

Huggingface Transformers(1)

课程介绍

1. Transformer models

什么是自然语言处理?

pipeline(不常用)

Transformers

Transformer 模型由两部分组成:

语言模型:

Architectures vs. checkpoints

2. Using 😊 Transformers

pipeline的幕后原理

使用tokenizer进行预处理

Full model

AutoModel架构的输出

```
from transformers import AutoModelForSequenceClassification
```

处理model的输出， 转化为概率

models

加载model

保存model

Tokenizers

word-based

character-based

subword-based

tokenizer

处理多个sequences

tokenizer处理完整示例

3. Fine-tuning a pretrained model

处理数据

使用tokenizer处理一对句子 (模型需要一对输入, 比如判断这对句子是否意思相同)

git@github.com:hihihe/HuggingFace_Course.git

使用tokenizer处理多对句子

数据预处理完整代码

Fine-tuning a model with the Trainer API

使用pytorch训练

使用😊 Accelerate加速

4. Sharing models and tokenizers

课程介绍

官方链接



经典模型的处理能力：

1. RNN 10–40
2. LSTM 50–100
3. Transformer BERT 512 GPT 1024
4. Transformer–XL,LXNet 3000–5000

word embedding

预训练模型 fine tune 方法小结

1. Transformer models

什么是自然语言处理？

- Classifying whole sentences: 评论情感分析，检测电子邮件是否为垃圾邮件，确定句子在语法上是否正确或判断两个句子在逻辑上是否相关。
- Classifying each word in a sentence: 识别句子的语法成分（名词、动词、形容词）或命名实体（人、地点、组织）。
- Generating text content: 用自动生成的文本完成提示，用屏蔽词填充文本中的空白。
- Extracting an answer from a text: 给定问题和上下文，根据上下文中提供的信息提取问题的答案。

git@github.com: hiih/he/Hugging_Face_Course.git

- Generating a new sentence from an input text: 将文本翻译成另一种语言，文本摘要。

pipeline(不常用)

官方示例代码

💡 Transformers library中最基本的对象是pipeline. 它将模型与其必要的预处理和后处理步骤连接起来，使我们能够直接输入任何文本并获得可理解的答案：

```
▼ Python | 复制代码
1  from transformers import pipeline
2
3  classifier = pipeline("sentiment-analysis")
4  classifier("I've been waiting for a HuggingFace course my whole life.")
5
6  # 输出结果为：
7  [{'label': 'POSITIVE', 'score': 0.9598047137260437}]
```

默认情况下，pipeline会选择一个特定的预训练模型，该模型已针对英语情感分析进行了微调。创建classifier对象时，将下载并缓存模型。如果您重新运行该命令，则将使用缓存的模型，无需再次下载模型。

将文本输入给pipeline，pipeline的处理涉及到以下三个主要步骤：

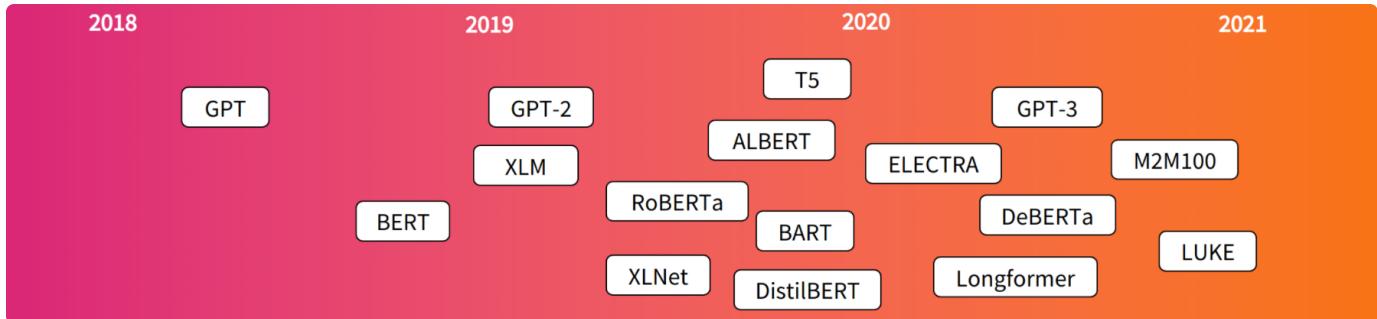
- 文本被预处理为model可以理解的格式。
- 预处理的输入被传递给model。
- model的预测是经过处理，转化为人可以理解的形式。

现在可以使用的pipelines

- feature-extraction (get the vector representation of a text)fill–mask
- ner (named entity recognition)
- question–answering
- sentiment–analysis
- summarization
- text–generation
- translation
- zero–shot–classification

Transformers

Transformer architecture是在2017年六月推出，当初的研究用于翻译任务。随后推出了几个有影响力的模型，如下图：



预训练模型的详细介绍，它们可以分为三类：

类 GPT (也称为 *auto-regressive* Transformer 模型)

类 BERT (也称为 *auto-encoding* Transformer 模型)

BART/T5 类 (也称为 *equence-to-sequence* Transformer 模型)

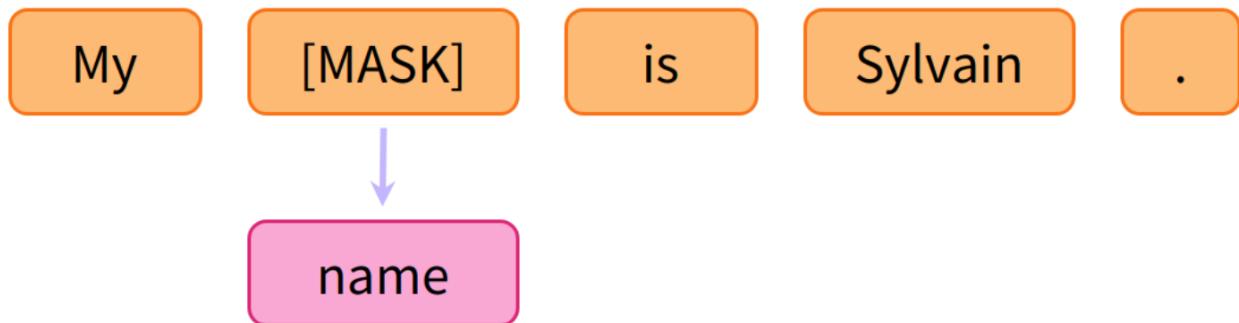
Transformer 模型由两部分组成：

- 编码器（左）：编码器接收输入并构建其表示（其特征）。这意味着模型经过优化以从输入中获取理解。
- 解码器（右）：解码器使用编码器的表示（特征）以及其他输入来生成目标序列。这意味着模型针对生成输出进行了优化。

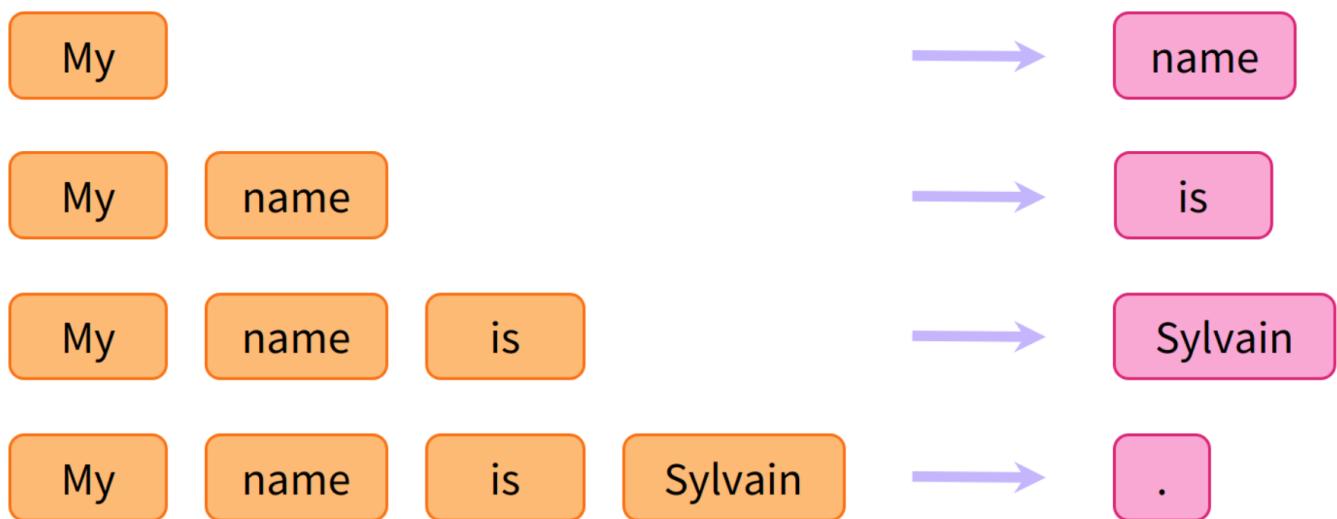
<p>encoder models</p> <p><i>auto-encoding models</i></p> <p>使用 Transformer 模型的编码器。在每个阶段，注意力层都可以访问初始句子中的所有单词。这些模型通常被描述为具有“双向”注意力。</p>	<p>这些模型的预训练通常围绕着以某种方式破坏给定的句子（例如，通过随机屏蔽其中的单词）并让模型找到或重建初始句子。</p>	<p>ALBERT, BERT, DistilBERT ,</p> <p>ELECTRA, RoBERTa</p>	<p>句子分类，命名实体识别（以及更一般的单词分类）和提取式问答。</p>
<p>decoder modles</p> <p><i>auto-regressive models</i></p> <p>使用 Transformer 模型的解码器。在每个阶段，对于给定的单词，注意力层只能访问句子中位于它之前的单词。</p>	<p>解码器模型的预训练通常围绕预测句子中的下一个单词。</p>	<p>CTRL, GPT, GPT-2, Transformer XL</p>	<p>文本生成。</p>
<p>encoder-decoder models</p> <p><i>sequence-to-sequence models</i></p> <p>使用 Transformer 架构的两个部分。在每个阶段，编码器的注意力层可以访问初始句子中的所有单词，而解码器的注意力层只能访问位于输入中给定单词之前的单词。</p>	<p>这些模型的预训练可以使用户编码器或解码器模型的目标来完成，但通常涉及一些更复杂的东西。例如，T5是通过用一个掩码特殊单词替换随机范围的文本(可以包含几个单词)进行预训练的，然后目标是预测这个掩码单词替换的文本。。</p>	<p>BART, T5, Marian, mBART</p>	<p>摘要、翻译或生成式问答。</p>

