Embedding场景应用及实战

开源模型本地部署

开源模型是较为常用的部署方式，相较于大模型，Embedding的开源模型部署和运行没有太高的硬件需求，通常使用CPU也是可以运行的。目前开源的Embedding模型有很多，如OpenAI的text-embedding、BAAI的bge系列模型，你可以在中文海量文本embedding任务排行榜(C-MTEB)FlagEmbedding/C_MTEB/README.md at master · FlagOpen/FlagEmbedding (github.com)上找到中文效果最好的模型，本节以**当前中文表现最佳的BAAI的bge模型作为本地模型部署**的示例。

(1)环境要求

Embedding开源模型的推理运行不需要太高的硬件要求，通常4G显存的N卡就可以进行推理，如果你想用显卡推理，那么还需要安装CUDA和cuDNN。如果你没有显卡，也可以使用CPU来运行推理，不过速度相较显卡要慢得多。除此之外，只需要一个最基本的Python环境就够了。

(2)安装依赖库

首先安装必要的依赖库，Transformers和torch是深度学习最常用的两个库，几乎所有深度学习项目都需要使用。

```
pip install transformers
pip install torch
```

(3)模型部署

①导入必要的依赖库

```
from transformers import AutoTokenizer, AutoModel
import torch
```

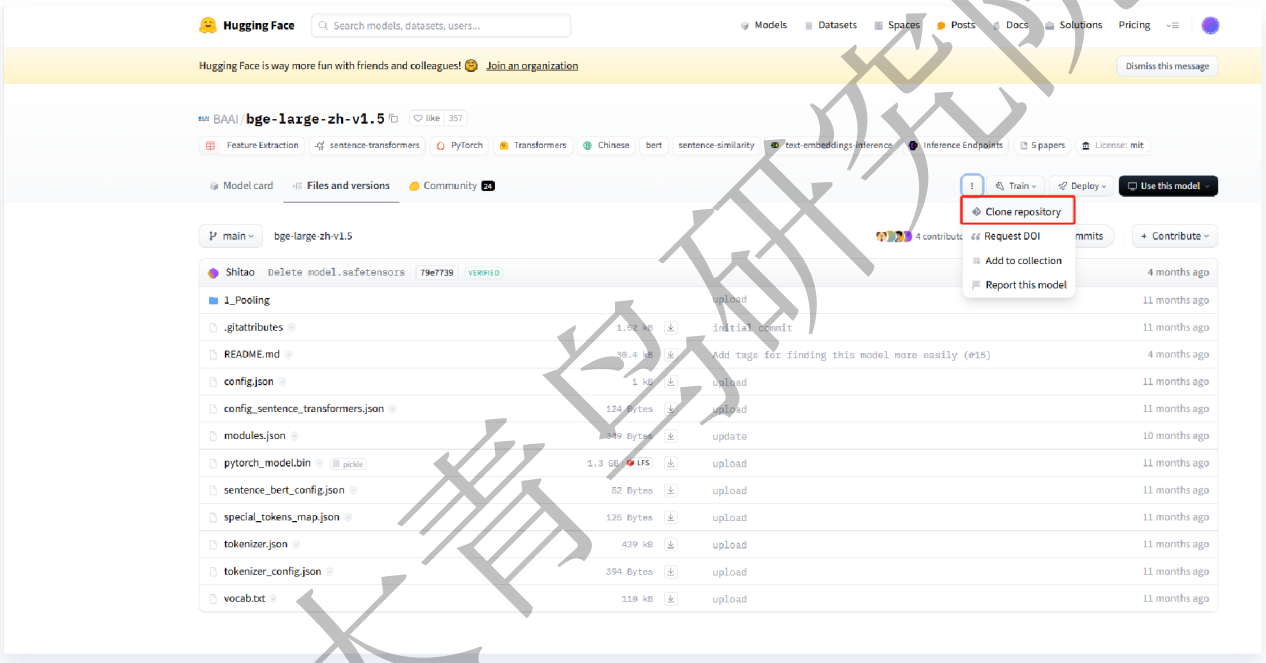
②加载tokenizer和模型

通过将下载的bge-large-zh-v1.5模型保存的本地，修改对应的路径地址，保证可以AutoTokenizer和AutoModel加载到预训练的词表和模型。

```
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-large-zh-v1.5')
model = AutoModel.from_pretrained('BAAI/bge-large-zh-v1.5')
```

