3.1-fine-tuning-with-trainer-api

Giriş

Huggingface Transformers kütüphanesindeki `Trainer` sınıfı sağlanan pretrained modellerden herhangi birisini, özel verisetleriyle eğitmeye imkan tanır. Önceki preprocessing adımlarını tamamladıysanız, fine-tune işlemlerinden birkaç adım uzaklıktasınız demektir. Buradaki asıl zorluk `Trainer.train()` oratmını hazır hale getirmektir çünkü bir CPU üzerinde oldukça yavaş çalışacaktır. Eğer bir GPU'ya erişiminiz yoksa, Google Colab üzerinden bir TPU ya da GPU'ya erişebilirsiniz.

Önceki örneklere göz attığınızı varsayarak, aşağıdaki örnek üzerinden bir tekrar yapabiliriz:

```
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)
```

Training

Trainer objesi tanımlamadan önceki ilk iş `TrainingArguments` sınıfından bir obje oluşturmaktır. Burada Trainer sınıfının kullanacağı bütün hiperparametreleri belirtebiliriz. Zorunlu olarak istenen argüman, eğitilen modelin saklanacağı dizinin yoludur. Geri kalan değerler defaul olarak kullanılabilir, temel fine-tune operasyonları için yeterince iyi çalışmaktadır.

```
from transformers import TrainingArguments

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

İkinci aşama modeli tanımlamakır. Önceki kısımlardaki gibi bu kısımda da `AutoModelForSequenceClassification` sınıfı kullanılacaktır ve modelin iki adet label için sınıflandırma yapması beklenecektir.

```
from transformers import AutoModelForSequenceClassification

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

Burada bir uyarı almanız olasıdır çünkü BERT modeli aslında bir sequence çifti sınıflandırmak için eğitilmemiştir. Bu yüzden prertained modelin head'i görmezden gelinecektir ve yerine yenisi eklenecektir. Uyarıda bazı ağırlıkların kullanılmayacağını, bazılarının da rastege değerler ile başlatılacağı belirtilir.

Base modeli yükledikten sonra, Trainer sınıfından bir obje türetebiliriz. Constructor için argüman olarak `model` ve `training_args` sağlayabiliriz. Aynı zamanda `train` ve `validasyon` veriselerine ek olarak `tokenizer` ve `data_collator` objelerini de sağlayabiliriz.

```
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,
)
```

Fine tune işlemini başlatmak için `trainer.train()` metodunu kullanabilirsiniz. Fine tune işlemi başlamış olacaktır. GPU ile birkaç dakika sürecektir ve her 500 adımda training loss bilgisini çıktı olarak paylaşacaktır. Modelin ne kadar başarılı olduğu bilgisi paylaşılmayacaktır çünkü:

- Trainer'a training esnasında evaluation yapmasını belirtmedik. Bunu sağlamak için `evaluation_strategy` argümanı `steps` ya da `epoch` olarak belirtilmelidir.
- Yukarıdaki evaluation'u yapması için Trainer'a `compute_metrics()` fonksiyonunu sağlamadık.

Evaluation

`compute_metrics()` fonksiyonu kullanılarak train fonksiyonunu yeniden çalıştırmayı deneyelim. Fonksiyon, bir `EvalPrediction` (aslında bir named tuple'dır, predictions ve label_ids alanlarından oluşur) objesi kabul eder ve string → float işaretlemesi yapan bir dictionary döndürür. Tahminleri alabilmek için, `Trainer.predict()` metodunu kullanabiliriz.

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

Çıktısı aşağıdaki gibi olacaktır:

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

`predict()` çıktı olarak başka bir named tuple döndürecektir ve 3 adet alanı vardır:

- predictions
- label_ids
- metrics

`metrics` alanı, dataset içindeki loss'u ve bazı zaman metriklerini içerir. Görüldüğü gibi tahminler iki boyutlu bir array'dir ve 408 verilen veri setinin uzunluğudur. Her veri noktası için bir logit döndürülmüştür. Bu değerleri label'lara dönüştürebilmek için aşağıdaki kodu çalıştırabiliriz:

```
import numpy as np

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

Yukarıdaki kodda, `preds` adında bir numpy array oluşturulur ve bu array, tahmin edilen label'ları içerir. Şimdi bu değişkeni, elimizdeki label'lar ile karşılaştırabiliriz. `compute_metrics()` fonksiyonunu inşa edebilmemiz için HuggingFace Evaluate kütüphanesini kullanacağız. MRPC veri setiyle ilişkili metrikleri yükleyebiliriz. Bunun için `evaluate.load` fonksiyonu kullanılabilir. Bu obje bir `compute()` metodu içerir ve metrik hesaplama işinde kullanılabilir:

```
import evaluate

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

Çıktı olarak aşağıdakini görürüz:

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

Bu sonuç, model random olarak initialize edildiği için değişken olabilir. MRPC dataset için iki adet metrik göz önünde bulundurulmuştur ve bu metrikler GLUE benchmark üzerindeki performansını belgeler.

BERT paper içinde, F1 skoru %88.9 olarak sunulmuştur ve bu uncased model içindir. Örnekte, cased model kullandık ve daha iyi çıkan F1 skorunun açıklaması bu olabilir. Toparlarsak, compute_metrics fonksiyonu aşağıdaki gibi tanımlanabilir:

```
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)
```

Her epoch için bu metriklerin raporlarını edinmek için Trainer objesine bu fonksiyonu argüman olarak tanıtabiliriz:

```
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,
)
```

`TrainingArguments` objesi oluşturduk ve `evaluation_strategy` argümanını `epoch` olarak işaretledik. Yeni bir model instance oluşturduk, çünkü eğitimi eski model üzerinden devam ettirmek istemedik.

Yeni durumda, her epoch sonunda training loss metriğini görebileceğiz. `Trainer`, çoklu GPU ve TPU üzerinde çalışma yeteneğine sahiptir, mixed-precision training gibi birçok özelliği destekler(fp16 = True eşitlenerek gerçekleştirilebilir).

Bu kısım, Trainer API kullanarak fine-tuning işlemine giriş olarak değerlendirilebilir. Aynı işlemi saf PyTorch ile nasıl yapılacağı sonraki bölümde anlatılacaktır.