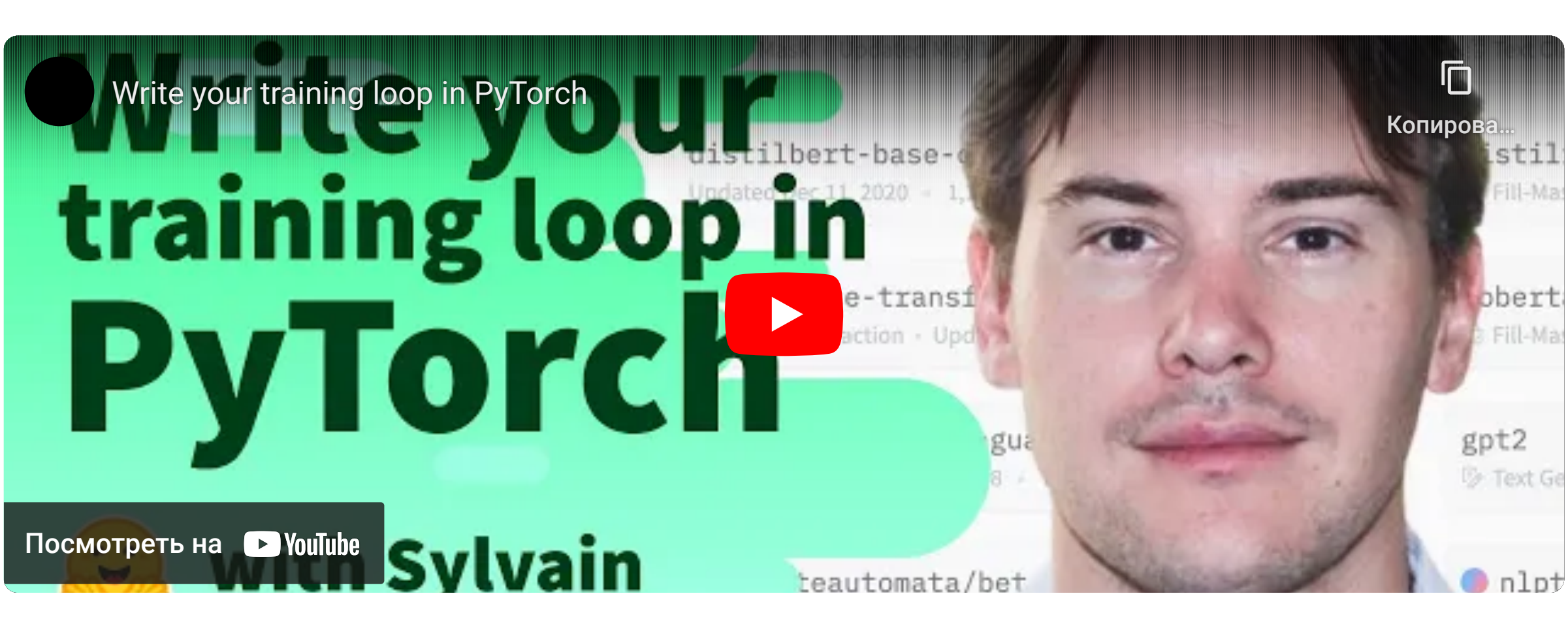
Полное обучение

Open in Colab | Open Studio Lab



Теперь мы посмотрим, как достичь результатов из предыдущей главы без использования класса `Trainer`. В этой главе мы предполагаем, что вы выполнили этапы препроцессинга раздела 2. Ниже короткая выжимка того, что вам понадобится:

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)


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


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

Подготовка к обучению

Перед реализацией цикла обучения необходимо задать несколько объектов. Первый: загрузчики данных (далее - dataloaders), которые мы будем использовать для итерирования по батчам данных. Перед этим нам необходимо применить несколько операций постпроцессинга к нашему `tokenized_datasets`. Это нужно сделать: в прошлый раз за нас это автоматически делал `Trainer`. Необходимо сделать следующее:

- Удалить колонки, соответствующие значениям, которые модель не принимает на вход (например, `sentence1` и `sentence2`).
- Переименовать колонку `label` в `labels` (потому что модель ожидает аргумент, названный `labels`).
- Задать тип данных в датасете pytorch tensors вместо списков.