语言模型：

MLM: *masked language modeling* 预测句子中的掩码词。



CLM: *causal language modeling* 根据当前的文本预测下一个词。



Architectures vs. checkpoints

- **Architecture:** 这是模型的骨架—模型中每一层和每个操作的定义。
- **Checkpoints:** 这些是将在给定架构中加载的权重。
- **Model:** 这是一个总称，不像“架构”或“检查点”那样精确：它可以同时表示两者。

2. Using 😊 Transformers

[官方示例代码](#)

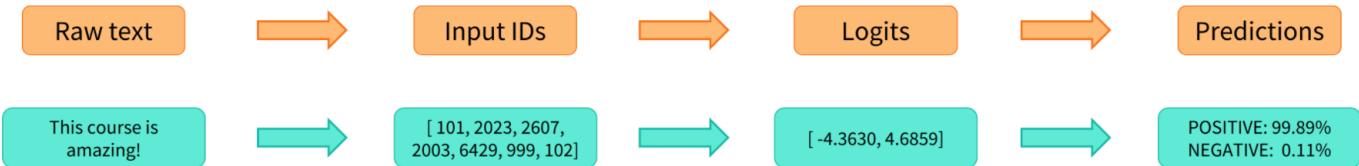
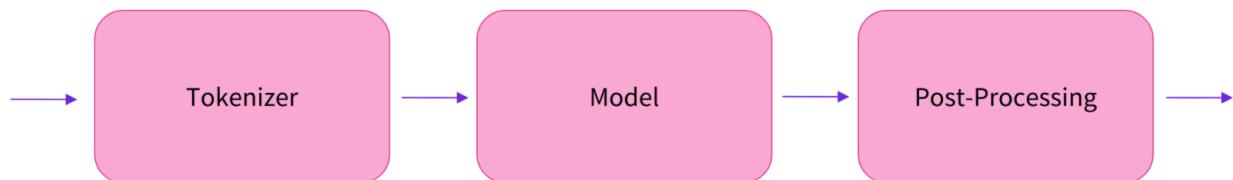
pipeline的背后原理

pipeline将三个步骤组合在一起：预处理、通过模型传递输入和后处理：[流程见下图](#)

```

1 from transformers import pipeline
2
3 classifier = pipeline("sentiment-analysis")
4 classifier([
5     "I've been waiting for a HuggingFace course my whole life.",
6     "I hate this so much!",
7 ])
8
9
10 # 输出结果:
11 [{"label": "POSITIVE", "score": 0.9598047137260437},
12 {"label": "NEGATIVE", "score": 0.9994558095932007}]

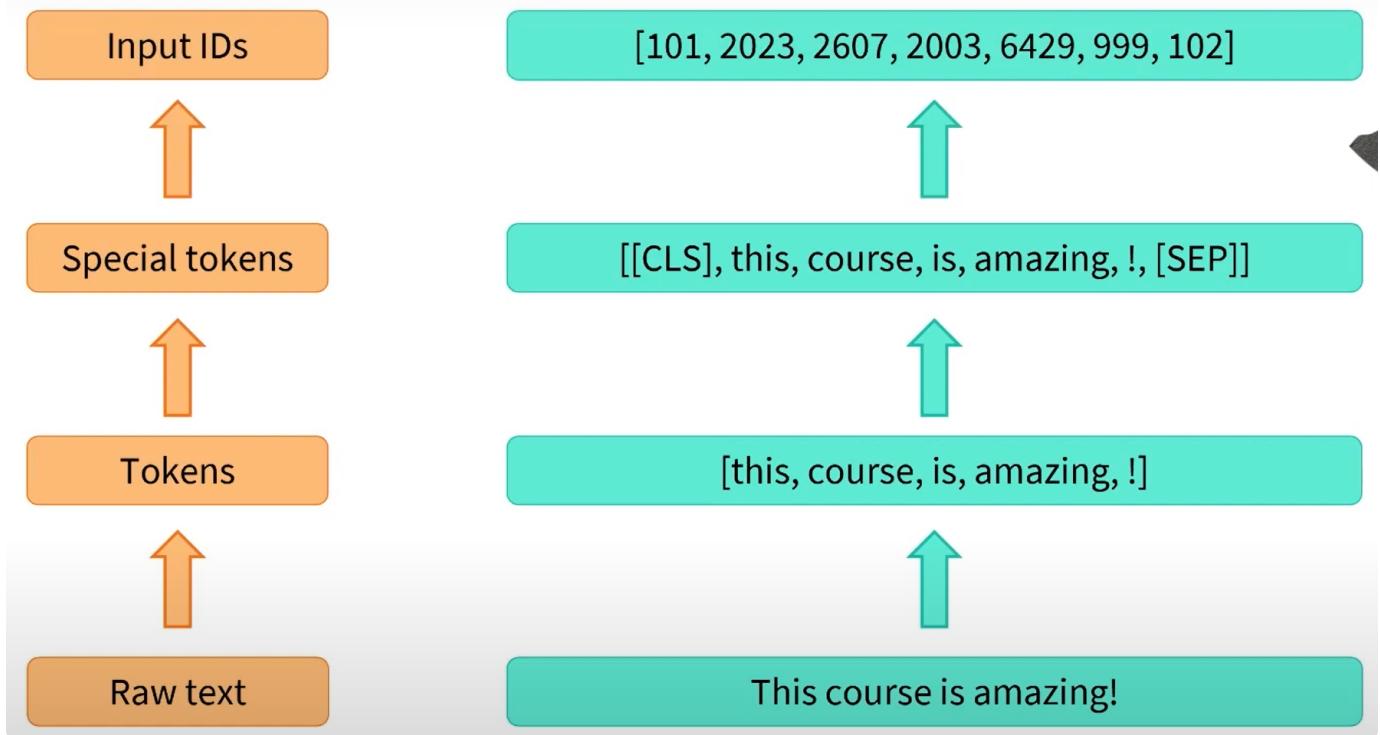
```



使用tokenizer进行预处理

与其他神经网络一样，Transformer 模型不能直接处理原始文本，因此我们管道的第一步是将文本输入转换为模型可以理解的数字（如下图流程）。为此，我们使用了一个tokenizer，它将负责：

- 将输入拆分为称为标记的单词、子词或符号（如标点符号）。
- 将每个标记映射到一个整数。
- 添加可能对模型有用的其他输入。



所有这些预处理都需要以与模型预训练时完全相同的方式完成，因此我们首先需要从[Model Hub](#)下载该信息。为此，我们使用**AutoTokenizer**类及其**from_pretrained**方法。使用我们模型的检查点名称，它将自动获取与模型的标记器关联的数据并将其缓存。