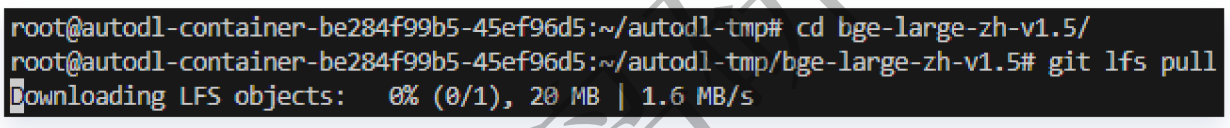
下载bge-large-zh-v1.5可以进入huggingface官网进行模型搜索，链接如下：BAAI/bge-large-zh-v1.5 · Hugging Face

进入后点击三个点图表，再点击 Clone repository，如图：

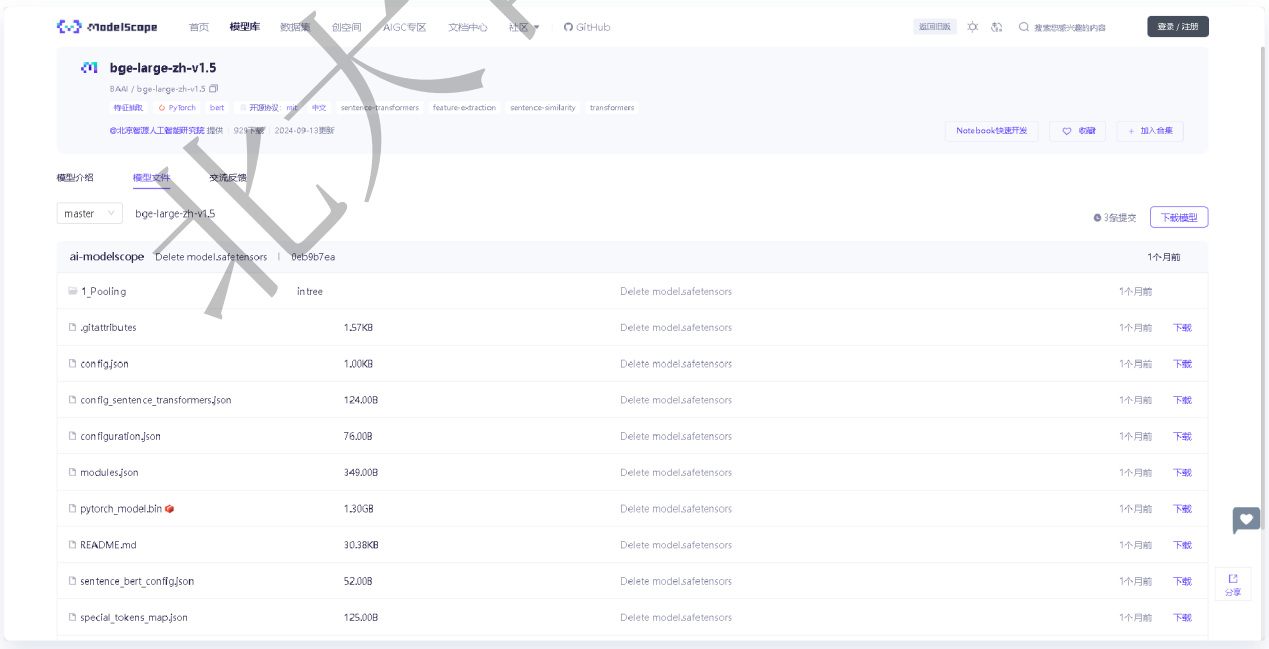


再按照下图执行操作：

这里需要注意，执行完 `git clone https://huggingface.co/BAAI/bge-large-zh-v1.5` 后，大文件可能会自动跳过，没有部署到本地，此时再进入到克隆的对应地址下，执行 `git lfs pull` 即可将大文件也进行拉取下载，当显示下载速度就证明模型已经开始下载啦。如图：



补充方式：如果huggingface下载缓慢，也可以尝试使用国内的魔搭社区平台下载，这是魔搭的模型地址：[bge-large-zh-v1.5 · 模型库 (modelscope.cn)]



魔搭提供了多种下载安装方式，下图中的两种方式选择其中一种即可：

SDK下载

```
#模型下载
from modelscope import snapshot_download
model_dir = snapshot_download('BAAI/bge-large-zh-v1.5')
```

Git下载

请确保 lfs 已经被正确安装

```
git lfs install
```

```
git clone https://www.modelscope.cn/BAAI/bge-large-zh-v1.5
```