Наш `tokenized_datasets` предоставляет возможность использовать встроенные методы для каждого из приведенных выше шагов:

```
tokenized_datasets = tokenized_datasets.remove_columns(["sentence1", "sentence2", "idx"])
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
tokenized_datasets.set_format("torch")
tokenized_datasets["train"].column_names
```

Мы можем проверить, что в результате у нас присутствуют только те поля, которые ожидает наша модель:

```
["attention_mask", "input_ids", "labels", "token_type_ids"]
```

Теперь, когда датасет готов, мы может задать dataloader:

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, batch_size=8, collate_fn=data_collator
)
eval_dataloader = DataLoader(
    tokenized_datasets["validation"], batch_size=8, collate_fn=data_collator
)
```

Для того, чтобы убедиться в отсутствии ошибок в сделанном нами препроцессинге, мы можем проверить один батч данных:

```
for batch in train_dataloader:
    break
{k: v.shape for k, v in batch.items()}
```

```
{'attention_mask': torch.Size([8, 65]),
 'input_ids': torch.Size([8, 65]),
 'labels': torch.Size([8]),
 'token_type_ids': torch.Size([8, 65])}
```

Обратите внимание, что фактические размеры, вероятно, будут немного отличаться для в вашем случае, так как мы установили `shuffle=True` для обучающего загрузчика данных, также мы дополняем (padding) до максимальной длины внутри батча.

Теперь мы полностью завершили этап препроцессинга (приятный, но неуловимый момент для любого специалиста по машинному обучению), перейдем к модели. Мы инициализируем ее точно так, как делали в предыдущем примере:

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

Чтобы убедиться, что обучение пойдет гладко, вы подадим на вход модели один батч:

```
outputs = model(**batch)
print(outputs.loss, outputs.logits.shape)
```

```
tensor(0.5441, grad_fn=<NllLossBackward>) torch.Size([8, 2])
```

Все модели 🤗 трансформеров возвращают значение функции потерь, если в данных были `labels`, а также логиты (в результате получается тензор 8 x 2).

Мы почти готовы к написанию обучающего цикла! Мы пропустили только две вещи: оптимизатор и планировщик скорости обучения (learning rate scheduler). Ввиду того, что мы пытаемся повторить вручную то, что делал за нас `Trainer`, мы будем использовать такие же значения по умолчанию. Оптимизатор, используемый в `Trainer` - `AdamW`, который является почти полной копией Adam, за исключением трюка с сокращением весов (далее - weight decay) (см. "Decoupled Weight Decay Regularization" за авторством Ilya Loshchilov и Frank Hutter).

```
from transformers import AdamW

optimizer = AdamW(model.parameters(), lr=5e-5)
```

Наконец, планировщик скорости обучения по умолчанию - просто линейное уменьшение весов с максимального значения (5e-5) до 0. Чтобы корректно задать его, нам нужно знать число шагов в обучении, которое задается как произведение числа эпох и числа батчей (длины нашего загрузчика данных). Число эпох по умолчанию в `Trainer` равно 3, так же мы зададим его и сейчас:

```
from transformers import get_scheduler

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)
print(num_training_steps)
```

```
1377
```

Обучающий цикл

Последний момент: мы хотим использовать GPU в случае, если у нас будет такая возможность (на CPU процесс может занять несколько часов вместо пары минут). Чтобы добиться этого, мы определим переменную `device` и «прикрепим» к видеокарте нашу модель и данные:

```
import torch

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
device
```

```
device(type='cuda')
```

Теперь мы готовы к обучению модели! Чтобы иметь представление о том, сколько времени это может занять, мы добавим прогресс-бар, который будет иллюстрировать, сколько шагов обучения уже выполнено. Это можно сделать с использованием бибилиотеки tqdm:

```
from tqdm.auto import tqdm

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

Вы можете заметить, что процесс обучения выглядит очень похожим на то, как он выглядел в наших первых примерах. Мы не указывали модели, чтобы она нам что-то возвращала в процессе обучения. Для этого мы добавим цикл валидации.

Валидационный цикл

Ранее мы использовали метрику, которую нам предоставляла библиотека 🤗 Evaluate. Мы уже знаем, что есть метод `metric.compute()`, однако метрики могут накапливать значения в процессе итерирования по батчу, для этого есть метод `add_batch()`. После того, как мы пройдемся по всем батчам, мы сможем вычислить финальный результат с помощью `metric.compute()`. Вот пример того, как это можно сделать в цикле валидации:

```
import evaluate

metric = evaluate.load("glue", "mrpc")
model.eval()
for batch in eval_dataloader:
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model(**batch)

    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)
    metric.add_batch(predictions=predictions, references=batch["labels"])

metric.compute()
```

```
{'accuracy': 0.8431372549019608, 'f1': 0.8907849829351535}
```

Повторим: результаты, которые получите вы, могут немного отличаться из-за наличия случайностей при инициализации параметров слоя модели и из-за случайного перемешивания датасета, однако их порядок должен совпадать.