由于**sentiment-analysis**管道的默认检查点是**distilbert-base-uncased-finetuned-sst-2-english**（您可以[在此处](#)查看其模型卡），我们运行以下命令：

```

1 from transformers import AutoTokenizer
2
3 checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
4 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
5
6 raw_inputs = [
7     "I've been waiting for a HuggingFace course my whole life.",
8     "I hate this so much!",
9 ]
10
11 # 要指定我们想要返回的张量类型 (PyTorch、TensorFlow 或普通 NumPy) , 我们使用
12 # return_tensors参数:
13 inputs = tokenizer(raw_inputs, padding=True, truncation=True,
14                     return_tensors="pt")
15 print(inputs)
16
17 # 输出结果为:
18 {
19     'input_ids': tensor([
20         [ 101,  1045,  1005,  2310,  2042,  3403,  2005,  1037,  17662,
21         12172,  2607,  2026,  2878,  2166,  1012,  102],
22         [ 101,  1045,  5223,  2023,  2061,  2172,    999,   102,      0,
23         0,      0,      0,      0,      0,      0]
24     ]),
25     'attention_mask': tensor([
26         [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
27         [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
28     ])
29 }
30
31 # 如果没有使用return_tensors参数, 则输出如下 (列表嵌套列表的格式) , 该格式无法直接输入给model
32 # Transformer 模型只接受tensors作为输入。
33 {'input_ids': [[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662,
34         12172, 2607, 2026, 2878, 2166, 1012, 102], [101, 1045, 5223, 2023, 2061,
35         2172, 999, 102, 0, 0, 0, 0, 0, 0]], 'attention_mask': [[1, 1, 1, 1,
36         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]]}

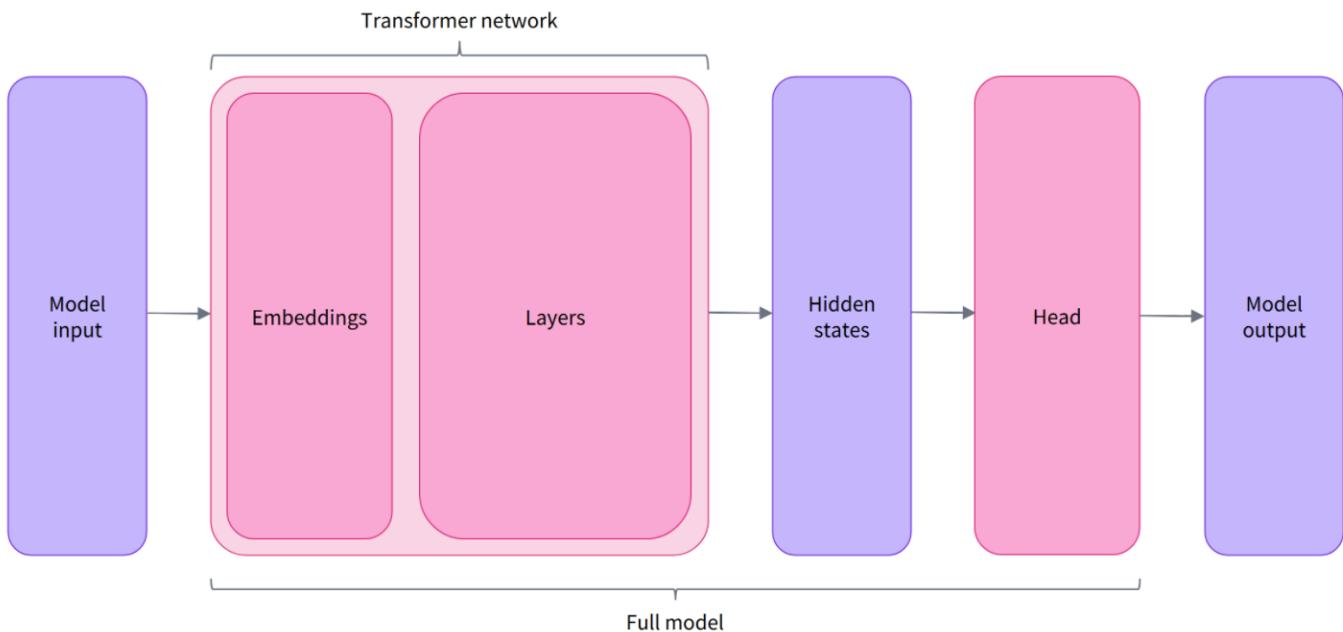
```

输出本身是一个包含两个键的字典，input_ids和attention_mask。

input_ids包含两行整数（每个句子一个），它们是每个句子中标记的唯一标识符。attention_mask是具有与input_ids张量完全相同形状的张量，填充0和1：1表示应注意的相应位置的标记，0表示不应注意的

相应位置的标记(即，模型的 attention layers 应忽略它们)。

Full model



AutoModel架构的输出

注意： 😊 Transformers 模型的输出表现得像namedtuples 或字典。您可以通过属性（就像我们所做的那样）或键 (outputs["last_hidden_state"]) 访问元素，如果您确切地知道要查找的内容在哪里，甚至可以通过索引访问元素outputs[0]。

AutoModel这个架构只包含基本的 Transformer 模块：给定一些输入，它输出我们称之为隐藏层的东西，也称为 *features*。对于每个模型输入，我们将得到一个高维向量，表示Transformer 模型对该输入的上下文理解，可以理解成是做了word embedding。

补充：对于不同架构输出会不一样：

1. `model = AutoModel.from_pretrained("bert-base-chinese")` , 输出为
BaseModelOutputWithPoolingAndCrossAttentions，包
含'last_hidden_state'和'pooler_output'两个元素。其中'last_hidden_state'的形状是 (batch
size,sequence length,768), 'pooler_output'的形状是(batch size,768)。

pooler output是取[CLS]标记处对应的向量后面接个全连接再接tanh激活后的输出。

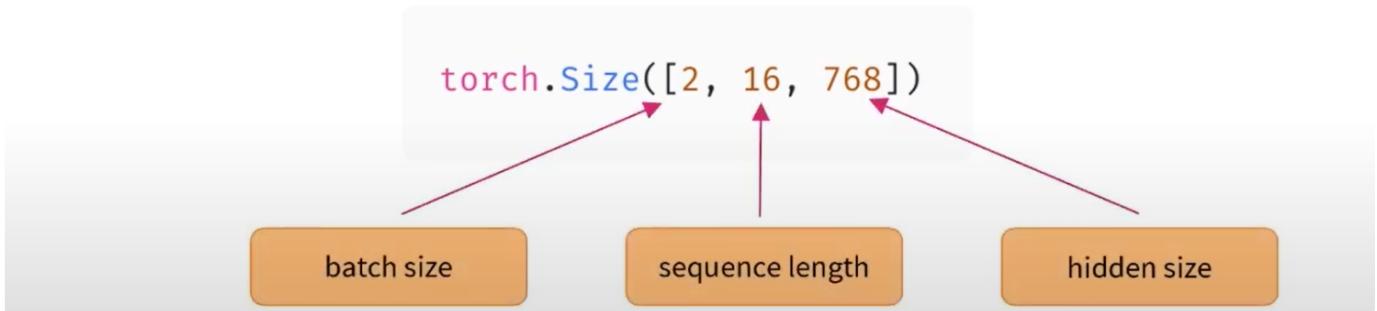
虽然这些隐藏状态本身就很有用，但它们通常是模型另一部分（称为head）的输入。在pipeline那一节中，可以使用相同的体系结构执行不同的任务，是因为这些任务中的每一个都有与之关联的不同头。

2. `model = AutoModelForMaskedLM.from_pretrained("bert-base-chinese")` ,输出为
MaskedLMOutput，包含'logits'元素，形状为[batch size,sequence length,21128],21128
是'vocab_size'。

3. `model = AutoModelForTokenClassification.from_pretrained("bert-base-chinese")`，输出为TokenClassifierOutput，包含'logits'元素，形状为[batch size,sequence length,2]。

Python | 复制代码

```
1  from transformers import AutoModel
2
3  checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
4  model = AutoModel.from_pretrained(checkpoint)
5
6  # inputs见上一个代码示例，model的输入可以是多个参数，但也可以只有input_ids，形状必须是2维张量。
7  outputs = model(**inputs)
8
9
10 print(outputs)
11 # 输出结果为：
12 BaseModelOutput(last_hidden_state=tensor([[[[-0.1798,  0.2333,  0.6321,
13   ..., -0.3017,  0.5008,  0.1481],
14     [ 0.2758,  0.6497,  0.3200,  ..., -0.0760,  0.5136,  0.1329],
15     [ 0.9046,  0.0985,  0.2950,  ...,  0.3352, -0.1407, -0.6464],
16     ...,
17     [ 0.1466,  0.5661,  0.3235,  ..., -0.3376,  0.5100, -0.0561],
18     [ 0.7500,  0.0487,  0.1738,  ...,  0.4684,  0.0030, -0.6084],
19     [ 0.0519,  0.3729,  0.5223,  ...,  0.3584,  0.6500, -0.3883]],
20   [[-0.2937,  0.7283, -0.1497,  ..., -0.1187, -1.0227, -0.0422],
21   [-0.2206,  0.9384, -0.0951,  ..., -0.3643, -0.6605,  0.2407],
22   [-0.1536,  0.8987, -0.0728,  ..., -0.2189, -0.8528,  0.0710],
23   ...,
24   [-0.3017,  0.9002, -0.0200,  ..., -0.1082, -0.8412, -0.0861],
25   [-0.3338,  0.9674, -0.0729,  ..., -0.1952, -0.8181, -0.0634],
26   [-0.3454,  0.8824, -0.0426,  ..., -0.0993, -0.8329, -0.1065]]],
27   grad_fn=<NativeLayerNormBackward>), hidden_states=None,
28   attentions=None)
29
30
31 print(outputs.last_hidden_state.shape)
32 # 输出结果为：
33 torch.Size([2, 16, 768])
```