注意：

1.SDK下载方式需要安装model scope的包，并通过python解释器运行 `python download.py`

```
pip install modelscope
```

2.Git下载方式同样需要确保 `lfs` 已安装，如果没有安装可以运行下列命令安装：

```
linux环境:
sudo apt update
sudo apt install git

windows环境:
git lfs install
```

③tokenizer计算和模型输出

```
text = "你好！"
encoded_input = tokenizer(text, max_length=512, padding=True, truncation=True,
return_tensors='pt')
with torch.no_grad():
    model_output = model(**encoded_input)

sentence_embeddings = model_output[0][:, 0]

print('向量：', sentence_embeddings)
```

输出为：

```
向量： tensor([[ 0.2386, -0.4734, -0.7710,  ..., -0.5432, -1.0447,  0.1269]])
```

当然，我们也可以批量处理文本，将多个文本封装在一个列表中进行输入：

```
text = ["你好，我是Molly", "Molly是个很可爱的机器人"]
encoded_input = tokenizer(text, max_length=512, padding=True, truncation=True,
return_tensors='pt')
with torch.no_grad():
     model_output = model(**encoded_input)
sentence_embeddings = model_output[0][:, 0]

print('向量：', sentence_embeddings)
```

输出为：

```
向量： tensor([[-0.1847, -0.4530, -0.1562,  ..., -0.1845, -0.9212,  0.6211],
        [-0.2706, -0.5714, -0.3028,  ...,  0.0525, -0.2178,  0.0513]])

进程已结束，退出代码为 0
```

完整代码为：

```
from transformers import AutoTokenizer, AutoModel
import torch
```

```
tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-large-zh-v1.5')
model = AutoModel.from_pretrained('BAAI/bge-large-zh-v1.5')

text = ["你好，我是Molly", "Molly是个很可爱的机器人"]
encoded_input = tokenizer(text, max_length=512, padding=True, truncation=True,
return_tensors='pt')  # 将文本转化为模型可读的形式（token索引组成的列表）
with torch.no_grad():
      model_output = model(**encoded_input)  # 输入模型进行嵌入

sentence_embeddings = model_output[0][:, 0]  # 这里的输出是一个二维tensor，
sentence_embeddings[0]代表第一个句子的向量，sentence_embeddings[1]代表第二个句子的向量

print('向量：', sentence_embeddings)
```

模型参数解析

(1)语言

在挑选模型时，首先需要关注的就是模型所适配的语言，比如 `bge-large-zh` 中的 `zh` 就代表中文。而 `en` 则代表英文，在选择模型时，首先根据官方文档，查找该模型适配的语言类型。这里**适配**的意思并不绝对，比如包含 `zh` 的模型并不是对英文一窍不通，只是其训练数据大多为中文数据，在中文效果上表现更好，部分模型还可能适配多个不同的语言。

(2)模型大小

在挑选模型时，我们同样需要关注模型的大小，比如在 `bge-large-zh` 中， `large` 就代表着大小，当然，不是所有的模型都会把模型大小写在命名里，在使用前，同样需要查找该模型的大小是多少。模型的大小通常由多种因素影响，如**Encoder层数**、**隐藏层维度**和**自注意力头数**等，在 `bge` 模型中， `large` 模型的隐藏层维度有1024维， `base` 模型有768维， `small` 模型有512维，在一句话被输入进模型时，模型会将这句话转换为向量，向量的维度越高，其包含的语义越丰富。

通常维度较高的模型也同样需要更多的层数和权重训练才得以表现更好，因此往往在模型体量上会表现的更大。 如 `large` 模型的**Encoder层数**为24层，而 `base` 模型只有12层，层数越大参数量越大，占用空间也越大，较大的模型可能有1-2个G，而较小的模型只有几百MB。在大部分情况下，模型所计算的维度越大，其语义理解的效果越好。因此如果你对语义理解有较高的精确度要求，那么请尽量使用维度较大的模型。

(3)最大句长限制

最大句长是模型对输入长度的限制，目前大部分Embedding模型都将最大句长控制在 `512` 左右，当然模型参数规模越大，这个最大句子长度也会随着增加，比如1024， 2048。我们这里将最大句长设置为512，这么做的原因是，Embedding模型在做语义判断时，512个字符以内的文本通常语义较为明确，而超过512个字符就将面临更多的文字冗余，导致语义发散，同样，Embedding模型的维度是有限的，无法支持更长句长文本的精准的语义表示，因此Embedding模型的最大句长通常在千字以内。如果你输入了超出最大句长的句子，在模型输入时通常会发生"错误"，出现**语义理解不完善**的现象，因为超过设定句长的部分会被截断掉，不能输入到模型中。

