3-fine-tuning-a-pretrained-model

Giriş

Önceki kısımda, tokenizer ve pretrained modellerin nasıl kullanılacağı üzerinde duruldu. Fakat özel bir veri seti ile çalışmak için birkaç adımın daha tamamlanması gereklidir. Bu kısımda bu adımlar üzerinde durulacaktır. Bölümün çıktıları şunlardır:

- Hub üzerinden büyük dataset hazırlamak.
- `Trainer` high-level API ile model fine-tune emek.
- Özelleşmiş train loop kullanmak.
- Herhangi bir dağıtık sistem üzerinde, HuggingFace Accelerate kütüphanesi kullanarak özelleşmiş train loop çalıştırmak.

Eğitim sonucunda elde edilen checkpoint'leri Hugging Face Hub'a yükleyebilmek için huggingface.co üzerinde bir hesap oluşturmanız gerekmektedir.

Processing the data

Önceki kısımdaki üzerinden devam edilecek olursa, aşağıdaki kod bir sequence classifier modelini nasıl train edebileceğinizi gösterir:

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

# Önceki kısım ile aynı
checkpoint = "bert-case-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AuoModelForSequenceClassificaion.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")

# This is new
batch["labels"] = torch.tensor([1, 1])

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

```
loss.backward()
optimizer.step()
```

Sadece iki adet cümle ile eğitim gerçekleştirmek, müthiş bir performans sağlamayacaktır. Daha iyi sonuçlar için daha büyük veri setleri kullanılmalıdır.

Bu kısımda, MRPC (Microsoft Research Paraphrase Corpus) veri seti kullanılacaktır. Bu veri seti, 5801 adet 5801 çift cümleden oluşur, ve verilen çiftlerin aynı anlama gelip gelmediğini belirten etiketler barındırır. Bu veri setinin seçilme nedeni küçük olmasıdır, böylece eğitim hızlıca tamamlanabilir.

Hub, sadece modellerden ibaret değildir, birçok dilden birçok veri seti içerir. Şimdilik MRPC ile devam edilecektir. Bu veri seti GLUE benchmark'ın barındırdığı 10 veri setinden biridir. HuggingFace Datasets kütüphanesi, veri setini indirip cache'lemek için kolay bir yol sunar. MRPC veri setini aşağıdaki gibi indirip saklayabiliriz:

```
from datasets import load_dataset

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

`raw_datasets` değişkeni aşağıdaki gibi gözükmektedir:

```
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
    })
})
```

Görülebildiği gibi, `DatatsetDict` objesi training, testing ve validation setlerini içerir. Her biri birkaç sütun içerir (sentence1, santence2, label ve idx). `num_rows` değişkeni de train, test ve validatiton setleri içindeki sütun sayısını belirtir. İndirilen veri seti `~/.cache/huggingface/datasets` yolunda saklanır.

Örnek bir cümleyi incelemek için `raw_datasets` objesini bir sözlük gibi kullanabiliriz. Örneğin:

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

Aşağıdaki sonucu üretecektir:

```
{'idx': 0,
 'label': 1,
 'sentence1': 'Amrozi accused his brother , whom he called " the witness " , of delibe
rately distorting his evidence .',
 'sentence2': 'Referring to him as only " the witness " , Amrozi accused his brother o
f deliberately distorting his evidence .'}
```

Etiketler halihazırda tamsayı olarak girilmiştir bu yüzden bir ön işleme gerek yoktur. Hangi sayının hangi etikete denk geldiğini öğrenebilmek için `raw_train_dataset.features` özelliğini çağırabiliriz:

```
{'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)}
```

Arkaplanda, label'in tipi `ClassLabel` 'dir ve integer'ların etiket karşılıklarını saklayan değişkenin ismi `names`'dir.

Preprocessing a dataset

Preprocess işlemi, metnin sayısal değerlere dönüşürülmesi işlemidir ve tokenizer'lar tarafından gerçekleştirilir. Tokenizer'lar bir cümle ya da cümle listesiyle çalışabilirler. Verilen veri seti için tüm sentence1 ve senetnce2 örneklerini aşağıdaki gibi tokenize edebiliriz:

```
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"])
```

Fakat modele direkt olarak iki farklı sequence verip bunların anlamsal olarak aynı olup olmadığını tahminlemesini bekleyemeyiz. Bunun yerine verilen sequence çiftlerini uygun bir preprocess işlemine tabi tutmak gerekir. Tokenizer'lar bir sequence çiftini girdi olarak kabul edip BERT modelinin beklediği formata uygun hale getirebilir:

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

Sonuç olarak aşağıdaki input'u elde ederi

