Fine-tuning модели с использованием Trainer API

Библиотека 🤗 Transformers предоставляет класс `Trainer`, который помогает произвести fine-tuning любой предобученной модели на вашем датасете. После предобработки данных, сделанных в прошлом разделе, вам останется сделать несколько шагов для определения `Trainer`. Самая сложная часть – подготовить окружение для запуска `Trainer.train()`, т.к. она медленно работает на центральном процессоре. Если у вас нет видеокарты, вы можете бесплатно воспользоваться сервисом Google Colab, который предоставляет доступ к вычислениям с использованием GPU и TPU.

Фрагменты кода ниже предполагают, что вы выполнили код из предыдущего раздела. Ниже приведено краткое резюме того, что вам нужно:

```
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)
```

Обучение

Первый шаг перед определением `Trainer`, задание класса `TrainingArguments`, который будет содержать все гиперпараметры для `Trainer` (для процессов обучения и валидации). Единственный аргумент, который вы должны задать — это каталог, в котором будет сохранена обученная модель, а также контрольные точки. Для всего остального вы можете оставить значения по умолчанию, которые должны подойти для базового fine-tuning.

```
from transformers import TrainingArguments

training_args = TrainingArguments("test-trainer")
```

> 💡 Если вы хотите автоматически загружать модель на Hub во время обучения, передайте аргумент `push_to_hub=True` в `TrainingArguments`. Мы больше узнаем об этом в главе 4.

Второй шаг – задание модели. Так же, как и в предыдущей главе, мы будем использовать класс `AutoModelForSequenceClassification` с двумя лейблами:

```
from transformers import AutoModelForSequenceClassification

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

После создания экземпляра предобученной модели будет распечатано предупреждение (в главе 2 мы с таким не сталкивались). Это происходит потому, что BERT не был предобучен для задачи классификации пар предложений, его последний слой не будет использован, вместо него будет добавлен слой, позволяющий работать с такой задачей. Предупреждения сообщают, что некоторые веса не будут использованы (как раз тех слоев, которые не будут использоваться) и для новых будут инициализированы случайные веса. В заключении предлагается обучить модель, что мы и сделаем прямо сейчас.

После того, как мы загрузили модель, мы можем определить `Trainer` и передать туда нужные объекты: `model`, `training_args`, обучающую и валидационную выборки, `data_collator` и `tokenizer`

```
from transformers import Trainer

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
)
```

Заметьте, когда вы передали `tokenizer` как в примере выше, значение по умолчанию для `data_collator` в `Trainer` будет `DataCollatorWithPadding` таким, как определено выше, так что эту строку (`data_collator=data_collator`) можно пропустить в этом вызове.

Для fine-tuning модели на нашем датасете мы просто должны вызвать метод `train()` у `Trainer`:

```
trainer.train()
```

Это запустит процесс дообучения (который должен занять несколько минут на GPU) и будет распечатывать значение лосса каждые 500 итераций. Однако эти значения не скажут нам, насколько хорошо или плохо модель работает. И вот почему:

1. Мы не сообщили `Trainer`, что необходимо проводить валидацию: для этого нужно присвоить аргументу `evaluation_strategy` значение `"steps"` (валидировать каждые `eval_steps`) или `"epoch"` (валидировать по окончании каждой эпохи).
2. Мы не указали `Trainer` аргумент `compute_metrics()` – функцию для вычисления метрики на валидационной части (в таком случае в процессе валидации будет только распечатываться значение лосса, что не очень информативно).

Валидация

Давайте посмотрим как мы можем создать и использовать в процессе обучения полезную функцию `compute_metrics()`. Функция должна принимать на вход объект `EvalPrediction` (именованный кортеж с полями `predictions` и `label_ids`) и возвращать словарь, где ключи - названия метрик, а значения - оценки этих метрик. Чтобы получить предсказания, мы можем использовать функцию `Trainer.predict()`:

```
predictions = trainer.predict(tokenized_datasets["validation"])
print(predictions.predictions.shape, predictions.label_ids.shape)
```

```
(408, 2) (408,)
```

Результат функции `predict()` - другой именованный кортеж с полями `predictions`, `label_ids` и `metrics`. Поле `metrics` будет содержать значение лосса на нашем датасете и значения метрик. После реализации функции `compute_metrics()` и передачи ее в `Trainer` поле `metrics` также будет содержать результат функции `compute_metrics()`.

Как можно заметить, `predictions` - массив 408 x 2 (408 - число элементов в датасете, который мы использовали). Это логиты для каждого элемента нашего датасета, переданного в `predict()` (как вы видели в предыдущей главе все модели Трансформеров возвращают логиты). Чтобы превратить их в предсказания и сравнить с нашими лейблами, нам необходимо узнать индекс максимального элемента второй оси:

```
import numpy as np

preds = np.argmax(predictions.predictions, axis=-1)
```

Теперь мы можем сравнить эти предсказания с лейблами. Для создания функции `compute_metric()` мы воспользуемся метриками из библиотеки 🤗 Evaluate. Мы можем загрузить подходящие для датасета MRPC метрики так же просто, как мы загрузили датасет, но на этот раз с помощью функции `evaluate.load()`. Возвращаемый объект имеет метод `compute()`, который мы можем использовать для вычисления метрики:

```
import evaluate

metric = evaluate.load("glue", "mrpc")
metric.compute(predictions=preds, references=predictions.label_ids)
```

```
{'accuracy': 0.8578431372549019, 'f1': 0.8996539792387542}
```

У вас эти результаты могут отличаться ввиду случайной инициализации параметров модели. Тут мы можем увидеть точность 85.78% и F1 89.97% на валидационной части выборки. Эти метрики используются для валидации результатов на MRPC датасете на бенчмарке GLUE. В таблице в статье о BERT указано значение F1 оценки в 88.9 для базовой модели. Это была оценка для варианта модели `uncased`, а мы использовали `cased`, этим и объясняется более хороший результат.

Собирая вместе все фрагменты выше, мы получим нашу функцию `compute_metrics()`:

```
def compute_metrics(eval_preds):
    metric = evaluate.load("glue", "mrpc")
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

Чтобы увидеть эту функцию в действии после каждой эпохи ниже мы определим еще один `Trainer` с функцией `compute_metrics()`:

```
training_args = TrainingArguments("test-trainer", evaluation_strategy="epoch")
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)
```

Обратите внимание, что мы создали новый объект `TrainingArguments` с собственным `evaluation_strategy` равным `"epoch"` и новой моделью - иначе мы бы продолжили обучать модель, которая уже является обученной. Чтобы запустить обучение заново, надо выполнить:

```
trainer.train()
```

На этот раз будет распечатываться валидационный лосс и метрики по окончанию каждой эпохи обучения. Напомним, что полученные значения точности и F1 могут не полностью совпадать с приведенными в примере из-за случайной инициализации слоев модели, но порядок должен быть примерно таким же.

`Trainer` может работать с несколькими GPU или TPU и предоставляет множество опций, например применение техники mixed-precision (установите `fp16 = True` в аргументах). Подробно об опциях мы поговорим чуть в Главе 10.

На этом введение в fine-tuning с использованием API `Trainer` подошло к концу. Пример того, как сделать это же для наиболее распространенных задач NLP мы рассмотрим в Главе 7, а сейчас взглянем на то, как реализовать то же самое на чистом PyTorch.

> ✏️ **Попробуйте!** Произведите fine-tuning модели на датасете GLUE SST-2 с использованием препроцессинга из раздела 2.

<> Update on GitHub