

本章简介

正如你在 [Chapter 1](#) 中看到的那样，Transformers模型通常非常大。对于数以百万计到数千万计数十亿的参数，训练和部署这些模型是一项复杂的任务。此外，由于几乎每天都在发布新模型，而且每种模型都有自己的实现，因此尝试它们绝非易事。

创建 🤖 Transformers库就是为了解决这个问题。它的目标是提供一个API，通过它可以加载、训练和保存任何Transformer模型。这个库的主要特点是：

- **易于使用**：下载、加载和使用最先进的NLP模型进行推理只需两行代码即可完成。
- **灵活**：所有型号的核心都是简单的PyTorch **nn.Module** 或者 TensorFlow **tf.keras.Model**，可以像它们各自的机器学习（ML）框架中的任何其他模型一样进行处理。
- **简单**：当前位置整个库几乎没有任何摘要。“都在一个文件中”是一个核心概念：模型的正向传递完全定义在一个文件中，因此代码本身是可以理解的，并且是可以破解的。

最后一个特性使 🤖 Transformers与其他ML库截然不同。这些模型不是基于通过文件共享的模块构建的；相反，每一个模型都有自己的菜单。除了使模型更加容易接受和更容易理解，这还允许你轻松地在一个模型上实验，而且不影响其他模型。

本章将从一个端到端的示例开始，在该示例中，我们一起使用模型和tokenizer分词器来复制 [Chapter 1](#) 中引入的函数pipeline()。接下来，我们将讨论模型API：我们将深入研究模型和配置类，并向您展示如何加载模型以及如何将数值输入处理为输出预测。

然后我们来看看标记器API，它是pipeline()函数的另一个主要组件。它是作用分词器负责第一个和最后一个处理步骤，处理从文本到神经网络数字输入的转换，以及在需要时转换回文本。最后，我们将向您展示如何处理在一个准备好的批处理中通过一个模型发送多个句子的问题，然后详细介绍pipeline()函数。

△ 为了从模型集线器和 🤖 Transformers的所有可用功能中获益，我们建议 [creating an account](#).
<FrameworkSwitchCourse {fw} />

管道的内部

这是第一部分，根据您使用PyTorch或者TensorFlow，内容略有不同。点击标题上方的平台，选择您喜欢的平台！

让我们从一个完整的示例开始，看看在[Chapter 1](#)中执行以下代码时在幕后发生了什么

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
classifier([
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
])
```

获得:

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437},
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

正如我们在[Chapter 1](#)中看到的，此管道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：

让我们快速浏览一下这些内容。

使用分词器进行预处理

与其他神经网络一样，Transformer模型无法直接处理原始文本，因此我们管道的第一步是将文本输入转换为模型能够理解的数字。为此，我们使用`tokenizer`(标记器)，负责：

- 将输入拆分为单词、子单词或符号（如标点符号），称为标记(token)
- 将每个标记(token)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](#)中下载这些信息。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相关联的数据，并对其进行缓存（因此只有在您第一次运行下面的代码时才会下载）。

因为 `sentiment-analysis`（情绪分析）管道的默认检查点是 `distilbert-base-uncased-finetuned-sst-2-english`（你可以看到它的模型卡[here](#)），我们运行以下程序：

```
from transformers import AutoTokenizer
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

一旦我们有了标记器，我们就可以直接将我们的句子传递给它，然后我们就会得到一本字典，它可以提供给我们的模型！剩下要做的唯一一件事就是将输入ID列表转换为张量。

您可以使用 🤖 Transformers，而不必担心哪个ML框架被用作后端；它可能是PyTorch或TensorFlow，或Flax。但是，Transformers型号只接受张量作为输入。如果这是你第一次听说张量，你可以把它们想象成NumPy数组。NumPy数组可以是标量（0D）、向量（1D）、矩阵（2D）或具有更多维度。它实际上是张量；其他ML框架的张量行为类似，通常与NumPy数组一样易于实例化。要指定要返回的张量类型（PyTorch、TensorFlow或plain NumPy），我们使用 `return_tensors` 参数：

```
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True,
return_tensors="pt")
print(inputs)
```

```
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True,
return_tensors="tf")
print(inputs)
```

现在不要担心填充和截断；我们稍后会解释这些。这里要记住的主要事情是，您可以传递一个句子或一组句子，还可以指定要返回的张量类型（如果没有传递类型，您将得到一组列表）。

以下是PyTorch张量的结果：

```
{
  'input_ids': tensor([
    [ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172,
    2607, 2026, 2878, 2166, 1012, 102],
    [ 101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0, 0,
    0, 0, 0, 0, 0, 0]
  ]),
  'attention_mask': tensor([
    [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, 0, 0, 0, 0, 0, 0]
  ])
}
```

以下是TensorFlow张量的结果：

```
{
  'input_ids': <tf.Tensor: shape=(2, 16), dtype=int32, numpy=
    array([
      [ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662,
    12172, 2607, 2026, 2878, 2166, 1012, 102],
      [ 101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0,
    0, 0, 0, 0, 0, 0, 0]
    ], dtype=int32)>,
  'attention_mask': <tf.Tensor: shape=(2, 16), dtype=int32, numpy=
    array([
      [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, 0, 0, 0, 0, 0, 0]
    ], dtype=int32)>
}
```