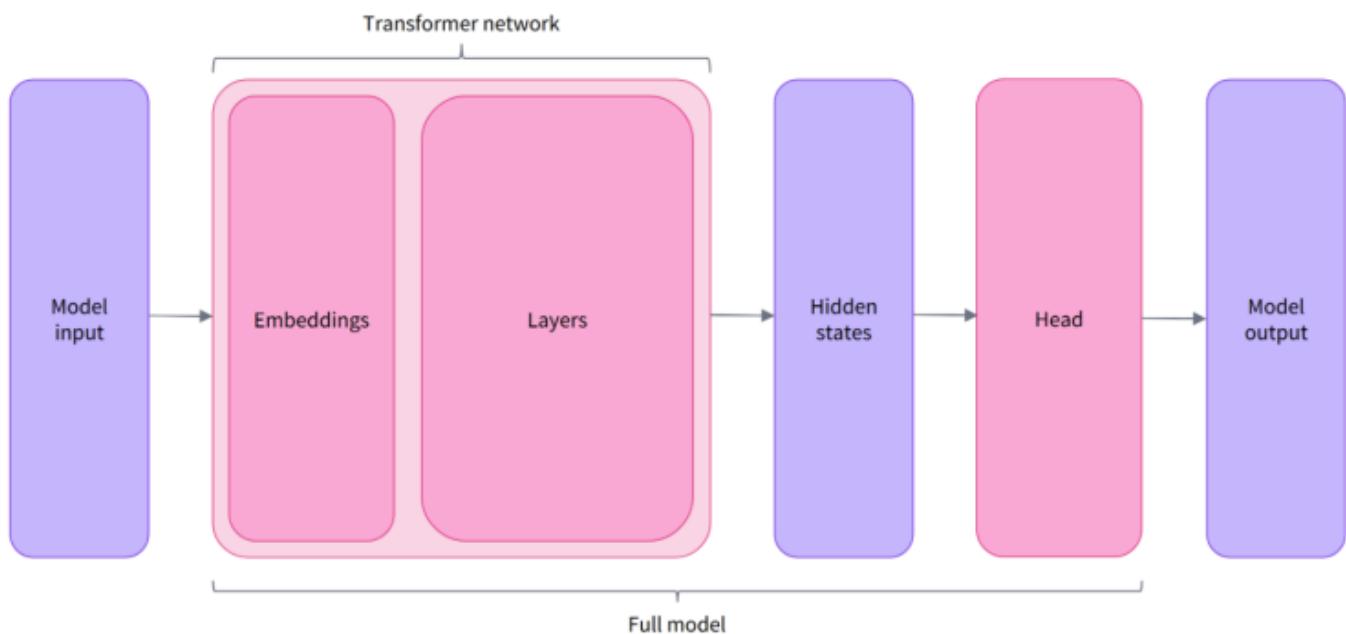


注意： 😊 Transformers 模型的输出表现得像namedtuples 或字典。您可以通过属性（就像我们所做的那样）或键（outputs["last_hidden_state"]）访问元素，如果您确切地知道要查找的内容在哪里，甚至可以通过索引访问元素outputs[0]。

```
from transformers import AutoModelForSequenceClassification
```

Model heads: Making sense out of numbers

模型头将隐藏状态的高维向量作为输入，并将它们投影到不同的维度上。它们通常由一个或几个线性层组成：



Transformer 模型的输出直接送到模型头部进行处理。

在此图中，transformer network由其嵌入层和后续层表示。嵌入层将标记化输入中的每个输入 ID 转换为表示关联标记的向量。随后的层使用注意力机制操纵这些向量以产生句子的最终表示。

😊 Transformers 中有许多不同的架构可用，每一种架构都围绕着处理特定任务而设计。这是一个非详尽列表：

- *Model (retrieve the hidden states)
- *ForCausalLM
- *ForMaskedLM
- *ForMultipleChoice
- *ForQuestionAnswering
- *ForSequenceClassification
- *ForTokenClassification
- and others 😊

对于我们上文的示例，我们将需要一个带有序列分类头的模型（能够将句子分类为正面或负面）。所以，我们不会使用这个AutoModel类，而是使用AutoModelForSequenceClassification：

Python | ⚡ 复制代码

```

1 from transformers import AutoModelForSequenceClassification
2
3 checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
4 model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
5 outputs = model(**inputs)
6
7 print(outputs.logits.shape)
8 # 输出结果为:
9 # 现在，我们查看输入的形状，维度会低得多：
10 # 模型头将我们之前看到的高维向量作为输入，并输出包含两个值的向量（每个标签一个）
11 torch.Size([2, 2])
12
13
14 print(outputs)
15 # 输出结果为：
16 SequenceClassifierOutput(loss=None, logits=tensor([-1.5607,  1.6123],
17                               [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>),
18 hidden_states=None, attentions=None)

```

处理model的输出，转化为概率

我们的模型预测第一句为[-1.5607, 1.6123]，第二句为[4.1692, -3.3464]。这些不是概率，而是logits，即模型最后一层输出的原始的、非标准化的分数。要转换为概率，它们需要经过softmax(所有😊)。transformers模型都会输出logits，因为用于训练的损失函数通常会将最后一个激活函数(如SoftMax)与实际损失函数(如交叉熵)融合在一起)：

Python | 复制代码

```
1 import torch
2
3 predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
4
5 print(predictions)
6 # 输出结果为:
7 tensor([[4.0195e-02, 9.5980e-01],
8         [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)
```

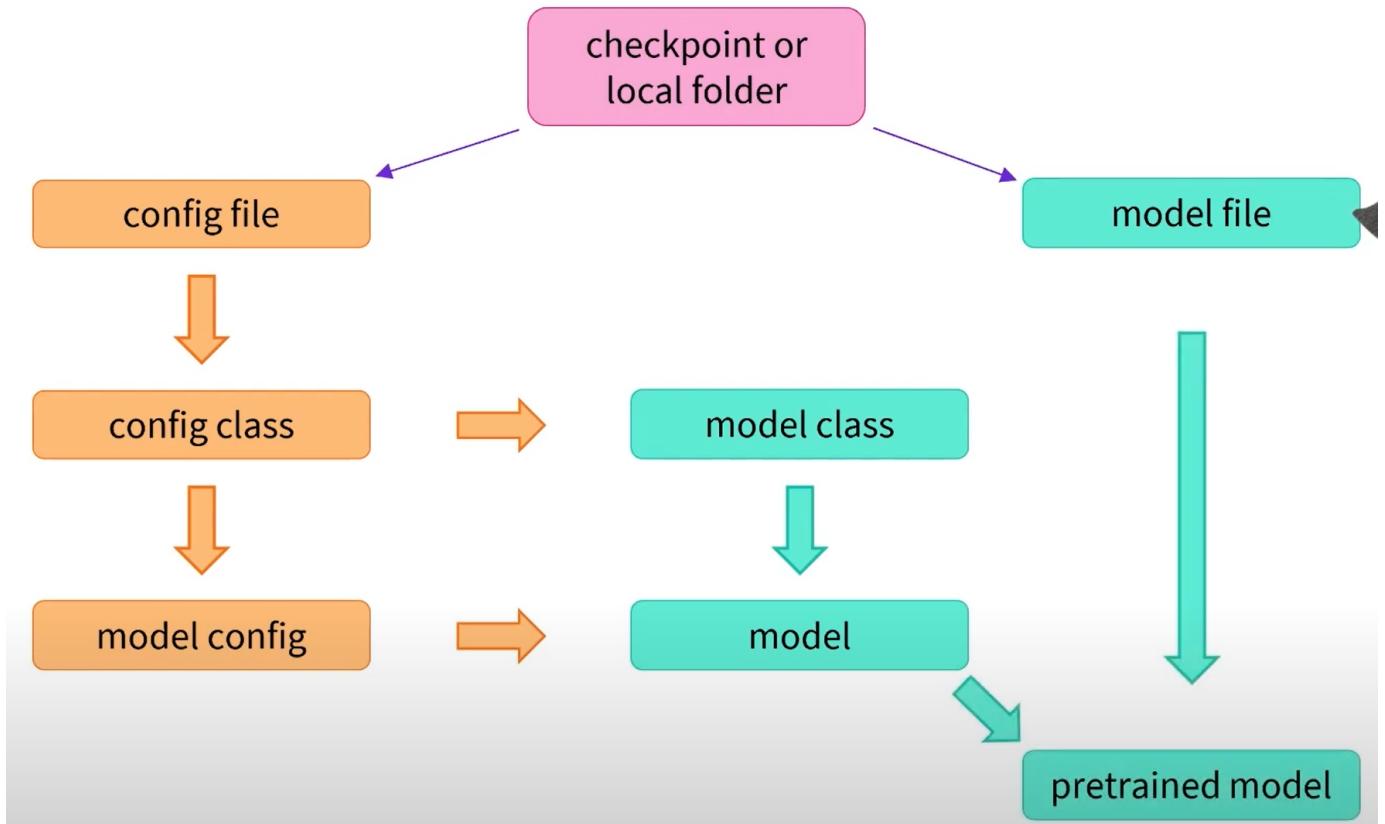
现在我们可以看到模型预测[0.0402, 0.9598]第一个句子和[0.9995, 0.0005]第二个句子。这些是可识别的概率分数。

要获得每个位置对应的标签，我们可以检查id2label模型配置的属性

Python | 复制代码

```
1 model.config.id2label
2 # 输出为:
3 {0: 'NEGATIVE', 1: 'POSITIVE'}
```

models



加载model

我们可以使用`from_pretrained`方法，来加载一个已经训练好的 Transformer 模型。

```

1 from transformers import BertModel
2
3 model = BertModel.from_pretrained("bert-base-cased")
  
```

保存model

保存模型就像加载模型一样简单——我们使用`save_pretrained`方法，它类似于`from_pretrained`方法：

```

1 model.save_pretrained("directory_on_my_computer")
  
```

该命令会将 `config.json` `pytorch_model.bin` 两个文件保存在硬盘。

- `config.json` 文件，构建model architecture所需的属性。此文件还包含一些元数据，例如上次保存 checkpoint时使用的 😊 Transformers 版本。文件内容如下：

[git@github.com:hiih/he/Hugging_Face_Course.git](https://github.com/hiih/he/Hugging_Face_Course.git)

```

1   {
2     "_name_or_path": "bert-base-cased",
3     "architectures": [
4       "BertModel"
5     ],
6     "attention_probs_dropout_prob": 0.1,
7     "gradient_checkpointing": false,
8     "hidden_act": "gelu",
9     "hidden_dropout_prob": 0.1,
10    "hidden_size": 768,
11    "initializer_range": 0.02,
12    "intermediate_size": 3072,
13    "layer_norm_eps": 1e-12,
14    "max_position_embeddings": 512,
15    "model_type": "bert",
16    "num_attention_heads": 12,
17    "num_hidden_layers": 12,
18    "pad_token_id": 0,
19    "position_embedding_type": "absolute",
20    "transformers_version": "4.8.1",
21    "type_vocab_size": 2,
22    "use_cache": true,
23    "vocab_size": 28996
24  }

```

- `pytorch_model.bin` 文件称为状态字典；它包含模型的所有权重。configuration是了解model architecture所必需的，而模型权重是模型的参数。

Tokenizers

参考链接

Tokenizers: 将文本转换为模型可以处理的数据。模型只能处理数字，因此Tokenizers需要将我们的文本输入转换为数字数据。tokenizer 主要分为如图所示三类：

Word-based

Character-based

Subword-based

word-based

git@github.com:hihihe/Hugging_Face_Course.git

Split on spaces				
Let's	do	tokenization!		