因此在上述代码中，我们需要使用 `tokenizer(text, max_lenth=512)` 来帮助截断文本。如果在 `tokenizer` 中处理的文本超过了最大句长限制，那么 `tokenizer` 会为我们截断最大句长后的所有文本。

语义相似度计算

在搭建Embedding模型后，我们可以试着计算两个向量的相似度，在RAG流程中，正是语义相似度计算，才得以找到和初始文本最为相似的目标文本。以下是一些常见的相似度计算方式：

1. **欧几里得距离（Euclidean Distance）**：
   - 用于计算两个点（通常在高维空间中）之间的直线距离。其公式是两点坐标差的平方和的平方根。
   - 在n维空间中，两点P_1和P_2之间的欧氏距离可以表示为：

$$\text{euclidean\_distance}(P_1, P_2) = \sqrt{\sum_{i=1}^{n}(x_{2i} - x_{1i})^2}$$

> 举个例子
>
> 给定向量：
> $\mathbf{a} = (1, 2)$
> $\mathbf{b} = (2, 1)$
>
> #### 计算过程
>
> 1. 计算对应元素的差的平方：
> $(a_1 - b_1)^2 = (1 - 2)^2 = (-1)^2 = 1$
> $(a_2 - b_2)^2 = (2 - 1)^2 = 1^2 = 1$
> 2. 将这些平方值相加：
> $1 + 1 = 2$

3. 取和的平方根：

$\sqrt{2}$

因此，向量 (\mathbf{a}) 和 (\mathbf{b}) 之间的欧几里得距离为：

$\boxed{\sqrt{2}}$

1. 余弦相似度（Cosine Similarity）：

- 衡量两个向量之间的夹角余弦值，常用于文本相似度计算。其值范围在 [-1, 1]，1 表示两个向量完全相同，0 表示完全不相似。
- 假设有两个(n)维向量(A = [a_1, a_2, ..., a_n])和(B = [b_1, b_2, ..., b_n])，则余弦相似度可以表示为：

$$\text{cosine\_similarity}(A, B) = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \times \sqrt{\sum_{i=1}^{n} b_i^2}}$$

简单举个例子

给定向量：

$\mathbf{a} = (1, 2)$

$\mathbf{b} = (2, 1)$

第一步：计算点积 $\mathbf{a} \cdot \mathbf{b}$

$\mathbf{a} \cdot \mathbf{b} = 1 \cdot 2 + 2 \cdot 1 = 2 + 2 = 4$

第二步：计算欧几里得范数 $\|\mathbf{a}\|$和 $\|\mathbf{b}\|$

$\|\mathbf{a}\| = \sqrt{1^2 + 2^2} = \sqrt{1 + 4} = \sqrt{5}$

$\|\mathbf{b}\| = \sqrt{2^2 + 1^2} = \sqrt{4 + 1} = \sqrt{5}$

第三步：计算余弦相似度

余弦相似度 $= \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \frac{4}{\sqrt{5} \cdot \sqrt{5}} = \frac{4}{5}$