> ✏️ **Попробуйте!** Измените обучающий цикл так, чтобы дообучить модель на датасете SST-2.

Ускорение обучающего цикла с помощью 🤗 Accelerate

Обучающий цикл, заданный выше, отлично работает на одном GPU или CPU. Однако использование библиотеки 🤗 Accelerate позволяет с небольшими изменениями сделать эту процедуру распределенной на несколько GPU или TPU. Начиная с момента создания обучающих и валидационных загрузчиков данных, наш «ручной» обучающий цикл выглядит так:

```
from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
optimizer = AdamW(model.parameters(), lr=3e-5)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

А вот изменения, которые нужно внести, чтобы ускорить процесс:

```
+ from accelerate import Accelerator
  from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

+ accelerator = Accelerator()

  model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
  optimizer = AdamW(model.parameters(), lr=3e-5)

- device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
- model.to(device)

+ train_dataloader, eval_dataloader, model, optimizer = accelerator.prepare(
+     train_dataloader, eval_dataloader, model, optimizer
+ )

  num_epochs = 3
  num_training_steps = num_epochs * len(train_dataloader)
  lr_scheduler = get_scheduler(
      "linear",
      optimizer=optimizer,
      num_warmup_steps=0,
      num_training_steps=num_training_steps
  )

  progress_bar = tqdm(range(num_training_steps))

  model.train()
  for epoch in range(num_epochs):
      for batch in train_dataloader:
-         batch = {k: v.to(device) for k, v in batch.items()}
          outputs = model(**batch)
          loss = outputs.loss
-         loss.backward()
+         accelerator.backward(loss)

          optimizer.step()
          lr_scheduler.step()
          optimizer.zero_grad()
          progress_bar.update(1)
```

Первая строка – это строка импорта библиотеки. Вторая строка инициализирует объект `Accelerator`, который проанализирует окружение и определит необходимые настройки. 🤗 Accelerate автоматически использует доступное оборудование, поэтому вы можете удалить строки, которые «прикрепляют» модель и данные к видеокарте (или, если вам так удобнее, можете изменить их на `accelerator.device` вместо просто `device`).

Далее главная часть работы выполняется в строке, которая отправляет данные, модель и оптимизатор на `accelerator.prepare()`. Этот метод «обернет» ваши объекты в контейнер и убедится, что распределенное обучение выполняется корректно. Оставшиеся изменения – удаление строки, которая отправляет батч на `device` (повторим: если вы хотите оставить эту строку, замените `device` на `accelerator.device`) и замените `loss.backward()` на `accelerator.backward(loss)`.

> ⚠️ Чтобы воспользоваться ускорением, предлагаемым облачными TPU, мы рекомендуем дополнять данные до фиксированной длины с помощью аргументов `padding="max_length"` и `max_length` токенизатора.

Если вы хотите скопировать и запустить этот код, это полная версия с использованием 🤗 Accelerate:

```
from accelerate import Accelerator
from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

accelerator = Accelerator()

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
optimizer = AdamW(model.parameters(), lr=3e-5)

train_dl, eval_dl, model, optimizer = accelerator.prepare(
    train_dataloader, eval_dataloader, model, optimizer
)

num_epochs = 3
num_training_steps = num_epochs * len(train_dl)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dl:
        outputs = model(**batch)
        loss = outputs.loss
        accelerator.backward(loss)

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

Добавление этого в скрипт `train.py` сделает процесс обучения универсальным для любой распределенной системы. Попробуйте запустить его на вашей распределенной системе:

```
accelerate config
```

эта строка предложит вам ответить на несколько вопросов и сохранит ваши ответы в конфигурационный файл, который будет использоваться при вызове команды:

```
accelerate launch train.py
```

запускающей распределенное обучение.

Если вы хотите попробовать запустить этот код в Jupyter Notebook (например, протестировать его с TPU на Google Colab), просто вставьте код в `training_function()` и запустите последнюю ячейку:

```
from accelerate import notebook_launcher

notebook_launcher(training_function)
```

Вы можете найти больше примеров в репозитории 🤗 Accelerate repo.

<> Update on GitHub