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（每个句子一行），它们是每个句子中标记的唯一标记（token）。我们将在本章后面解释什么是 `attention_mask`。

浏览模型

我们可以像使用标记器一样下载预训练模型。🤖 Transformers 提供了一个 `AutoModel` 类，该类还具有 `from_pretrained()` 方法：

```
from transformers import AutoModel
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModel.from_pretrained(checkpoint)
```

我们可以像使用标记器一样下载预训练模型。🤖 Transformers 提供了一个 `TFAutoModel` 类，该类还具有 `from_pretrained()` 方法：

```
from transformers import TFAutoModel
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = TFAutoModel.from_pretrained(checkpoint)
```

在这个代码片段中，我们下载了之前在管道中使用的相同检查点（它实际上应该已经被缓存），并用它实例化了一个模型。

这个架构只包含基本转换器模块：给定一些输入，它输出我们将调用的内容 *隐藏状态 (hidden states)*，亦称 *特征 (features)*。对于每个模型输入，我们将检索一个高维向量，表示 **Transformer 模型对该输入的上下文理解**。

如果这不合理，不要担心。我们以后再解释。

虽然这些隐藏状态本身可能很有用，但它们通常是模型另一部分（称为 *头部(head)*）的输入。在 [Chapter 1](#) 中，可以使用相同的体系结构执行不同的任务，但这些任务中的每个任务都有一个与之关联的不同头。

高维向量？

Transformers 模块的矢量输出通常较大。它通常有三个维度：

- **Batch size**: 一次处理的序列数（在我们的示例中为2）。
 - **Sequence length**: 序列的数值表示的长度（在我们的示例中为16）。
 - **Hidden size**: 每个模型输入的向量维度。
由于最后一个值，它被称为“高维”。隐藏的大小可能非常大（768通常用于较小的型号，而在较大的型号中，这可能达到3072或更大）。
- 如果我们将预处理的输入输入到模型中，我们可以看到这一点：

```
outputs = model(**inputs)
print(outputs.last_hidden_state.shape)
```

```
torch.Size([2, 16, 768])
```

```
outputs = model(inputs)
print(outputs.last_hidden_state.shape)
```

```
(2, 16, 768)
```

注意🧠 Transformers 模型的输出与 `namedtuple` 或字典相似。您可以通过属性（就像我们所做的那样）或键（输出 `["last_hidden_state"]`）访问元素，甚至可以通过索引访问元素，前提是您确切知道要查找的内容在哪里（`outputs[0]`）。

模型头：数字的意义

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

Transformers 模型的输出直接发送到模型头进行处理。

在此图中，模型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 (token) 的向量。后续层使用注意机制操纵这些向量，以生成句子的最终表示。

🧠 Transformers 中有许多不同的体系结构，每种体系结构都是围绕处理特定任务而设计的。以下是一个非详尽的列表：

- `*Model` (retrieve the hidden states)
- `*ForCausalLM`
- `*ForMaskedLM`
- `*ForMultipleChoice`
- `*ForQuestionAnswering`
- `*ForSequenceClassification`

- `*ForTokenClassification`

- 以及其他 🤖

对于我们的示例，我们需要一个带有序列分类头的模型（能够将句子分类为肯定或否定）。因此，我们实际上不会使用 `AutoModel` 类，而是使用 `AutoModelForSequenceClassification`：

```
from transformers import AutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(**inputs)
```

For our example, we will need a model with a sequence classification head (to be able to classify the sentences as positive or negative). So, we won't actually use the `TFAutoModel` class, but `TFAutoModelForSequenceClassification`:

```
from transformers import TFAutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(inputs)
```

现在，如果我们观察输入的形状，维度将低得多：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）：

```
print(outputs.logits.shape)
```

```
torch.Size([2, 2])
```

```
(2, 2)
```

因为我们只有两个句子和两个标签，所以我们从模型中得到的结果是 2×2 的形状。

对输出进行后处理

我们从模型中得到的输出值本身并不一定有意义。我们来看看，

```
print(outputs.logits)
```

```
tensor([[ -1.5607,  1.6123],
        [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>)
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
  array([[ -1.5606991,  1.6122842],
        [ 4.169231 , -3.3464472]], dtype=float32)>
```

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非标准化分数。要转换为概率，它们需要经过 [SoftMax](#) 层（所有 🤖 Transformers 模型输出 logits，因为用于训练的损耗函数通常会将最后的激活函数（如 SoftMax）与实际损耗函数（如交叉熵）融合）：

```
import torch
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
```

```
import tensorflow as tf
predictions = tf.math.softmax(outputs.logits, axis=-1)
print(predictions)
```

```
tensor([[4.0195e-02, 9.5980e-01],
        [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)
```

```
tf.Tensor(
  [[4.01951671e-02 9.59804833e-01]
   [9.9945587e-01 5.4418424e-04]], shape=(2, 2), dtype=float32)
```