因此，向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的余弦相似度为：

$\boxed{\frac{4}{5}}$

3. 杰卡尔指数（Jaccard Index）：

- 计算两个集合的交集大小与并集大小的比值，常用于集合相似度分析。
- 对于两个n维向量(A)和(B)，Jaccard相似度可以表示为：

$$\text{jaccard\_similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

4. 曼哈顿距离（Manhattan Distance）：

- 计算两个点之间在各个坐标轴上的绝对差的总和，有时也称为“城市街区”距离。

- 在n维空间中，两点P_1和P_2之间的曼哈顿距离可以表示为：

  $$\text{manhattan\_distance}(P_1, P_2) = \sum_{i=1}^{n} |x_{2i} - x_{1i}|$$

5. 皮尔逊相关系数（Pearson Correlation Coefficient）：

   - 衡量两组数据之间的线性相关性，范围为 [-1, 1]。值为 1 表示完全正相关，0 表示无关联，-1 表示完全负相关。

6. 汉明距离（Hamming Distance）：

   - 用于比较两个长度相同的字符串之间的差异，计算不同字符的位置的数量。

7. 布雷柯蒂斯距离（Bray-Curtis Distance）：

   - 一种统计学中常用的距离计算方法，通常用于生态学和生物多样性分析。

8. 动态时间规整（Dynamic Time Warping, DTW）：

   - 特别适用于时间序列数据，用于计算非同步信号的相似度。

9. Levenshtein距离（编辑距离）：

- 计算将一个字符串转换为另一个字符串所需的插入、删除或替换操作的最小次数。

其中，在基于Embedding模型的语义相似度计算中，最常用的方法是余弦相似度。

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

首先，导入必要的依赖库

```
import torch.nn.functional as F
```

在输入文本中，我们定义需要进行相似度计算的两段文本。

```
text_list = ["Molly是谁? ", "Molly是一个会编程的AI机器人"]
```

接下来输入进模型

```
encoded_input = tokenizer(text_list, max_length=512, padding=True,
truncation=True, return_tensors='pt')
with torch.no_grad():
    model_output = model(**encoded_input)

sentence_embeddings = model_output[0][:, 0]
```

分别取出两个向量

```
tensor_a = sentence_embeddings[0]
tensor_b = sentence_embeddings[1]
```

计算余弦相似度的值

```
# 函数默认是传入两个二维向量，然后对第二个维度做计算
# 这里的两个向量都是一维的，所以要对第一个维度做计算
cosine_similarity = F.cosine_similarity(tensor_a, tensor_b, dim=0)
print('cos_similar: ', cosine_similarity.item())  # 使用.item()从中取出标量
```

输出为：

```
cos_similar:  0.6483673453330994
```

这里的余弦相似度值越大，代表两个句子越相似。如果我们使用两个完全不相关的句子来试试：

```
text_list = ["Molly是谁？", "我喜欢快乐星球"]
```

则输出如下，可以看到，相关性较低的两个句子余弦相似度值比较小。

```
cos_similar:  0.21184760332107544

进程已结束，退出代码为 0
```

完整代码如下

```
from transformers import AutoTokenizer, AutoModel
import torch
```

```
import torch.nn.functional as F

tokenizer = AutoTokenizer.from_pretrained('BAAI/bge-large-zh-v1.5')
model = AutoModel.from_pretrained('BAAI/bge-large-zh-v1.5')

text_list = ["Molly是谁? ", "我喜欢快乐星球"]

encoded_input = tokenizer(text_list, max_length=512, padding=True,
truncation=True, return_tensors='pt')
with torch.no_grad():
      model_output = model(**encoded_input)

sentence_embeddings = model_output[0][:, 0]

# 定义两个向量
tensor_a = sentence_embeddings[0]
tensor_b = sentence_embeddings[1]
# 计算余弦相似度
cosine_similarity = F.cosine_similarity(tensor_a, tensor_b, dim=0)

print('cos_similar: ', cosine_similarity.item())  # 使用.item()从中取出标量
```

本节内容中，我们学习了如何部署开源的Embedding本地模型，了解了模型比较重要的参数，最后学习了如何计算两个文本的语义相似度。接下来，我们将利用我们部署的BGE模型对pdf文本数据进行嵌入实战。

实战：数据集的embedding处理

快速开始

首先，加载刚刚部署的bge-large-v1.5向量模型：

```
from FlagEmbedding import FlagModel

model = FlagModel('BAAI/bge-base-en-v1.5',
                  query_instruction_for_retrieval="Represent this sentence
for searching relevant passages:",
                  use_fp16=True)
```

将语句作为模型输入，得到向量：

```
sentences_1 = ["I love NLP", "I love machine learning"]
sentences_2 = ["I love BGE", "I love text retrieval"]
embeddings_1 = model.encode(sentences_1)
embeddings_2 = model.encode(sentences_2)
```

取得向量后，通过内积计算相似度：

```
similarity = embeddings_1 @ embeddings_2.T
print(similarity)
```

中文数据集Embedding

刚刚我们使用我们部署的Embedding模型进行了快速演示，相信大家对如何使用Embedidng模型有了一个初步的认知。接下来我们将带大家对一个中文数据集进行Embedding操作：

1.导入必要的库

```
from transformers import AutoTokenizer, AutoModel
import torch
from pdfminer.high_level import extract_text
```

