3.2-a-full-training

Giriş

Bu kısımda, önceki bölümde Trainer sınıfı ile yaptığımız eğitimi, manual olarak gerçekleştireceğiz. Kısa bir özet aşağıdaki gibidir:

```
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)
```

Prepare for training

Training loop'a başlamadan önce, birkaç objenin tanımlanması gerekir. Birincisi dataloader'lardır ve batch'ler üzerinde iterasyon yaparken kullanılacaktır. Bundan önce `tokenized_datasets` üzerine post-processing işlemi uygulanabilir. Yapılması gerekenler şu şekildedir:

- Modelin kabul etmediği sütunların kaldırılması (sentence1 ve sentence2 gibi).
- label → labels dönüşümü yapılır. Çünkü model labels adlı sütunu input olarak bekler.
- Dataset için format belirlenir çünkü liste yerine PyTorch tensor'e ihtiyaç duyulacaktır.

`tokenized_datasets` her bir adım için bir metod barındırır:

```
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
```

Yukarıdaki adımın sonunda elimizdeki column'lar şu şekilde listelenebilir:

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

Bu aşamadan sonra dataloader'lar tanımlanabilir.

```
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
)
```

Data processing aşamasında bir hata olmadığını şu şekilde doğrulayabiliriz:

```
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])}
```

`shuffle = True` olarak belirlendiği için, bu boyutlar biraz farklı olabilir çünkü batch'e göre maksimum uzunluk belirleniyor. Şu aşamada pre-processing işlemini tamamlamış sayılırız. Önceki aşamalarda yaptıklarımızı yeniden yapabiliriz:

```
from transformers import AutoModelForSequenceClassification

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

Her şeyin kusursuz çalıştığından emin olmak için, modele bir batch'leri verebiliriz:

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

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

HuggingFace Transformer modelleri, labels sağlandığı zaman loss değerlerini ve logit'leri döndürecektir.

Bir train loop oluşturmak için neredeyse hazırız. Sadece iki şey eksik: bir optimizer ve learning rate scheduler. Trainer sınıfını replike etmeye çalıştığımız için, bu örnekte Trainer'deki default değerleri kullanacağız.

Trainer, `AdamW` optimizerini kullanır. Adam ile aynıdır, ekstra olarak **weight decay regularization** adlı konsepti barındırır.

```
from transformers import AdamW

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

Learning rate scheduler, default olarak maksimum değer olan 5e-5'ten lineer olarak azaltım yapacak şekilde ayarlanır. Düzgün bir biçimde tanımlayabilmek için, training steps miktarını bilmemiz gerekir. Bu sayı şu şekilde hesaplanabilir: epochs * batches. Bu aynı zamanda training dataloader'in uzunluğuna eştir. Default olarak Trainer sınıfı 3 adet epoch kullanmaktadır. Bu yüzden aşağıdaki gibi devam edeceğiz.

```
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)
```

The training loop

Son bir konu daha var: Eğer bir GPU'ya erişiminiz varsa, bunu kullanabilmek için bir device tanımlamamız gerekecektir:

```
import torch

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

"""
[in 1]: device
[out 1]: device(type='cuda')
"""
```

Şu aşamda train işlemi başlatmaya hazırız. Training işleminin ne zaman biteceğini kestirebilmek için bir progress bar ekleyebiliriz. Bunun için `tqdm` kütüphanesinden faydalanılabililr.

```
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)
```

Bu loop, raporlama adına herhangi bir kod içermez ve temel bir loop'tur. Bu iş için bir evaluation loop eklenebilir.

The evaluation loop

Daha önce de bahsedildiği gibi, HuggingFace bir Evaluate kütüphanesi içerir ve `metric.compute()` metodununun nasıl kullanılacağından bahsedilmişti. Aşağıda bir prediction loop oluşturulmuş ve aslında add_batch her çalıştığında, o batch'e ait metrikler otomatik olarak metric içine kaydedilmiştir. `metric.compute()` ile sonuç metriği elde edebiliriz.

```
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}
```

Supercharge your training loop with 🤗 Accelerate

Yukarıdaki training loop bira det CPU veya GPU üzerinde sorunsuz çalışacaktır. Fakat Accelerate kütüphanesi, birkaç düzenleme ile bize dağıtık eğitim fırsatı sunar. Bu sayede birçok cihazda, GPU ve TPU'lar üzerinde eğitime devam edebiliriz. Örnek bir manuel training loop aşağıdaki gibi görülebilir:

```
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)
```

İlk aşamada gerekli kütüphaneler ekleniyor. Daha sonra bir `Accelerator()` objesi oluşturuluyor ve cihazdaki özelliklere göre bir setup oluşturuyor. Device otomatik olarak eklendiği için device ile ilgili satırlar silinebilir.

Ana değişiklik, dataloader'ların modele gönderildiği bölümde ve accelerator.prepare() yazan yerde gerçekleşmiştir. Bu sayede dağıtık eğitim işlemi için gerekli konteyner kurulmuş olur. Dönüştürülmüş kod aşağıdaki gibi olacaktır:

```
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)
```

bu kodu `train.py` dosyasında sakladıktan sonra, dağıtık eğitim sürecini başlatmak için aşağıdaki adımları takip edebilirsiniz:

1. `accelerate config` : Burada birkaç soru sorulacaktır.
2. `accelerate launch` `train.py` : Eğtimi başlatacaktır.

Eğer yukarıdakileri bir notebook içinde yapacaksanız, yukarıdakileri bir `training_function` içine kopyaladıktan sonra bu bloğu çalıştırabilirsiniz:

```
from accelerate import notebook_launcher

notebook_launcher(training_function)
```

Daha fazla örnek, HuggingFace Accelerate repo üzerinde bulunabilir.