现在我们可以看到，模型预测第一句为 [0.0402, 0.9598]，第二句为 [0.9995, 0.0005]。这些是可识别的概率分数。

为了获得每个位置对应的标签，我们可以检查模型配置的 `id2label` 属性（下一节将对此进行详细介绍）：


```
model.config.id2label
```

```
{0: 'NEGATIVE', 1: 'POSITIVE'}
```

现在我们可以得出结论，该模型预测了以下几点：

- 第一句：否定：0.0402，肯定：0.9598
- 第二句：否定：0.9995，肯定：0.0005

我们已经成功地复制了管道的三个步骤：使用标记化器进行预处理、通过模型传递输入以及后处理！现在，让我们花一些时间深入了解这些步骤中的每一步。

 **试试看！** 选择两个（或更多）你自己的文本并在管道中运行它们。然后自己复制在这里看到的步骤，并检查是否获得相同的结果！

模型

在本节中，我们将更详细地了解如何创建和使用模型。我们将使用 `AutoModel` 类，当您希望从检查点实例化任何模型时，这非常方便。

这个 `AutoModel` 类及其所有相关项实际上是对库中各种可用模型的简单包装。它是一个聪明的包装器，因为它可以自动猜测检查点的适当模型体系结构，然后用该体系结构实例化模型。

在本节中，我们将更详细地了解如何创建和使用模型。我们将使用 `AutoModel` 类，当您希望从检查点实例化任何模型时，这非常方便。

这个 `AutoModel` 类及其所有相关项实际上是对库中各种可用模型的简单包装。它是一个聪明的包装器，因为它可以自动猜测检查点的适当模型体系结构，然后用该体系结构实例化模型。

但是，如果您知道要使用的模型类型，则可以使用直接定义其体系结构的类。让我们看看这是如何与 BERT 模型一起工作的。

创建转换器

初始化 BERT 模型需要做的第一件事是加载配置对象：

```
from transformers import BertConfig, BertModel
# Building the config
config = BertConfig()
# Building the model from the config
model = BertModel(config)
```

```
from transformers import BertConfig, TFBertModel
# Building the config
config = BertConfig()
# Building the model from the config
model = TFBertModel(config)
```

配置包含许多用于构建模型的属性：

```
print(config)
```

```
BertConfig {
  [...]
  "hidden_size": 768,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  [...]
}
```

虽然您还没有看到所有这些属性都做了什么，但您应该认识到其中的一些属性：hidden_size属性定义了hidden_状态向量的大小，num_hidden_layers定义了Transformer模型的层数。

不同的加载方式

从默认配置创建模型会使用随机值对其进行初始化：

```
from transformers import BertConfig, BertModel
config = BertConfig()
model = BertModel(config)
# Model is randomly initialized!
```

```
from transformers import BertConfig, TFBertModel
config = BertConfig()
model = TFBertModel(config)
# Model is randomly initialized!
```

该模型可以在这种状态下使用，但会输出胡言乱语；首先需要对其进行训练。我们可以根据手头的任务从头开始训练模型，但正如您在

[Chapter 1](#)

，这将需要很长的时间和大量的数据，并将产生不可忽视的环境影响。为了避免不必要的重复工作，必须能够共享和重用已经训练过的模型。

加载已经训练过的Transformers模型很简单-我们可以使用from_pretrained()方法：


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

正如您之前看到的，我们可以用等效的AutoModel类替换Bert模型。从现在开始，我们将这样做，因为这会产生检查点不可知的代码；如果您的代码适用于一个检查点，那么它应该与另一个检查点无缝地工作。即使体系结构不同，这也适用，只要检查点是针对类似任务（例如，情绪分析任务）训练的。

```
from transformers import TFBertModel
model = TFBertModel.from_pretrained("bert-base-cased")
```

正如您之前看到的，我们可以用等效的AutoModel类替换Bert模型。从现在开始，我们将这样做，因为这会产生检查点不可知的代码；如果您的代码适用于一个检查点，那么它应该与另一个检查点无缝地工作。即使体系结构不同，这也适用，只要检查点是针对类似任务（例如，情绪分析任务）训练的。

在上面的代码示例中，我们没有使用BertConfig

，而是通过Bert base cased标识符加载了一个预训练模型。这是一个模型检查点，由BERT的作者自己训练；您可以在

[model card](#)中找到更多细节。

该模型现在使用检查点的所有权重进行初始化。它可以直接用于对训练过的任务进行推理，也可以对新任务进行微调。通过预先训练权重而不是从头开始的训练，我们可以很快取得好的效果。

权重已下载并缓存在缓存文件夹中（因此将来对from_pretrained()方法的调用将不会重新下载它们）默认为

~/.cache/huggingface/transformers

.您可以通过设置

HF_HOME

环境变量来自定义缓存文件夹。

用于加载模型的标识符可以是模型中心Hub上任何模型的标识符，只要它与BERT体系结构兼容。可以找到可用的BERT检查点的完整列表

[here](#)

.