- `transformers` 是一个流行的机器学习库，用于处理自然语言处理任务，这里导入了 `AutoTokenizer` 和 `AutoModel` 类，分别用于加载分词器和预训练模型。
- `torch` 是PyTorch库，用于构建和训练深度学习模型，这里主要用于处理张量数据。
- `pdfminer.high_level.extract_text` 是一个用于从PDF文件中提取文本的功能，简化了文本提取的过程。

2.加载模型和分词器

```
# 加载预训练模型和分词器
tokenizer = AutoTokenizer.from_pretrained('bge-large-zh-v1.5')
model = AutoModel.from_pretrained('bge-large-zh-v1.5')
```

- 使用 `AutoTokenizer.from_pretrained` 和 `AutoModel.from_pretrained` 方法加载预训练的中文分词器和模型。这里的 `'bge-large-zh-v1.5'` 指定了模型和分词器的存储路径。

3.加载PDF文件、提取文本并切割文本：

```
# 加载PDF文件并提取文本
def load_pdf_text(pdf_path):
    return extract_text(pdf_path)

# 加载PDF文件
pdf_path = "document.pdf"
pdf_text = load_pdf_text(pdf_path)

# 将PDF文本分割成多个句子或段落
sentences = pdf_text.split('。')  # 根据换行符分割文本
```

- `load_pdf_text` 函数:接受一个PDF文件的路径作为输入，使用 `extract_text` 库函数来提取文件中的文本内容，并返回这个文本字符串。
- `加载PDF文件`：指定要处理的PDF文件的路径，并调用之前定义的 `load_pdf_text` 函数来获取文件中的文本。
- `分割文本`：将提取到的PDF文本按照中文句号 `。` 分割成多个句子。注意，这种简单的分割方法可能不会总是准确地识别出句子边界，特别是对于复杂或非标准格式的文本。

4.对每个句子进行embedding：

```
encoded_inputs = tokenizer(sentences, max_length=512, padding=True,
truncation=True, return_tensors='pt')
with torch.no_grad():
    model_output = model(**encoded_inputs)
```

- 使用分词器对所有句子进行编码，设置最大长度为512（这是许多Transformer模型的默认限制），并确保所有的输入都填充到相同的长度或截断到指定的最大长度。 `return_tensors='pt'` 表示返回PyTorch张量。

- > **Tips:**
  >
  > `encoded_inputs` 是一个字典，通常包含以下键值对：
  >
  > `input_ids`：编码后的 token IDs。
  >
  > `attention_mask`：注意力掩码，用于指示哪些 token 是有效的，哪些是填充的。

- 使用 `torch.no_grad()` 上下文管理器来禁用梯度计算，因为在这个阶段不需要反向传播计算梯度。

- 将编码后的输入传递给模型，得到模型输出。

> 这里的 `**` 符号用于将 `encoded_inputs` 字典解包，即将字典中的键值对作为关键字参数传递给 `model` 函数。例如，假设 `encoded_inputs` 包含 `{'input_ids': tensor(...), 'attention_mask': tensor(...)}`，那么 `model(**encoded_inputs)` 相当于 `model(input_ids=tensor(...), attention_mask=tensor(...))`。
>
> 举个例子大家就明白了，假设 `encoded_inputs` 是：
>
> ```
> encoded_inputs = {
>  'input_ids': tensor([[ 101, 1110, 1111, 102]]),
>  'attention_mask': tensor([[1, 1, 1, 1]])
> }
> ```
>
> 那么 `model(**encoded_inputs)` 实际上等价于：
>
> ```
> model(input_ids=tensor([[ 101, 1110, 1111, 102]]),
> attention_mask=tensor([[1, 1, 1, 1]]))
> ```

5.获取embedding并打印：

```
# 获取每个句子的嵌入向量
sentence_embeddings = model_output.last_hidden_state[:, 0]  # [CLS] token的嵌入向量

# 打印每个句子的嵌入向量
for i, sentence in enumerate(sentences):
    print(f'句子: {sentence}')
    print(f'向量: {sentence_embeddings[i]}')
    print('-' * 50)
```

- 从模型输出中提取每个句子的嵌入向量。在大多数情况下，`last_hidden_state[:, 0]` 提取的是 `[CLS]` token 的嵌入向量，这通常被用作整个句子的表示。
- 遍历所有句子及其对应的嵌入向量，打印每个句子及其向量表示。这样可以帮助理解句子与向量之间的对应关系。

