(dtype='string', id=None),
 'label': ClassLabel(num_classes=2, names=['not_equivalent', 'equivalent'], names_file=None, id=None),
 'idx': Value(dtype='int32', id=None)}
```

Переменная `label` типа `ClassLabel` соответствует именам в *names*. `0` соответствует `not_equivalent`, `1` соответствует `equivalent`.

> ✏️ **Попробуйте!** Посмотрите на 15-й элемент обучающей выборки и на 87-й элемент вадидационной выборки. Какие у них лейблы?

Предобработка датасета

Чтобы предобработать датасет, нам необходимо конвертировать текст в числа, которые может обработать модель. Как вы видели в предыдущей главе, это делается с помощью токенайзера. Мы можем подать на вход токенайзеру одно или список предложений, т.е. можно токенизировать предложения попарно таким образом:

```
from transformers import AutoTokenizer

checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokenized_sentences_1 = tokenizer(raw_datasets["train"]["sentence1"])
tokenized_sentences_2 = tokenizer(raw_datasets["train"]["sentence2"])
```

Однако мы не можем просто передать две последовательности в модель и получить прогноз того, являются ли эти два предложения парафразами или нет. Нам нужно обрабатывать две последовательности как пару и применять соответствующую предварительную обработку. К счастью, токенизатор также может взять пару последовательностей и подготовить их так, как ожидает наша модель BERT:

```
inputs = tokenizer("This is the first sentence.", "This is the second one.")
inputs
```

```
{
  'input_ids': [101, 2023, 2003, 1996, 2034, 6251, 1012, 102, 2023, 2003, 1996, 2117, 2028, 1012, 102],
  'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

Мы уже обсуждали ключи `input_ids` и `attention_mask` в главе 2, но не упоминали о `token_type_ids`. В этом примере мы указываем модели какая часть входных данных является первым предложением, а какая вторым.

> ✏️ **Попробуйте!** Токенизируйте 15-й элемент обучающей выборки как два предложения, и как пару предложений. В чем разница между двумя результатами?

Если мы декодируем ID из `input_ids` обратно в слова:

```
tokenizer.convert_ids_to_tokens(inputs["input_ids"])
```

мы получим

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is', 'the', 'second', 'one', '
```

Видно, что модель ожидает входные данные в следующем формате: `[CLS] sentence1 [SEP] sentence2 [SEP]` в случае двух предложений. Посмотрим соответствие элементов и `token_type_ids`

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is', 'the', 'second', 'one', '
[      0,      0,    0,     0,       0,          0,   0,       0,      1,    1,     1,        1,     1,
```

Как вы можете заметить, части входных данных, соответствующих `[CLS] sentence1 [SEP]` имеют тип токена `0`, в то время как остальные части, соответствующие второму предложению `sentence2 [SEP]`, имеют тип токена `1`.

Обратите внимание, что если вы выберете другой чекпоинт, `token_type_ids` необязательно будут присутствовать в ваших токенизированных входных данных (например, они не возвращаются, если вы используете модель DistilBERT). Они возвращаются только тогда, когда модель будет знать, что с ними делать, потому что она видела их во время предобучения.

В данном случае BERT был обучен с информацией о идентификаторах типов токенов, и помимо задачи маскированной языковой модели, о которой мы говорили в главе 1, он может решать еще одну задачу: предсказание следующего предложения (*next sentence prediction*). Суть этой задачи - смоделировать связь между предложениями.

В этой задаче модели на вход подаются пары предложений (со случайно замаскированными токенами), от модели требуется предсказать, является ли следующее предложение продолжением текущего. Чтобы задача не была слишком тривиальной, половина времени модель обучается на соседних предложениях из одного документа, другую половину на парах предложений, взятых из разных источников.

В общем случае вам не нужно беспокоиться о наличии `token_type_ids` в ваших токенизированных данных: пока вы используете одинаковый чекпоинт и для токенизатора, и для модели – токенизатор будет знать, как нужно обработать данные.

Теперь мы знаем, что токенизатор может подготовить сразу пару предложений, а значит мы можем использовать его для целого датасета: так же как и в предыдущей главе можно подать на вход токенизатору список первых предложений и список вторых предложений. Это также сработает и для механизмов дополнения (padding) и усечения до максимальной длины (truncation) - об этом мы говорили в главе 2. Итак, один из способов предобработать обучающий датасет такой:

```
tokenized_dataset = tokenizer(
    raw_datasets["train"]["sentence1"],
    raw_datasets["train"]["sentence2"],
    padding=True,
    truncation=True,
)
```

Это хорошо работает, однако есть недостаток, который формирует токенизатор (с ключами, `input_ids`, `attention_mask`, и `token_type_ids`, и значениями в формате списка списков). Это будет работать только если у нас достаточно оперативной памяти (RAM) для хранения целого датасета во время токенизации (в то время как датасеты из библиотеки 🤗 Datasets являются Apache Arrow файлами, хранящимися на диске; они будут загружены только в тот момент, когда вы их будете запрашивать).

Чтобы хранить данные в формате датасета, мы будем использовать методы `Dataset.map()`. Это позволит нам сохранить высокую гибкость даже если нам нужно что-то большее, чем просто токенизация. Метод `map()` работает так: применяет некоторую функцию к каждому элементу датасета, давайте определим функцию, которая токенизирует наши входные данные:

```
def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)
```

Эта функция принимает на вход слвоарь (похожий на элементы нашего словаря) и возвращает новый словарь с ключами `input_ids`, `attention_mask` и `token_type_ids`. Заметьте, это также работает если словарь `example` содержит несколько элементов (каждый ключ в виде списка предложений), поскольку `tokenizer` работает и со списками пар предложений, как мы и видели ранее. Это позволит нам использовать аргумент `batched=True` в вызове `map()`, которая ускорит процесс токенизации. `tokenizer` внутри реализован на языке Rust из библиотеки 🤗 Tokenizers. Этот токенизатор может быть очень быстрым, но только если мы подадим большой объем данных за раз.