```
{
  '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]
}
```

Burada yeni olarak `token_type_ids` key'ini görürüz. İfade edilmeye çalışılan şey hangi token'in hangi sequence ait olduğudur. Eğer verilen token'ları decode edersek, aşağıdaki listeyi elde ederiz:

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

Yukarıdaki kod, aşağıdaki çıktıyı üretecektir.

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

[CLS] ve ilk [SEP] dahil, aradaki bütün token'lar tip id olarak 0 almıştır, geri kalanlara da 1 değeri atanmıştır.

Farklı bir checkpoint için bu token_type_ids özelliği üretilmeyebilir. Örneğin DistilBERT için bu özellik tokenizer tarafından üretilmez. BERT, bu token_type_ids özelliğini kabul eder çünkü bu model next sentence prediction görevi için eğitilmiştir. Bu durumda, tokenleri aşağıdaki gibi elde edebiliriz:

```
tokenized_dataset = tokenizer(
    raw_datasets["train"]["sentence1"],
    raw_datasets["train"]["sentence2"],
```

```
    padding=True,
    truncation=True,
)
```

Bu kod kusursuz çalışacaktır ancak bir dictionary return etmesi bir dezavantajdır ve bütün veri setini saklayabilecek yeterli RAM varsa çalışacaktır.

Bu problemi `Dataset.map()` metodu ile aşabiliriz. Eğer tokenizasyon aşamasından önce bir preprocessing ihtiyacı duyarsak, bu yöntem konuda da faydalı olabilir. `map()` fonksiyonu veri setindeki her elemana bir fonksiyonu uygulayarak çalışır. Uygulanacak fonksiyonu aşağıdaki gibi yazabililriz:

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

Yazılan fonksiyon bir dictionary alır ve yeni bir dictionary return eder. Burada example dictionary birkaç sample içerebilir, çünkü daha önce anlatıldığı gibi tokenizer birkaç sample için de çalışabilir. Bu özellik map() metodunu çağırdığımız zaman `batched=True` opsiyonunu çalıştırmamıza imkan sunar ve tokenizasyon işlemini oldukça hızlandırır.

Tokenizer içindeki `padding` argümanının kaldırıldığını not ediniz. Çünkü bütün sample'ları maksimum uzunluğa göre pad'lemek verimli değildir. Batch'leri oluştururken padding yapmak daha verimlidir çünkü bütün veri seti üzerindeki maksimum uzunluk yerine batch içindeki maksimum uzunluğa göre padding yapılır. Bu oldukça zaman kazandırabilir.

Bütün veri setini tokenize ettmek için aşağıdaki kod kullanılabilir:

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

Yukarıdaki kod çalıştığı zaman dataset üzerine birkaç alan daha eklendiği gözlenebilir:

```
DatasetDict({
    train: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'senten
ce2', 'token_type_ids'],
        num_rows: 3668
    })
    validation: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'senten
```

```
ce2', 'token_type_ids'],
        num_rows: 408
    })
    test: Dataset({
        features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'senten
ce2', 'token_type_ids'],
        num_rows: 1725
    })
})
```

Son aşamada, padding işlemlerinin tamamlanması gerekir. Batch içindeki en uzun cümle referans alınarak padding yapılmasına dynamic padding denir.

Dynamic padding

Sample'ları bir batch içine toplamakla görevli olan fonksiyonun ismi collate fonksiyonudur. Bu fonksiyon, PyTorch içindeki DataLoader objesini oluştururken bir argüman olarak verilebilir. Güncel durumda bu mümkün değildir çünkü sahip olduğumuz input'ların boyutları aynı değildir.

Başlamak için collate fonksiyonunun tanımlanması gerekir. Fonksiyonun amacı batch'lemek istediğimiz verilere uygulanacak olan padding miktarını tayin etmektir. HuggingFace Transformers kütüphanesi tarafından sağlanan `DataCollatorWithPadding` fonksiyonu bu işe yarayabilir. Girdi olarak bir tokenizer'a ihtiyaç duyacaktır (modelin paddingi başa mı sona mı kabul etiğini ve modelin kabul ettiği padding token'ı bilmesi gerektiği için). Aşağıdaki gibi oluşturulabilir:

```
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Çalışıp çalışmadığını test etmek için train veri setinden ilk 8 cümleyi çekip uzunluklarına bakabiliriz:

```
samples = tokenized_datasets["train"][:8]
samples = {k: v for k, v in samples.items() if k not in ["idx", "sentence1", "sentence
2"]}
[len(x) for x in samples["input_ids"]]
```

Aşağıdaki sonuç görülebilir:

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

Şu anlaşılabilir: Örneklerin token uzunluğu 32-67 arası değişmektedir. `data_collator`'dan beklenen iş bu sample'ları uygun bir şekilde batch'lemesidir. bunu sağlamak için aşağıdaki kodu çalıştırabiliriz:

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

Sonuç olarak üretilen dictionary'de, batch size'ın 67 olarak belirlendiği görülebilir.

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