输出如下：

```
句子：第一章：开源大模型私有化部署
向量：tensor([ 0.3171, -0.0078, -1.0051,  ..., -0.7039, -0.2333, -0.5625])
--------------------------------------------------
句子：
向量：tensor([ 0.3110, -0.4797, -0.1707,  ..., -1.2277,  0.5674,  0.1316])
--------------------------------------------------
句子：本次课程是实战营，所以是以实战为主，会涉及大量实操和项目实践，大家无论课上还是课下，
向量：tensor([ 0.1330,  0.1218,  0.1451,  ..., -0.1217,  0.0662, -0.4682])
--------------------------------------------------
句子：
向量：tensor([ 0.3110, -0.4797, -0.1707,  ..., -1.2277,  0.5674,  0.1316])
--------------------------------------------------
句子：都一定要尽可能的多加练习，提升自己的动手能力。
向量：tensor([ 0.2449, -0.1398,  0.4169,  ..., -0.1902, -0.2938, -0.9162])
--------------------------------------------------
句子：
向量：tensor([ 0.3110, -0.4797, -0.1707,  ..., -1.2277,  0.5674,  0.1316])
--------------------------------------------------
句子：我们在进行实操课时，可以先跑起来，遇到一些不太懂的地方可以暂时忽略，不要太在意细节，
向量：tensor([-0.2206,  0.1136,  0.8173,  ..., -0.1326, -0.7689, -1.0463])
--------------------------------------------------
句子：
向量：tensor([ 0.3110, -0.4797, -0.1707,  ..., -1.2277,  0.5674,  0.1316])
--------------------------------------------------
```

附录：完整代码

```python
from transformers import AutoTokenizer, AutoModel
import torch
from pdfminer.high_level import extract_text

# 加载预训练模型和分词器
tokenizer = AutoTokenizer.from_pretrained('bge-large-zh-v1.5')
model = AutoModel.from_pretrained('bge-large-zh-v1.5')

# 加载PDF文件并提取文本
def load_pdf_text(pdf_path):
    return extract_text(pdf_path)

# 加载PDF文件
pdf_path = "document.pdf"
pdf_text = load_pdf_text(pdf_path)

# 将PDF文本分割成多个句子或段落
sentences = pdf_text.split('。')  # 根据换行符分割文本
```

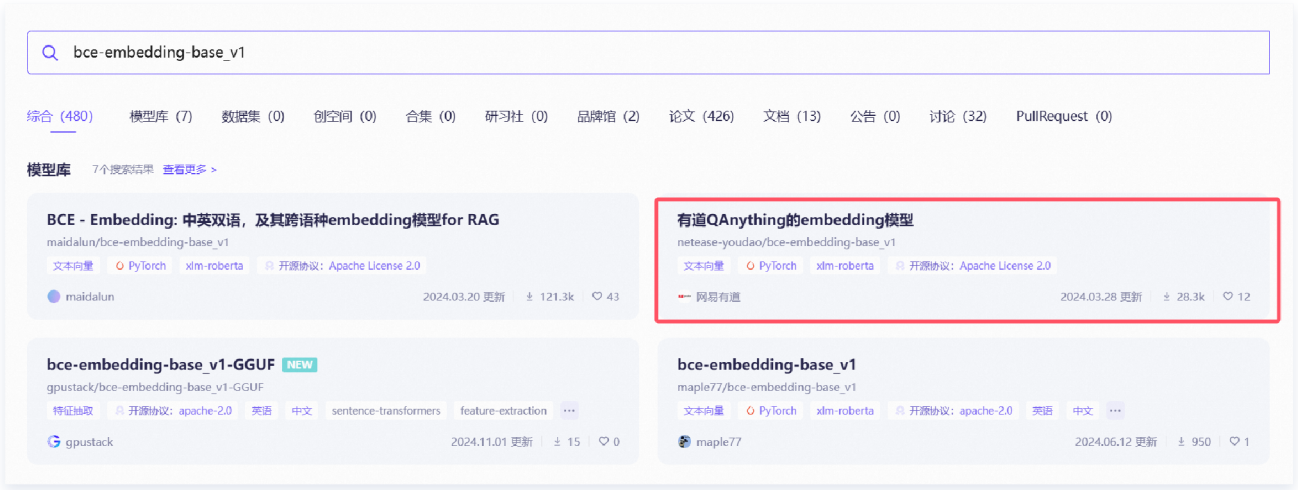
```
# 对每个句子进行embedding
encoded_inputs = tokenizer(sentences, max_length=512, padding=True,
truncation=True, return_tensors='pt')
with torch.no_grad():
    model_output = model(**encoded_inputs)

# 获取每个句子的嵌入向量
sentence_embeddings = model_output.last_hidden_state[:, 0]  # [CLS] token的嵌入
向量

# 打印每个句子的嵌入向量
for i, sentence in enumerate(sentences):
    print(f'句子: {sentence}')
    print(f'向量: {sentence_embeddings[i]}')
    print('-' * 50)
```