保存模型

保存模型和加载模型一样简单--我们使用

save_pretrained()

方法，类似于

from_pretrained()

方法：

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

这会将两个文件保存到磁盘：

```
ls directory_on_my_computer
config.json  pytorch_model.bin
```

```
ls directory_on_my_computer
config.json  tf_model.h5
```


如果你看一下

config.json

文件，您将识别构建模型体系结构所需的属性。该文件还包含一些元数据，例如检查点的来源以及上次保存检查点时使用的 🤖 Transformers 版本。

这个 `pytorch_model.bin` 文件就是众所周知的 *state dictionary*；它包含模型的所有权重。这两个文件齐头并进；配置是了解模型体系结构所必需的，而模型权重是模型的参数。

这个 `pytorch_model.bin` 文件就是众所周知的 *state dictionary*；它包含模型的所有权重。这两个文件齐头并进；配置是了解模型体系结构所必需的，而模型权重是模型的参数。

使用Transformers模型进行推理

既然您知道了如何加载和保存模型，那么让我们尝试使用它进行一些预测。Transformer 模型只能处理数字——分词器生成的数字。但在我们讨论标记化器之前，让我们先探讨模型接受哪些输入。

标记化器可以将输入转换为适当的框架张量，但为了帮助您了解发生了什么，我们将快速了解在将输入发送到模型之前必须做什么。

假设我们有几个序列：

```
sequences = ["Hello!", "Cool.", "Nice!"]
```

分词器将这些转换为词汇表索引，通常称为

input IDs

. 每个序列现在都是一个数字列表！结果是：

```
encoded_sequences = [  
    [101, 7592, 999, 102],  
    [101, 4658, 1012, 102],  
    [101, 3835, 999, 102],  
]
```

这是一个编码序列列表：一个列表列表。张量只接受矩形（想想矩阵）。此“数组”已为矩形，因此将其转换为张量很容易：

```
import torch  
model_inputs = torch.tensor(encoded_sequences)
```

```
import tensorflow as tf  
model_inputs = tf.constant(encoded_sequences)
```

使用张量作为模型的输入

在模型中使用张量非常简单-我们只需将输入称为模型：

```
output = model(model_inputs)
```

虽然模型接受许多不同的参数，但只需要

input IDs。我们稍后将解释其他参数的作用以及何时需要它们，但首先我们需要更仔细地了解 Transformer 模型可以理解的输入的标记

标记器 (Tokenizer)

标记器(Tokenizer)是 NLP 管道的核心组件之一。它们有一个目的：将文本转换为模型可以处理的数据。模型只能处理数字，因此标记器(Tokenizer)需要将我们的文本输入转换为数字数据。在本节中，我们将确切地探讨标记化管道中发生的事情。

在 NLP 任务中，通常处理的数据是原始文本。这是此类文本的示例

```
Jim Henson was a puppeteer
```

但是，模型只能处理数字，因此我们需要找到一种将原始文本转换为数字的方法。这就是标记器(tokenizer)所做的，并且有很多方法可以解决这个问题。目标是找到最有意义的表示——即对模型最有意义的表示——并且如果可能的话，找到最小的表示。

让我们看一下标记化算法的一些示例，并尝试回答您可能对标记化提出的一些问题。

基于词的(Word-based)

想到的第一种标记器是基于词的(*word-based*)。它通常很容易设置和使用，只需几条规则，并且通常会产生不错的结果。例如，在下图中，目标是将原始文本拆分为单词并为每个单词找到一个数字表示：

Split on spaces

Let's	do	tokenization!
-------	----	---------------

Split on punctuation

Let	's	do	tokenization	!
-----	----	----	--------------	---

有多种方法可以拆分文本。例如，我们可以通过应用Python的 `split()` 函数，使用空格将文本标记为单词：

```
tokenized_text = "Jim Henson was a puppeteer".split()
print(tokenized_text)
```

```
['Jim', 'Henson', 'was', 'a', 'puppeteer']
```

还有一些单词标记器的变体，它们具有额外的标点符号规则。使用这种标记器，我们最终可以得到一些非常大的“词汇表”，其中词汇表由我们在语料库中拥有的独立标记的总数定义。

每个单词都分配了一个 ID，从 0 开始一直到词汇表的大小。该模型使用这些 ID 来识别每个单词。

如果我们想用基于单词的标记器(tokenizer)完全覆盖一种语言，我们需要为语言中的每个单词都有一个标识符，这将生成大量的标记。例如，英语中有超过 500,000 个单词，因此要构建从每个单词到输入 ID 的映射，我们需要跟踪这么多 ID。此外，像“dog”这样的词与“dogs”这样的词的表示方式不同，模型最初无法知道“dog”和“dogs”是相似的：它会将这两个词识别为不相关。这同样适用于其他相似的词，例如“run”和“running”，模型最初不会认为它们是相似的。