Split on punctuation				
Let	's	do	tokenization	!

采用word-based vocabulary, 英语有17W的单词。中文有2W个字, 超过37.5W个词。

character-based

L	e	t	'	s	d	o	t	o	k	e	n	i	z	a	t	i	o	n	!
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

word-based和character-based tokenization的缺点

Word-based tokenization

Very large vocabularies

Large quantity of out-of-vocabulary tokens

Loss of meaning across very similar words

Character-based tokenization

Very long sequences

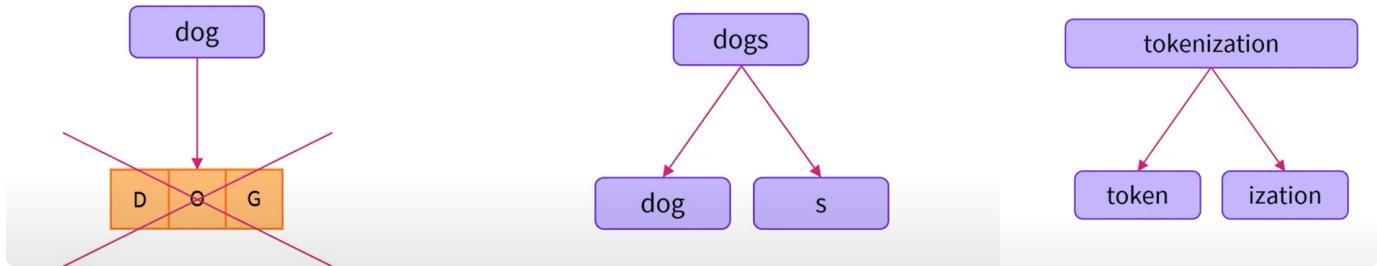
Less meaningful individual tokens

subword-based

Let's</w>	do</w>	token	ization</w>	!</w>
-----------	--------	-------	-------------	-------

原则:

- 经常使用的words不应该被切分成更小的subwords。
- 罕见的words应该被分解成有意义的subwords。



tokenizers小结(差不多就这样吧)

其他技术

- Byte-level BPE, as used in GPT-2
- WordPiece, as used in BERT
- SentencePiece or Unigram, as used in several multilingual models



tokenizer

```

▼ Python | 复制代码

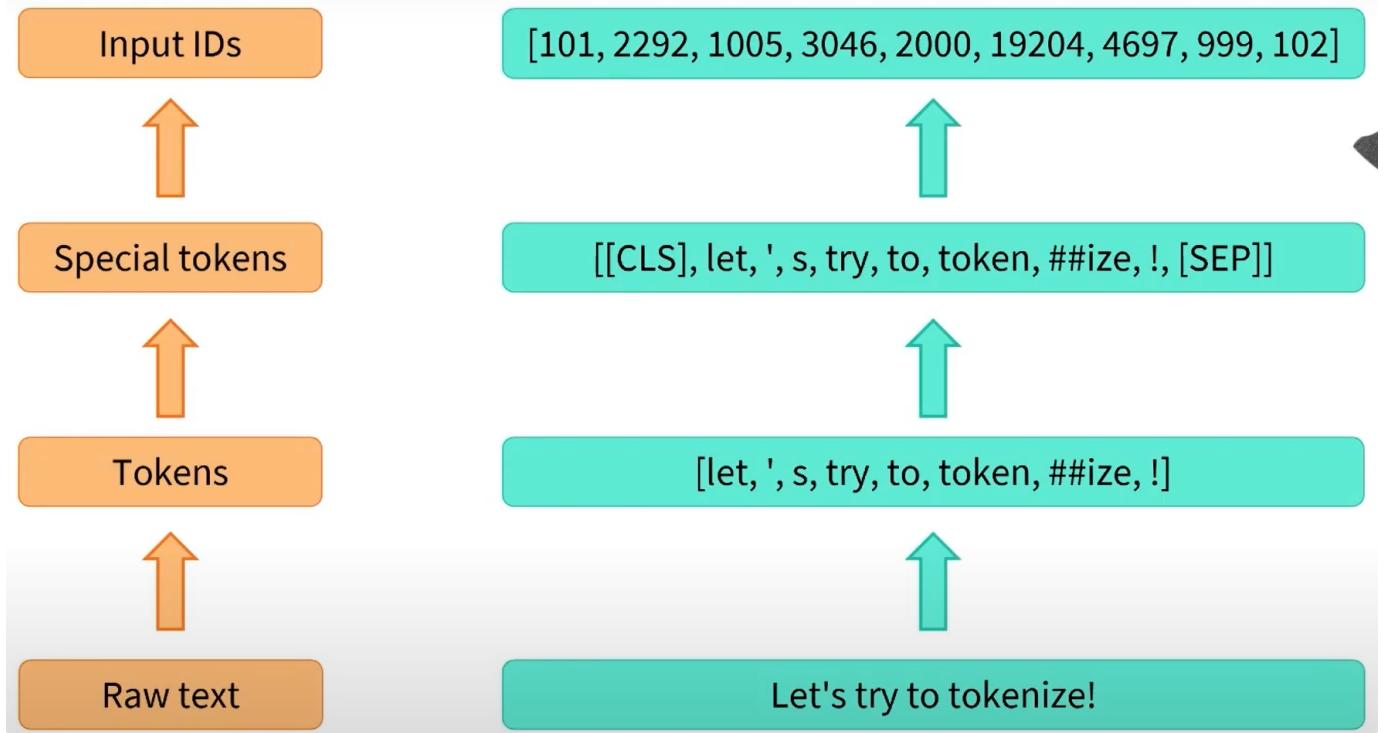
1 from transformers import AutoTokenizer
2
3 tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
4
5 tokenizer("Using a Transformer network is simple")
# 输出为:
6 {'input_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],
7  'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
8  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}

```

tokenizer将文本转换为数字称为编码。编码分两步完成：标记化，然后转换为输入 ID。

第一步是将文本拆分为单词（或单词的一部分、标点符号等），通常称为标记。**有多个规则可以管理该过程，这就是为什么我们需要使用模型名称来实例化分词器，以确保我们使用模型预训练时使用的相同规则。**

第二步是将这些标记转换为数字，这样我们就可以用它们构建一个张量并将它们提供给模型。为此，分词器有一个词汇表，这是我们在使用from_pretrained方法实例化它时下载的部分。同样，我们需要使用模型预训练时使用的相同词汇。



tokenizer方法实现了tokenization, 处理special tokens和转化为input ids三个过程。

我们可以使用 `tokenizer.tokenize(sequence)` 来实现tokenization这个过程, 输出是 list of strings, or tokens。使用 `tokenizer.convert_tokens_to_ids(tokens)` 来实现将tokens转换为ids。

注意: `tokenizer(sequence)`方法, 可能在开头添加[CLS]结尾添加[SEP]等特殊词。这是因为模型是用这些进行预训练的, 所以为了获得相同的推理结果, 我们还需要添加它们。**注意有些checkpoint不加特殊词, 或者加不同的词;** 模型也可以仅在开头或仅在结尾添加这些特殊词。在任何情况下, `tokenizer()`都会自动处理这些。。。而使用`tokenizer.tokenize(sequence)`则不会处理这些。

Python | 复制代码

```
1 from transformers import AutoTokenizer
2 tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
3
4 sequence = "Using a Transformer network is simple"
5 tokens = tokenizer.tokenize(sequence)
6
7 print(tokens)
8 # 输出为:
9 ['Using', 'a', 'Trans', '##former', 'network', 'is', 'simple']
10 # 这个分词器是一个子词分词器：它对词进行拆分，直到获得可以用其词汇表表示的标记。
11 # transformer，它被分成两个标记：trans和##former。
12
13 ids = tokenizer.convert_tokens_to_ids(tokens)
14 print(ids)
15 # 输出为:
16 [7993, 170, 13809, 23763, 2443, 1110, 3014]
17 # 将其转换为张量后，可以用作模型的输入。
```

Python | 复制代码

```
1 decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110,
2 3014])
3 print(decoded_string)
4 # 输出为:
5 Using a transformer network is simple
```

请注意，该decode方法不仅将索引转换回标记，还将属于相同单词的标记组合在一起以生成可读的句子。当我们使用预测新文本的模型（从提示生成的文本，或序列到序列问题（如翻译或摘要））时，这种行为将非常有用。