扩展：其他开源模型部署

bce-embedding-base_v1 部署

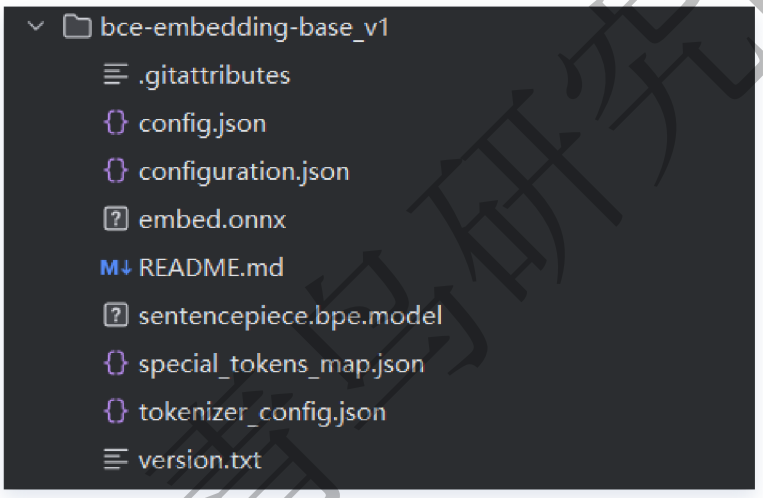
下载 `bce-embedding-base_v1` 模型，可以进入模型库首页 · 魔搭社区在右上角搜索栏处搜索模型名称

在搜索结果中找到 网易有道 官方发布的模型点击进入



下载方式和之前我们下载 `bge-large-zh-v1.5` 一样，这里就不再赘述了

下载完成后，文件夹中包含以下文件



这里我们将导入一个BCE专属的包来调用模型

使用 `pip install BCEmbedding==0.1.1` 指令来安装包

执行下列代码来获取embedding并打印

```
from BCEmbedding import EmbeddingModel

model = EmbeddingModel(model_name_or_path='bce-embedding-base_v1')
sentences = ['你好, 我是Molly',"Molly是个很可爱的机器人"]
embeddings = model.encode(sentences)

print(embeddings)
```

输出如下：

```
[[ 0.06181085  0.02614181 -0.00317803 ...  0.00614105 -0.05492359
   0.00033408]
 [ 0.0514755   0.02845236  0.01158819 ...  0.03150848 -0.05001081
  -0.00858175]]
```

GTE文本向量-中文-通用领域-large 部署

接下来我们介绍的这个embedding模型，功能十分强大，在魔搭平台上搜索 `Embedding` ，这个模型是通用领域的下载量第一。

点进模型界面， `模型介绍` 部分有这个模型的相关讲解，同学们可以课下进行学习

接下来我们先来简单学习一下怎么调用这个模型，其实就和之前调用 `bge-large-zh-v1.5` 的步骤是一样的，我们借这个模型再来加深一下印象

①导入必要的依赖库

```
from transformers import AutoTokenizer, AutoModel
import torch
```

②加载tokenizer和模型

通过将下载的**GTE文本向量-中文-通用领域-large**模型保存的本地，修改对应的路径地址，保证可以AutoTokenizer和AutoModel加载到预训练的词表和模型。

```
tokenizer = AutoTokenizer.from_pretrained('nlp_gte_sentence-embedding_chinese-large')
model = AutoModel.from_pretrained('nlp_gte_sentence-embedding_chinese-large')
```

③tokenizer计算和模型输出

```
text = ["你好，我是Molly", "Molly是个很可爱的机器人"]
encoded_input = tokenizer(text, max_length=512, padding=True, truncation=True, return_tensors='pt')
with torch.no_grad():
    model_output = model(**encoded_input)
sentence_embeddings = model_output[0][:, 0]

print(sentence_embeddings[0])
print(sentence_embeddings[1])
```

输出如下：

```
tensor([-0.9115,  0.5058,  0.0119,  ...,  0.7815,  1.1237, -0.0601])
tensor([-0.9070, -0.4182,  0.2724,  ...,  0.6226,  1.1238,  0.2900])
```