最后，我们需要一个自定义标记(token)来表示不在我们词汇表中的单词。这被称为“未知”标记(token)，通常表示为“[UNK]”或“<unk>”。如果你看到标记器产生了很多这样的标记，这通常是一个不好的迹象，因为它无法检索到一个词的合理表示，并且你会在这个过程中丢失信息。制作词汇表时的目标是以这样一种方式进行，即标记器将尽可能少的单词标记为未知标记。

减少未知标记数量的一种方法是使用更深一层的标记器(tokenizer)，即基于字符的(*character-based*)标记器(tokenizer)。

基于字符(Character-based)

基于字符的标记器(tokenizer)将文本拆分为字符，而不是单词。这有两个主要好处：

- 词汇量要小得多。

- 词汇外（未知）标记(token)要少得多，因为每个单词都可以从字符构建。但是这里也出现了一些关于空格和标点符号的问题：

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

这种方法也不是完美的。由于现在表示是基于字符而不是单词，因此人们可能会争辩说，从直觉上讲，它的意义不大：每个字符本身并没有多大意义，而单词就是这种情况。然而，这又因语言而异；例如，在中文中，每个字符比拉丁语言中的字符包含更多的信息。

另一件要考虑的事情是，我们的模型最终会处理大量的词符(token)：虽然使用基于单词的标记器(tokenizer)，单词只会是单个标记，但当转换为字符时，它很容易变成 10 个或更多的词符(token)。为了两全其美，我们可以使用结合这两种方法的第三种技术：子词标记化(subword tokenization)。

子词标记化

子词分词算法依赖于这样一个原则，即不应将常用词拆分为更小的子词，而应将稀有词分解为有意义的子词。

例如，“annoyingly”可能被认为是一个罕见的词，可以分解为“annoying”和“ly”。这两者都可能作为独立的子词出现得更频繁，同时“annoyingly”的含义由“annoying”和“ly”的复合含义保持。

这是一个示例，展示了子词标记化算法如何标记序列“Let's do tokenization!”：

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

这些子词最终提供了很多语义含义：例如，在上面的示例中，“tokenization”被拆分为“token”和“ization”，这两个具有语义意义同时节省空间的词符(token)（只需要两个标记(token)代表一个长词）。这使我们能够对较小的词汇表进行相对较好的覆盖，并且几乎没有未知的标记。这种方法在土耳其语等粘着型语言(agglutinative languages)中特别有用，您可以通过将子词串一起来形成（几乎）任意长的复杂词。

还有更多！

不出所料，还有更多的技术。仅举几例：

- Byte-level BPE, 用于 GPT-2
 - WordPiece, 用于 BERT
 - SentencePiece or Unigram, 用于多个多语言模型
- 您现在应该对标记器(tokenizers)的工作原理有足够的了解，以便开始使用 API。

加载和保存

加载和保存标记器(tokenizer)就像使用模型一样简单。实际上，它基于相同的两种方法：

`from_pretrained()` 和 `save_pretrained()`。这些方法将加载或保存标记器(tokenizer)使用的算法（有点像建筑学(architecture)的模型）以及它的词汇（有点像权重(weights)模型）。加载使用与 BERT 相同的检查点训练的 BERT 标记器(tokenizer)与加载模型的方式相同，除了我们使用 `BertTokenizer` 类：

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
```

如同 `AutoModel` , `AutoTokenizer` 类将根据检查点名称在库中获取正确的标记器(tokenizer)类, 并且可以直接与任何检查点一起使用:

如同 `TFAutoModel` , `AutoTokenizer` 类将根据检查点名称在库中获取正确的标记器(tokenizer)类, 并且可以直接与任何检查点一起使用:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

我们现在可以使用标记器(tokenizer), 如上一节所示:

```
tokenizer("Using a Transformer network is simple")
```

```
{'input_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

保存标记器(tokenizer)与保存模型相同:

```
tokenizer.save_pretrained("directory_on_my_computer")
```

我们在[Chapter 3](#)中将更多地谈论 `token_type_ids` , 稍后我们将解释 `attention_mask` 键。首先, 让我们看看 `input_ids` 如何生成。为此, 我们需要查看标记器(tokenizer)的中间方法。

编码

将文本翻译成数字被称为编码(encoding)。编码分两步完成: 标记化, 然后转换为输入 ID。

正如我们所见, 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(token)*。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标记器(tokenizer), 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, 标记器(tokenizer)有一个 *词汇(vocabulary)*, 这是我们在实例化它时下载的部分 `from_pretrained()` 方法。同样, 我们需要使用模型预训练时使用的相同词汇。

为了更好地理解这两个步骤, 我们将分别探讨它们。请注意, 我们将使用一些单独执行部分标记化管道的方法向您展示这些步骤的中间结果, 但实际上, 您应该直接在您的输入上调用标记器(tokenizer) (如第 2 部分所示) 。

标记化

标记化过程由标记器(tokenizer)的 `tokenize()` 方法实现:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
sequence = "Using a Transformer network is simple"
tokens = tokenizer.tokenize(sequence)
print(tokens)
```

此方法的输出是一个字符串列表或标记(token):