处理多个sequences

```

1 import torch
2 from transformers import AutoTokenizer,
3 AutoModelForSequenceClassification
4
5 checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
6 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
7 model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
8
9 sequence = "I've been waiting for a HuggingFace course my whole life."
10 tokens = tokenizer.tokenize(sequence)
11 ids = tokenizer.convert_tokens_to_ids(tokens)
12 input_ids = torch.tensor(ids)
13 # 这里无法执行，将会报错
14 model(input_ids)

```

 5 frames

```

/usr/local/lib/python3.7/dist-packages/transformers/models/distilbert/modeling_distilbert.py in forward(self, input_ids)
101     embeddings
102     """
--> 103     seq_length = input_ids.size(1)
104     position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device) # (max_seq_length)
105     position_ids = position_ids.unsqueeze(0).expand_as(input_ids) # (bs, max_seq_length)

```

IndexError: Dimension out of range (expected to be in range of [-1, 0], but got 1)

报错原因是向model发送了单个序列，而 😅 Transformers model默认需要多个句子即2维张量。

修改方法：

添加一个维度 `input_ids = torch.tensor([ids])`。

对于不同长度的序列，需要进行截断或者填充。可以使用`tokenizer.pad_token_id`找到padding token ID。

tokenizer处理完整示例

官方代码

tokenizer能将输出处理为特定框架的tensor— "pt" returns PyTorch tensors, "tf" returns TensorFlow tensors, and "np" returns NumPy arrays。

```
1 import torch
2 from transformers import AutoTokenizer,
3 AutoModelForSequenceClassification
4
5 checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
6 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
7 model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
8 sequences = [
9     "I've been waiting for a HuggingFace course my whole life.",
10    "So have I!"
11 ]
12 # 一定要有return_tensors参数, 后续才能直接输入给model。
13
14 # padding=True/"longest"是将每个sequence填充到该批输入的最大长度
15 # padding="max_length" 填充每个sequence到model的最大长度(512 for BERT or
16 DistilBERT)
17 # padding="max_length", max_length=8 Will pad the sequences up to the
18 specified max_length
19
20 # truncation=True 截断每个sequence到modle的最大长度(512 for BERT or
21 DistilBERT)
22 # max_length=8, truncation=True 截断每个sequence到指定的max_length长度
23
24 # padding=True, truncation=True, max_length=8 3个参数一起使用, 将所有
# sequence处理为长度8。
25 # padding=True, truncation=True 2个参数一起使用, 将所有sequence处理为该批最长
# sequence的长度或者模型允许的最大长度。
26 tokens = tokenizer(sequences, padding=True, truncation=True,
27 return_tensors="pt")
28 output = model(**tokens)
```

3. Fine-tuning a pretrained model

处理数据

官方代码

下面是我们如何在 PyTorch 中训练一个批次的序列分类器：

```
1 import torch
2 from transformers import AdamW, AutoTokenizer,
3 AutoModelForSequenceClassification
4
5 # Same as before
6 checkpoint = "bert-base-uncased"
7 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
8 model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
9 sequences = [
10     "I've been waiting for a HuggingFace course my whole life.",
11     "This course is amazing!",
12 ]
13 batch = tokenizer(sequences, padding=True, truncation=True,
14 return_tensors="pt")
15
16 # This is new
17 batch["labels"] = torch.tensor([1, 1])
18
19 optimizer = AdamW(model.parameters())
20 loss = model(**batch).loss
21 loss.backward()
22 optimizer.step()
```

当然，仅仅在两个句子上训练模型不会产生很好的结果。为了获得更好的结果，您需要准备更大的数据集。

在下文中在本节中，我们将使用MRPC(Microsoft Research Paraphrase Corpus)数据集作为示例，该数据集在William B.Dolan和Chris Brockett的一篇论文中介绍。该数据集由5801对句子组成，并有一个标签表明它们是否是意译(即，如果两个句子的意思是相同的)。我们选择了它，因为它是一个很小的数据集，因此很容易在上面进行训练。通过以下代码获取该数据集：

```
1 from datasets import load_dataset
2
3 raw_datasets = load_dataset("glue", "mrpc")
4 raw_datasets
# 输出结果为:
5 DatasetDict({
6     train: Dataset({
7         features: ['idx', 'label', 'sentence1', 'sentence2'],
8         num_rows: 3668
9     })
10    validation: Dataset({
11        features: ['idx', 'label', 'sentence1', 'sentence2'],
12        num_rows: 408
13    })
14    test: Dataset({
15        features: ['idx', 'label', 'sentence1', 'sentence2'],
16        num_rows: 1725
17    })
18 })
19 # 我们得到一个DatasetDict包含训练集、验证集和测试集的对象。
20 # 每一个都包含几列 (sentence1、sentence2、label和idx) 和可变数量的行
21
22
23
24 # 查看训练集中的一条数据
25 raw_train_dataset = raw_datasets["train"]
26 raw_train_dataset[0]
27 # 输出为:
28 {'idx': 0,
29  'label': 1,
30  'sentence1': 'Amrozi accused his brother , whom he called " the witness '
31  " , of deliberately distorting his evidence .',
32  'sentence2': 'Referring to him as only " the witness " , Amrozi accused
33  his brother of deliberately distorting his evidence .'}
34
35 # 查看训练集的features
36 raw_train_dataset.features
37 # 输出为:
38 {'idx': Value(dtype='int32', id=None),
39  'label': ClassLabel(num_classes=2, names=['not_equivalent',
40  'equivalent']), names_file=None, id=None),
41  'sentence1': Value(dtype='string', id=None),
42  'sentence2': Value(dtype='string', id=None)}
43 # label是ClassLabel类型，整数到标签名称的映射存储在names中。0对应
44 not_equivalent，1对应equivalent。
```

我们得到一个DatasetDict包含训练集、验证集和测试集的对象。每一个都包含几列 (sentence1、sentence2、label和idx) 和可变数量的行，它们是每个集合中元素的数量（因此，训练集中有 3668 对句子，验证集中有 408 对，还有 1725 在测试集中）。

使用tokenizer处理一对句子（模型需要一对输入，比如判断这对句子是否意思相同）

```
from transformers import AutoTokenizer

checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokenizer("My name is Sylvain.", "I work at Hugging Face.")
```

ID	101	2026	2171	2003	25353	22144	2378	1012	102	1045	2147	2012	17662	2227	1012	102
token	[CLS]	my	name	is	sy	##lva	##in	.	[SEP]	I	work	at	hugging	face	.	[SEP]
token type	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
attention	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

token_type_ids是告诉模型输出的哪一部分是第一句，哪一部分是第二句。

请注意，如果选择不同的checkpoint，则在标记化的输入中不一定有**token_type_ids**(例如，如果您使用DistilBERT模型，则不会返回它们)。只有当模型知道如何处理它们时，它们才会被return，因为它在预训期间看到了它们。在这里，Bert预先接受了token type IDs的训练，在前文谈到的 masked language modeling objective目标之上，Bert还有一个额外的目标，称为下一句预测。这项任务的目标是对句子对之间的关系进行建模。对于下一句预测，模型被提供句子对(带有随机掩蔽的标记)，并被要求预测**第二句是否在第一句之后**。

通常，我们不需要担心标记化输入中是否有 **token_type_ids**: 只要您对标记器和模型使用相同的检查点，一切都会自动处理。

使用tokenizer处理多对句子

```

from transformers import AutoTokenizer

checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokenizer(
    ["My name is Sylvain.", "Going to the cinema."],
    ["I work at Hugging Face.", "This movie is great."],
    padding=True
)

```

注意：送入tokenizer的两个list的对应关系。如上图的代码，'my name is sylvain'和'i work at hugging face'是一对。输出解释看下面两个图。

```

{
  'input_ids': [
    [101, 2026, 2171, 2003, 25353, 22144, 2378, 1012, 102, 1045, 2147, 2012, 17662, 2227, 1012, 102],
    [101, 2183, 2000, 1996, 5988, 1012, 102, 2023, 3185, 2003, 2307, 1012, 102, 0, 0, 0]
  ],
  'token_type_ids': [
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0]
  ],
  'attention_mask': [
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
  ]
}

```



ID	101	2026	2171	2003	25353	22144	2378	1012	102	1045	2147	2012	17662	
token	[CLS]	my	name	is	sy	#lva	#in	.	[SEP]	I	work	at	hugging	
token type	0	0	0	0	0	0	0	0	0	1	1	1	1	1
attention	1	1	1	1	1	1	1	1	1	1	1	1	1	1
ID	101	2183	2000	1996	5988	1012	102	2023	3185	2003	2307	1012	102	0
token	[CLS]	going	to	the	cinema	.	[SEP]	this	movie	is	great	.	[SEP]	[PAD]
token type	0	0	0	0	0	0	0	1	1	1	1	1	1	0
attention	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Python | 复制代码

```
1 from datasets import load_dataset
2
3 raw_datasets = load_dataset("glue", "mrpc")
4
5 tokenized_dataset = tokenizer(
6     raw_datasets["train"]["sentence1"],
7     raw_datasets["train"]["sentence2"],
8     padding=True,
9     truncation=True,
10    return_tensors = 'pt'
11 )
```

上面的示例代码在内存使用上有缺点，为此我们使用`Dataset.map()`方法，该方法提供了更大的灵活性。如果我们要完成更多的预处理而不仅仅是tokenization。该map方法通过在数据集的每个元素上应用一个函数来工作，所以让我们定义一个函数tokenizes our inputs：