Обратите внимание, в этот раз мы оставили аргумент `padding` пустым, потому что дополнение данных до максимальной длины неэффективно: гораздо быстрее делать это во время формирования батча, в таком случае мы будем дополнять до максимальной длины только элементы батча, а не целого датасета. Это поможет сэкономить время в случае длинных последовательностей.

Ниже пример того, как мы применяем функцию токенизации к целому датасету. Мы указываем `batched=True` в нашем вызове `map` и функция будет применена сразу к нескольким элементам датасета одновременно, а не к каждому по отдельности. Это позволяет сделать токенизацию более быстрой.

```
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
tokenized_datasets
```

Библиотека 🤗 Datasets применяет обработку, добавляя новые поля в наборы данных, по одному для каждого ключа в словаре, который возвращает функция предварительной обработки:

```
DatasetDict({
    train: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_i
        num_rows: 3668
    })
    validation: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_i
        num_rows: 408
    })
    test: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_i
        num_rows: 1725
    })
})
```

В функции `map()` можно использовать мультипроцессинг: за это отвечает аргумент `num_proc`. Мы его не применяли, потому что библиоетека 🤗 Tokenizers сразу использует несколько потоков для токенизации, но если вы будете использовать функцию не из 🤗 Tokenizers, это может ускорить процесс.

Наша функция `tokenize_function` возвращает словарь с ключами `input_ids`, `attention_mask` и `token_type_ids`, они уже добавлены ко всем разбиениями нашего датасета. Обратите внимание, что мы могли бы также изменить существующие поля, если бы наша функция препроцессинга вернула новое значение для существующего ключа в наборе данных, к которому мы применили `map()`.

Последнее, что нам нужно сделать, это дополнить все примеры до длины самого длинного элемента, когда мы собираем элементы вместе — метод, который мы называем *динамическим пэддингом* (*dynamic padding*).

Dynamic padding

Функция, отвечающая за объединение элементов внутри батча, называется *collate function* (функция сопоставления). Это аргумент, который вы можете передать при создании `DataLoader`, по умолчанию это функция, которая просто преобразует ваши образцы в тензоры PyTorch и объединяет их (рекурсивно, если вашими элементами являются списки, кортежи или словари). В нашем случае это невозможно, поскольку входные данные, которые у нас есть, не будут иметь одинакового размера. Мы намеренно не стали делать пэддинг, чтобы применять его только по мере необходимости в каждом пакете и избегать слишком длинных входных данных с большим количеством отступов. Это немного ускорит обучение, но учтите, что если вы тренируетесь на TPU, это может вызвать проблемы — TPU предпочитают фиксированные формы, даже если для этого требуется дополнительн пэддинг.

Для того, чтобы сделать это на практике, мы должны задать функцию сопоставления, которая будет осуществлять корректный пэддинг элементов выборки, которые мы хотим объединить в батч. К счастью, библиотека 🤗 Transformers предоставляет нам эту функцию через класс `DataCollatorWithPadding`. При создании экземпляра требуется указать токенизатор (чтобы знать, какой токен использовать для пэддинга и слева или справа нужно дополнять данные), а дальше функция сделает все, что вам нужно:

```
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Чтобы протестировать это, давайте возьмем несколько элементов обучающей выборки, которые мы хотим объединить в батч. Мы удалим колонки `idx`, `sentence1` и `sentence2` т.к. они содержат строки (а мы не можем превратить строки в тензоры) и посмотрим на длину каждой записи в батче:

```
samples = tokenized_datasets["train"][:8]
samples = {k: v for k, v in samples.items() if k not in ["idx", "sentence1", "sentence2"]}
[len(x) for x in samples["input_ids"]]
```

```
[50, 59, 47, 67, 59, 50, 62, 32]
```

Неудивительно: мы получили объекты разной длины от 32 до 67. Динамический пэддинг подразумевает, что все объекты будут дополнены до максимальной длины, до 67.

No surprise, we get samples of varying length, from 32 to 67. Dynamic padding means the samples in this batch should all be padded to a length of 67, the maximum length inside the batch. Без динамического заполнения все выборки должны быть дополнены до максимальной длины во всем наборе данных, или до максимальной длины, которую может принять модель. Давайте дважды проверим, что наш `data_collator` динамически правильно дополняет батч:

```
batch = data_collator(samples)
{k: v.shape for k, v in batch.items()}
```

```
{'attention_mask': torch.Size([8, 67]),
 'input_ids': torch.Size([8, 67]),
 'token_type_ids': torch.Size([8, 67]),
 'labels': torch.Size([8])}
```

Выглядит неплохо! Теперь мы пришли от обычного текста к батчу, с которым может работать наша модель. Можем приступить к fine-tuning!

> ✏️ **Попробуйте!** Повторите этап препроцессинга для набора данных GLUE SST-2. Он немного отличается, так как состоит из отдельных предложений, а не пар, но в остальном это то же самое, что мы сделали. Для более сложной задачи попробуйте написать функцию предварительной обработки, которая работает с любой из задач GLUE.