```
['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']
```

这个标记器(tokenizer)是一个子词标记器(subword tokenizer)：它对词进行拆分，直到获得可以用其词汇表表示的标记(token)。`transformer` 就是这种情况，它分为两个标记：`transform` 和 `##er`。

从词符(token)到输入 ID

输入 ID 的转换由标记器(tokenizer)的 `convert_tokens_to_ids()` 方法实现：

```
ids = tokenizer.convert_tokens_to_ids(tokens)
print(ids)
```

```
[7993, 170, 11303, 1200, 2443, 1110, 3014]
```

这些输出一旦转换为适当的框架张量，就可以用作模型的输入，如本章前面所见。

✍ **试试看！** 我们在第 2 节中使用的输入句子 (“I've been waiting for a HuggingFace course my whole life.”和“I hate this so much!”) 复制最后两个步骤（标记化和转换为输入 ID）。检查您获得的输入 ID 是否与我们之前获得的相同！

解码

解码(Decoding) 正好相反：从词汇索引中，我们想要得到一个字符串。这可以通过 `decode()` 方法实现，如下：

```
decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])
print(decoded_string)
```

```
'Using a Transformer network is simple'
```

请注意，`decode` 方法不仅将索引转换回标记(token)，还将属于相同单词的标记(token)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（根据提示生成的文本，或序列到序列问题（如翻译或摘要））时，这种行为将非常有用。

到现在为止，您应该了解标记器(tokenizer)可以处理的原子操作：标记化、转换为 ID 以及将 ID 转换回字符串。然而，我们只是刮到了冰山一角。在下一节中，我们将采用我们的方法来克服它的限制，并看看如何克服它们。

处理多个序列

在上一节中，我们探讨了最简单的用例：对一个小长度的序列进行推理。然而，一些问题已经出现：

- 我们如何处理多个序列？
- 我们如何处理多个序列不同长度？
- 词汇索引是让模型正常工作的唯一输入吗？
- 是否存在序列太长的問題？

让我们看看这些问题会带来什么样的问题，以及如何使用 🤖 Transformers API 解决它们

模型需要一批输入

在上一个练习中，您看到了序列如何转换为数字列表。让我们将此数字列表转换为张量，并将其发送到模型：

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequence = "I've been waiting for a HuggingFace course my whole life."
tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
input_ids = torch.tensor(ids)
# This line will fail.
model(input_ids)
```

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

```
import tensorflow as tf
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
sequence = "I've been waiting for a HuggingFace course my whole life."
tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
input_ids = tf.constant(ids)
# This line will fail.
model(input_ids)
```

InvalidArgumentError: Input to reshape is a tensor with 14 values, but the requested shape has 196 [Op:Reshape]

哦，不！为什么失败了？“我们遵循了第2节中管道的步骤。

问题是我们向模型发送了一个序列，而🤖 Transformers模型默认情况下需要多个句子。在这里，当我们将分词器应用于一个应用程序时，我们尝试在幕后完成分词器所做的一切，但如果仔细观察，您会发现它不仅将输入ID列表转换为张量，还在其顶部添加了一个维度：

```
tokenized_inputs = tokenizer(sequence, return_tensors="pt")
print(tokenized_inputs["input_ids"])
```

```
tensor([[ 101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172,
          2607,  2026,  2878,  2166,  1012,   102]])
```

```
tokenized_inputs = tokenizer(sequence, return_tensors="tf")
print(tokenized_inputs["input_ids"])
```

```
tf.Tensor: shape=(1, 16), dtype=int32, numpy=
array([[ 101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662,
          12172,  2607,  2026,  2878,  2166,  1012,   102]], dtype=int32)>
```

让我们重试并添加一个新维度：

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequence = "I've been waiting for a HuggingFace course my whole life."
tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
input_ids = torch.tensor([ids])
print("Input IDs:", input_ids)
output = model(input_ids)
print("Logits:", output.logits)
```

```
import tensorflow as tf
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
sequence = "I've been waiting for a HuggingFace course my whole life."
tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
input_ids = tf.constant([ids])
print("Input IDs:", input_ids)
output = model(input_ids)
print("Logits:", output.logits)
```

我们打印输入ID以及生成的logits-以下是输出:


```
Input IDs: [[ 1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172,
 2607, 2026,  2878,  2166,  1012]]
Logits: [[-2.7276,  2.8789]]
```

```
Input IDs: tf.Tensor(
[[ 1045  1005  2310  2042  3403  2005  1037 17662 12172  2607  2026  2878
 2166  1012]], shape=(1, 14), dtype=int32)
Logits: tf.Tensor([[-2.7276208  2.8789377]], shape=(1, 2), dtype=float32)
```

Batching 是一次通过模型发送多个句子的行为。如果你只有一句话，你可以用一个序列构建一个批次：