Python | 复制代码

```
1 from datasets import load_dataset
2
3 raw_datasets = load_dataset("glue", "mrpc")
4
5 # 注意：这里不需要使用padding，返回的也不是tensors。
6 def tokenize_function(example):
7     return tokenizer(example["sentence1"], example["sentence2"],
8                      truncation=True)
9
10 # batched=True在调用中使用，map因此该函数一次应用于数据集的多个元素（默认是1000），而不是分别应用于每个元素。这允许更快的预处理。
11 tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
```

💡 Datasets library 应用这种处理的方式是向数据集添加新字段，预处理函数返回的字典中的每个键对应一个字段。处理后的raw_datasets内容如下：

```
1 DatasetDict({
2     train: Dataset({
3         features: ['attention_mask', 'idx', 'input_ids', 'label',
4         'sentence1', 'sentence2', 'token_type_ids'],
5         num_rows: 3668
6     })
7     validation: Dataset({
8         features: ['attention_mask', 'idx', 'input_ids', 'label',
9         'sentence1', 'sentence2', 'token_type_ids'],
10        num_rows: 408
11    })
12    test: Dataset({
13        features: ['attention_mask', 'idx', 'input_ids', 'label',
14        'sentence1', 'sentence2', 'token_type_ids'],
15        num_rows: 1725
16    })
17 })
```

Dataset.map通过传递num_proc参数来应用预处理函数时，您甚至可以使用多线程。我们在这里没有这样做，因为 😊 Tokenizers 库已经使用多个线程来更快地处理我们的样本，但是如果您没有使用由该库支持的tokenizer，可以通过num_proc参数加快预处理。

数据预处理完整代码

```
1 from datasets import load_dataset
2
3 raw_datasets = load_dataset("glue", "mrpc")
4
5 # 注意：这里不需要使用padding，返回的也不是tensors。
6 def tokenize_function(example):
7     return tokenizer(example["sentence1"], example["sentence2"],
8 truncation=True)
9
10 # batched=True在调用中使用，map因此该函数一次应用于数据集的多个元素(默认是1000)，而
11 # 不是分别应用于每个元素。这允许更快的预处理。
12 tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
13
14 # 通过remove_columns删除与模型不期望的值对应的列
15 tokenized_datasets = tokenized_datasets.remove_columns(
16     ["sentence1", "sentence2", "idx"]
17 )
18
19 # 通过rename_column重命名一些列
20 tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
21
22 # 设置数据集的格式，使其返回 PyTorch 张量而不是列表。
23 tokenized_datasets.set_format("torch")
24
25 # 检查是否只有我们的模型可以接受的列：
26 tokenized_datasets["train"].column_names
27 # 输出结果为：
28 [ 'attention_mask', 'input_ids', 'labels', 'token_type_ids' ]
```

在 PyTorch 中，负责将一批样本放在一起的函数称为collate_fn函数。这是您在构建DataLoader时可以传递的参数，默认值是一个函数，它只会将您的样本转换为 PyTorch 张量并连接它们（如果您的元素是列表、元组或字典，则递归）。在我们的情况下这是不可能的，因为我们拥有的输入不会全部具有相同的大小。**我们特意推迟了填充，只在每批次必要时应用它，避免过长的输入和大量的填充。这将大大加快训练速度**，但请注意，如果您在 TPU 上进行训练，它可能会导致问题——TPU 更喜欢固定形状。

```
from datasets import load_dataset
from transformers import AutoTokenizer

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

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

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
tokenized_datasets = tokenized_datasets.remove_columns(["idx", "sentence1", "sentence2"])
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
tokenized_datasets = tokenized_datasets.with_format("torch")
```

为了能动态填充，我们在tokenize_function函数中没有做padding处理，在之后的data_collator函数做了padding。

```
1 from transformers import DataCollatorWithPadding
2
3 data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

```
from torch.utils.data import DataLoader
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer)
train_dataloader = DataLoader(
    tokenized_datasets["train"], batch_size=16, shuffle=True, collate_fn=data_collator
)

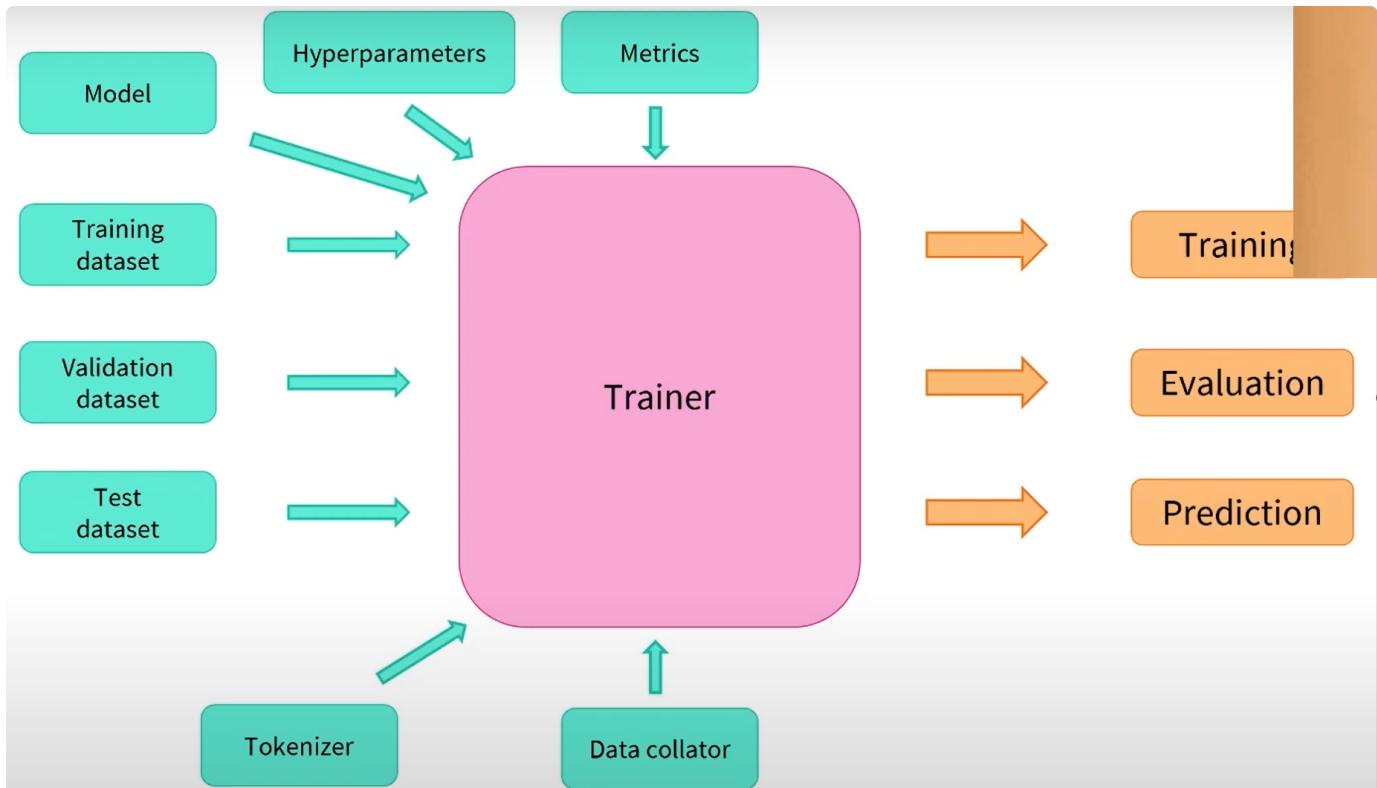
for step, batch in enumerate(train_dataloader):
    print(batch["input_ids"].shape)
    if step > 5:
        break
```

```
torch.Size([16, 77])
torch.Size([16, 80])
torch.Size([16, 80])
torch.Size([16, 90])
torch.Size([16, 73])
```

Fine-tuning a model with the Trainer API

[huggingface transformers使用指南1](#)

[huggingface transformers使用指南2](#)