```
batched_ids = [ids, ids]
```

这是一批两个相同的序列！

 **Try it out!** 试试看！将此列表转换为张量并通过模型传递。检查您是否获得与之前相同的登录（但是只有两次）

批处理允许模型在输入多个句子时工作。使用多个序列就像使用单个序列构建批一样简单。不过，还有第二个问题。当你试图将两个（或更多）句子组合在一起时，它们的长度可能不同。如果您以前使用过张量，那么您知道它们必须是矩形，因此无法将输入ID列表直接转换为张量。为了解决这个问题，我们通常填充输入。

填充输入

以下列表不能转换为张量：

```
batched_ids = [  
    [200, 200, 200],  
    [200, 200]  
]
```

为了解决这个问题，我们将使用填充使张量具有矩形。Padding通过在值较少的句子中添加一个名为Padding token的特殊单词来确保我们所有的句子长度相同。例如，如果你有10个包含10个单词的句子和1个包含20个单词的句子，填充将确保所有句子都包含20个单词。在我们的示例中，生成的张量如下所示：

```
padding_id = 100  
batched_ids = [  
    [200, 200, 200],  
    [200, 200, padding_id],  
]
```

可以在tokenizer.pad_token_id中找到填充令牌ID。让我们使用它，将我们的两句话分别发送到模型中，并分批发送到一起：

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)  
sequence1_ids = [[200, 200, 200]]  
sequence2_ids = [[200, 200]]  
batched_ids = [  
    [200, 200, 200],  
    [200, 200, tokenizer.pad_token_id],  
]  
print(model(torch.tensor(sequence1_ids)).logits)  
print(model(torch.tensor(sequence2_ids)).logits)  
print(model(torch.tensor(batched_ids)).logits)
```

```
tensor([[ 1.5694, -1.3895]], grad_fn=<AddmmBackward>)  
tensor([[ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)  
tensor([[ 1.5694, -1.3895],  
        [ 1.3373, -1.2163]], grad_fn=<AddmmBackward>)
```

```
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)  
sequence1_ids = [[200, 200, 200]]  
sequence2_ids = [[200, 200]]  
batched_ids = [  
    [200, 200, 200],  
    [200, 200, tokenizer.pad_token_id],  
]  
print(model(tf.constant(sequence1_ids)).logits)  
print(model(tf.constant(sequence2_ids)).logits)  
print(model(tf.constant(batched_ids)).logits)
```

```
tf.Tensor([[ 1.5693678 -1.3894581]], shape=(1, 2), dtype=float32)  
tf.Tensor([[ 0.5803005 -0.41252428]], shape=(1, 2), dtype=float32)  
tf.Tensor(  
    [[ 1.5693681 -1.3894582]  
     [ 1.3373486 -1.2163193]], shape=(2, 2), dtype=float32)
```

我们批处理预测中的logits有点问题：第二行应该与第二句的logits相同，但我们得到了完全不同的值！这是因为Transformer模型的关键特性是关注层，它将每个标记上下文化。这些将考虑填充标记，因为它们涉及序列中的所有标记。为了在通过模型传递不同长度的单个句子时，或者在传递一批应用了相同句子和填充的句子时获得相同的结果，我们需要告诉这些注意层忽略填充标记。这是通过使用attention mask来实现的。

注意力面具

*Attention masks*是与输入ID张量形状完全相同的张量，用0和1填充：1s表示应注意相应的标记，0s表示不应注意相应的标记（即，模型的注意力层应忽略它们）。

让我们用attention mask完成上一个示例：

```
batched_ids = [
    [200, 200, 200],
    [200, 200, tokenizer.pad_token_id],
]
attention_mask = [
    [1, 1, 1],
    [1, 1, 0],
]
outputs = model(torch.tensor(batched_ids),
                 attention_mask=torch.tensor(attention_mask))
print(outputs.logits)
```

```
tensor([[ 1.5694, -1.3895],
        [ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)
```

```
batched_ids = [
    [200, 200, 200],
    [200, 200, tokenizer.pad_token_id],
]
attention_mask = [
    [1, 1, 1],
    [1, 1, 0],
]
outputs = model(tf.constant(batched_ids),
                 attention_mask=tf.constant(attention_mask))
print(outputs.logits)
```

```
tf.Tensor(
[[ 1.5693681 -1.3894582 ]
 [ 0.5803021 -0.41252586]], shape=(2, 2), dtype=float32)
```

现在我们得到了该批中第二个句子的相同登录。

请注意，第二个序列的最后一个值是一个填充ID，它在attention mask中是一个0值。

🔪 试试看！在第2节中使用的两个句子上手动应用标记化（“我一生都在等待拥抱课程。”和“我非常讨厌这个！”）。通过模型传递它们，并检查您是否获得与第2节中相同的登录。现在使用填充标记将它们批处理在一起，然后创建适当的注意掩码。检查通过模型时是否获得相同的结果！

长序列

对于Transformers模型，我们可以通过模型的序列长度是有限的。大多数模型处理多达512或1024个令牌的序列，当要求处理更长的序列时，会崩溃。此问题有两种解决方案：

- 使用支持的序列长度较长的模型。
- 截断序列。

模型有不同的支持序列长度，有些模型专门处理很长的序列。

[Longformer](#)

这是一个例子，另一个是

[LED](#)

. 如果您正在处理一项需要很长序列的任务，我们建议您查看这些模型。

否则，我们建议您通过指定max_sequence_length参数：

```
sequence = sequence[:max_sequence_length]
```

把它们放在一起

在最后几节中，我们一直在尽最大努力手工完成大部分工作。我们探讨了标记器的工作原理，并研究了标记化、到输入ID的转换、填充、截断和注意掩码。

然而，正如我们在第2节中所看到的，🤖 Transformers API可以通过一个高级函数为我们处理所有这些，我们将在这里深入讨论。当你直接在句子上调用标记器时，你会得到准备通过模型传递的输入

```
from transformers import AutoTokenizer
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
sequence = "I've been waiting for a HuggingFace course my whole life."
model_inputs = tokenizer(sequence)
```

这里，`model_inputs`

变量包含模型良好运行所需的一切。对于DistilBERT，它包括输入ID和注意力掩码(attention mask)。其他接受额外输入的模型也会有标记器对象的输出。

正如我们将在下面的一些示例中看到的，这种方法非常强大。首先，它可以标记单个序列：

```
sequence = "I've been waiting for a HuggingFace course my whole life."
model_inputs = tokenizer(sequence)
```

它还一次处理多个序列，并且API没有任何变化：

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
model_inputs = tokenizer(sequences)
```

它可以根据几个目标进行填充：

```
# will pad the sequences up to the maximum sequence length
model_inputs = tokenizer(sequences, padding="longest")
# will pad the sequences up to the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, padding="max_length")
# will pad the sequences up to the specified max length
model_inputs = tokenizer(sequences, padding="max_length", max_length=8)
```

它还可以截断序列：

```

sequences = ["I've been waiting for a HuggingFace course my whole life.", "So
have I!"]
# Will truncate the sequences that are longer than the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, truncation=True)
# Will truncate the sequences that are longer than the specified max length
model_inputs = tokenizer(sequences, max_length=8, truncation=True)

```

标记器对象可以处理到特定框架张量的转换，然后可以直接发送到模型。例如，在下面的代码示例中，我们提示标记器从不同的框架返回张量——`"pt"` 返回Py Torch张量，`"tf"` 返回TensorFlow张量，`"np"` 返回NumPy数组：

```

sequences = ["I've been waiting for a HuggingFace course my whole life.", "So
have I!"]
# Returns PyTorch tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="pt")
# Returns TensorFlow tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="tf")
# Returns NumPy arrays
model_inputs = tokenizer(sequences, padding=True, return_tensors="np")

```

特殊词符(token)

如果我们看一下标记器返回的输入 ID，我们会发现它们与之前的略有不同：

```

sequence = "I've been waiting for a HuggingFace course my whole life."
model_inputs = tokenizer(sequence)
print(model_inputs["input_ids"])
tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
print(ids)

```

```

[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878,
2166, 1012, 102]
[1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166,
1012]

```

一个在开始时添加了一个标记(token) ID，一个在结束时添加了一个标记(token) ID。让我们解码上面的两个ID序列，看看这是怎么回事：

```

print(tokenizer.decode(model_inputs["input_ids"]))
print(tokenizer.decode(ids))

```

```

"[CLS] i've been waiting for a huggingface course my whole life. [SEP]"
"i've been waiting for a huggingface course my whole life."

```

标记器在开头添加了特殊单词 `[CLS]`，在结尾添加了特殊单词 `[SEP]`。这是因为模型是用这些数据预训练的，所以为了得到相同的推理结果，我们还需要添加它们。请注意，有些模型不添加特殊单词，或者添加不同的单词；模型也可能只在开头或结尾添加这些特殊单词。在任何情况下，标记器都知道需要哪些词符，并将为您处理这些词符。

结束：从标记器到模型

现在我们已经看到了标记器对象在应用于文本时使用的所有单独步骤，让我们最后一次看看它如何处理多个序列（填充！），非常长的序列（截断！），以及多种类型的张量及其主要API：

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
tokens = tokenizer(sequences, padding=True, truncation=True,
return_tensors="pt")
output = model(**tokens)
```

```
import tensorflow as tf
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = TFAutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]
tokens = tokenizer(sequences, padding=True, truncation=True,
return_tensors="tf")
output = model(**tokens)
```

基本用法完成！

很好地完成了到这里的课程！总而言之，在本章中，您可以：

- 学习了Transformers模型的基本构造块。
- 了解了标记化管道的组成。
- 了解了如何在实践中使用Transformers模型。
- 学习了如何利用分词器将文本转换为模型可以理解的张量。
- 将分词器和模型一起设置，以从文本到预测。
- 了解了inputs IDs的局限性，并了解了attention mask。
- 使用多功能和可配置的分词器方法。

从现在起，您应该能够自由浏览🔗 Transformers文档：词汇听起来很熟悉，并且您已经看到了大部分时间将使用的方法。