```
1 from datasets import load_dataset
2 from transformers import AutoTokenizer, DataCollatorWithPadding
3 from transformers import TrainingArguments
4 from transformers import AutoModelForSequenceClassification
5 from transformers import Trainer
6
7
8 raw_datasets = load_dataset("glue", "mrpc")
9 checkpoint = "bert-base-uncased"
10 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
11
12 def tokenize_function(example):
13     return tokenizer(example["sentence1"], example["sentence2"],
14                      truncation=True)
15
16 tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
17 data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
18
19 # 在定义训练器之前，第一步是定义一个TrainingArguments类
20 # 它将包含训练器用于培训和评估的所有超参数。
21 # 必须提供的唯一参数是保存训练模型的目录。
22 # TrainingArguments参数指定了训练的设置：输出目录、总的epochs、
23 # 训练的batch_size、预测的batch_size、warmup的step数、weight_decay和log目录。
24 training_args = TrainingArguments("test-trainer")
25
26 model = AutoModelForSequenceClassification.from_pretrained(checkpoint,
27 num_labels=2)
28
29 trainer = Trainer(
30     model,
31     training_args,
32     train_dataset=tokenized_datasets["train"],
33     eval_dataset=tokenized_datasets["validation"],
34     data_collator=data_collator,
35     tokenizer=tokenizer,
36 )
37
38 trainer.train()
```

上述代码每500步报告一次训练损失，但是不会报告性能如何。这是因为：

1. 我们没有通过将`evaluation_strategy`设置为"`steps`"(评估每个评估步骤)或"`epoch`"(在每个时期结束时评估)来告诉`Trainer`在训练期间进行评估。

2. 我们没有为Trainer提供**compute_metrics**函数来在评估期间计算指标(否则评估只会打印loss,这不是一个非常直观的数字)。

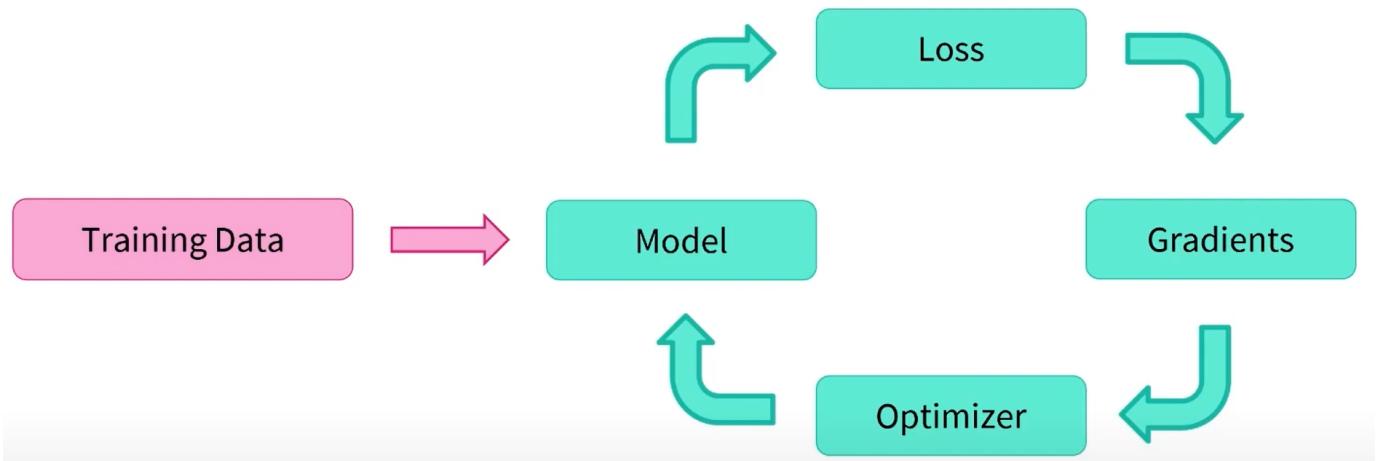
代码进行如下修改:

```
1 def compute_metrics(eval_preds):
2     metric = load_metric("glue", "mrpc")
3     logits, labels = eval_preds
4     predictions = np.argmax(logits, axis=-1)
5     return metric.compute(predictions=predictions, references=labels)
6
7 training_args = TrainingArguments("test-trainer",
8 evaluation_strategy="epoch")
9 model = AutoModelForSequenceClassification.from_pretrained(checkpoint,
10 num_labels=2)
11
12 trainer = Trainer(
13     model,
14     training_args,
15     train_dataset=tokenized_datasets["train"],
16     eval_dataset=tokenized_datasets["validation"],
17     data_collator=data_collator,
18     tokenizer=tokenizer,
19     compute_metrics=compute_metrics
20 )
```

使用训练的模型预测:

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

使用pytorch训练



[官方代码](#)

```
1 from datasets import load_dataset
2 from transformers import AutoTokenizer, DataCollatorWithPadding
3 from torch.utils.data import DataLoader
4 from transformers import AutoModelForSequenceClassification
5 from transformers import AdamW
6 from transformers import get_scheduler
7 import torch
8 from tqdm.auto import tqdm
9 from datasets import load_metric
10
11 device = torch.device("cuda") if torch.cuda.is_available() else
12 torch.device("cpu")
13
14 raw_datasets = load_dataset("glue", "mrpc")
15 checkpoint = "bert-base-uncased"
16 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
17
18 def tokenize_function(example):
19     return tokenizer(example["sentence1"], example["sentence2"],
20 truncation=True)
21
22 tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
23 data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
24
25 tokenized_datasets = tokenized_datasets.remove_columns(
26     ["sentence1", "sentence2", "idx"])
27 tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
28 tokenized_datasets.set_format("torch")
29 tokenized_datasets["train"].column_names
30
31 train_dataloader = DataLoader(
32     tokenized_datasets["train"], shuffle=True, batch_size=8,
33 collate_fn=data_collator
34 )
35 eval_dataloader = DataLoader(
36     tokenized_datasets["validation"], batch_size=8,
37 collate_fn=data_collator
38 )
39
40 model = AutoModelForSequenceClassification.from_pretrained(checkpoint,
41 num_labels=2)
42 optimizer = AdamW(model.parameters(), lr=5e-5)
```

```

41 num_epochs = 3
42 num_training_steps = num_epochs * len(train_dataloader)
43 lr_scheduler = get_scheduler(
44     "linear",
45     optimizer=optimizer,
46     num_warmup_steps=0,
47     num_training_steps=num_training_steps
48 )
49
50
51 model.to(device)
52
53 progress_bar = tqdm(range(num_training_steps))
54
55 model.train()
56 for epoch in range(num_epochs):
57     for batch in train_dataloader:
58         batch = {k: v.to(device) for k, v in batch.items()}
59         outputs = model(**batch)
60         loss = outputs.loss
61         loss.backward()
62
63         optimizer.step()
64         lr_scheduler.step()
65         optimizer.zero_grad()
66         progress_bar.update(1)
67
68
69 metric = load_metric("glue", "mrpc")
70 model.eval()
71 for batch in eval_dataloader:
72     batch = {k: v.to(device) for k, v in batch.items()}
73     with torch.no_grad():
74         outputs = model(**batch)
75
76         logits = outputs.logits
77         predictions = torch.argmax(logits, dim=-1)
78         metric.add_batch(predictions=predictions, references=batch["labels"])
79
80 metric.compute()
81
82 model.save_pretrained('./my_model/')

```

使用😊 Accelerate 加速

```

+ from accelerate import Accelerator

+ accelerator = Accelerator()

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

    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)

```

```

from datasets import load_metric

metric= load_metric("glue", "mrpc")
model.eval()

+ eval_dataloader = accelerator.prepare(eval_dataloader)
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.add_batch(
+         predictions=accelerator.gather(predictions), references=accelerator.gather(batch["labels"])
+     )

metric.compute()

```

[accelerate官方地址](#)

git@github.com:hihihe/Hugging_Face_Course.git

```
1 from datasets import load_dataset
2 from transformers import AutoTokenizer, DataCollatorWithPadding
3 from torch.utils.data import DataLoader
4 from transformers import AutoModelForSequenceClassification
5 from transformers import AdamW
6 from transformers import get_scheduler
7 import torch
8 from tqdm.auto import tqdm
9 from datasets import load_metric
10 from accelerate import Accelerator
11 import os
12 from multiprocessing import cpu_count
13
14 cpu_num = cpu_count() # 自动获取最大核心数目
15 os.environ['OMP_NUM_THREADS'] = str(cpu_num)
16 os.environ['OPENBLAS_NUM_THREADS'] = str(cpu_num)
17 os.environ['MKL_NUM_THREADS'] = str(cpu_num)
18 os.environ['VECLIB_MAXIMUM_THREADS'] = str(cpu_num)
19 os.environ['NUMEXPR_NUM_THREADS'] = str(cpu_num)
20 torch.set_num_threads(cpu_num)
21
22 raw_datasets = load_dataset("glue", "mrpc")
23 checkpoint = "bert-base-uncased"
24 tokenizer = AutoTokenizer.from_pretrained(checkpoint)
25
26 def tokenize_function(example):
27     return tokenizer(example["sentence1"], example["sentence2"],
28 truncation=True)
29
30 tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
31 data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
32
33 tokenized_datasets = tokenized_datasets.remove_columns([
34     "sentence1", "sentence2", "idx"])
35 tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
36 tokenized_datasets.set_format("torch")
37 tokenized_datasets["train"].column_names
38
39 train_dataloader = DataLoader(
40     tokenized_datasets["train"], shuffle=True, batch_size=8,
41     collate_fn=data_collator)
42 eval_dataloader = DataLoader(
```

```
43     tokenized_datasets["validation"], batch_size=8,
44     collate_fn=data_collator
45 )
46 accelerator = Accelerator()
47 model = AutoModelForSequenceClassification.from_pretrained(checkpoint,
48     num_labels=2)
49 optimizer = AdamW(model.parameters(), lr=5e-5)
50
51 num_epochs = 3
52 num_training_steps = num_epochs * len(train_dataloader)
53 lr_scheduler = get_scheduler(
54     "linear",
55     optimizer=optimizer,
56     num_warmup_steps=0,
57     num_training_steps=num_training_steps
58 )
59
60
61 progress_bar = tqdm(range(num_training_steps))
62
63 model.train()
64 for epoch in range(num_epochs):
65     for batch in train_dataloader:
66         outputs = model(**batch)
67         loss = outputs.loss
68         accelerator.backward(loss)
69
70         optimizer.step()
71         lr_scheduler.step()
72         optimizer.zero_grad()
73         progress_bar.update(1)
74
75
76 model.save_pretrained('./my_model/')
77
```

启动方式：

```
1 # 假设代码为train.py文件
2
3 # 运行以下命令，它将提示您回答几个问题，并将答案转储到此命令使用的配置文件中
4 accelerate config
5
6 # 启动train.py文件
7 accelerate launch train.py
8
```

4. Sharing models and tokenizers

略