1. Transformer

Transformer 库源于 Google 在 2017 年提出的 Transformer 模型（《Attention is All You Need》论文），这是第一个完全基于自注意力机制的模型，极大地推动了自然语言处理（NLP）和生成任务的发展。该模型的结构跳过了传统 RNN 和 CNN 的限制，从而提高了训练效率和模型性能。

Hugging Face 开发的 Transformers 库是一种开源的工具包，支持不同的 NLP 任务，如文本分类、情感分析、翻译和生成等。这个库包含了大量预训练模型，例如 BERT、GPT、T5 等，可以通过少量代码和训练资源应用于各种任务，极大地简化了模型的应用。

Transformers 库的优点：

- **预训练模型**：Transformers 库提供了多种大型预训练模型，涵盖各种语言和任务，为研究人员和开发者节省了大量训练时间。
- **简单易用的 API**：库的 API 设计友好，可以快速加载模型、进行推理和微调，非常适合初学者和开发者。
- **广泛支持的任务**：除 NLP 任务外，该库还支持图像和多模态任务，在更广泛的应用场景中提供了深度学习支持。

1.1. Transformer调用Qwen模型（Qucik start)

```
from transformers import AutoModelForCausalLM, AutoTokenizer


def load_model(model_path):
    # device = "cuda"  # 将模型加载到 GPU 上
    device = "cuda:0"  # 将模型加载到指定GPU 上
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        torch_dtype="auto",
        # device_map="auto"
        device_map={"": 0}  # 指定模型加载到第 5 张 GPU 上
    )
    tokenizer = AutoTokenizer.from_pretrained(model_path)#加载分词器
    return device, tokenizer, model

# 定义一个名为 chat_qwen 的函数，用于与模型进行交互。
def chat_qwen(device, tokenizer, model, prompt):
    messages = (
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt}
    )
    # 使用分词器的 apply_chat_template 方法将消息格式化为模型可理解的输入格式
    text = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    model_inputs = tokenizer((text), return_tensors="pt").to(device)
    #生成模型输出
    generated_ids = model.generate(
        model_inputs.input_ids,
        max_new_tokens=512
    )
```

```
    # 由于模型输出包括输入模型，这里切去输入部分
    generated_ids = [output_ids[len(input_ids):] for input_ids, output_ids in
zip(model_inputs.input_ids, generated_ids)]
    # 将模型输出解码为文本
    response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
    print(response)

# 载入模型，这里需要更改为本地模型的具体路径
model_path = './qwen/Qwen2___5-7B-Instruct'

device, tokenizer, model = load_model(model_path)

# 进行测试
chat_qwen(device, tokenizer, model,'你好，请你介绍一下自己')
```

接下来，我们依次解释这段代码

1.1.1. 加载基座大模型和分词器

```
def load_model(model_path):
    # device = "cuda"  # 将模型加载到 GPU 上
    device = "cuda:0"  # 将模型加载到指定GPU 上
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        torch_dtype="auto",
        # device_map="auto"
        device_map={"": 0}  # 指定模型加载到指定 GPU 上
    )
    tokenizer = AutoTokenizer.from_pretrained(model_path)#加载分词器
    return device, tokenizer, model
```

1.1.1.1. AutoModelForCausalLM

`AutoModelForCausalLM` 是一个自动类，用于加载任何支持因果语言建模（Causal Language Modeling，CLM）的预训练模型。因果语言建模主要用于生成任务，如文本生成。这个类能够自动识别和加载给定的预训练模型配置，无需指定具体的模型类（如 GPT、GPT-2）。这使得它在处理不同的模型时非常灵活，简化了代码的复杂性。

（1）AutoModelForCausalLM.from_pretrained

在 Hugging Face 的 Transformers 库中，`AutoModelForCausalLM.from_pretrained` 方法是用来加载一个预训练的因果语言模型（Causal Language Model）。这里使用`from_pretrained`方法从指定的本地目录加载模型和分词器。目录"./Qwen/Qwen1___5-4B-Chat"应包含预训练模型的文件。`torch_dtype="auto"`自动选择合适的torch数据类型，`device_map="auto"`允许自动分配模型到可用的设备上。以下是一些主要参数及其作用：

1.model_name_or_path

- **作用**：指定预训练模型的名称或路径。可以是 Hugging Face Model Hub 上的模型标识符，也可以是本地文件系统上的路径。

- **示例**：`"gpt2"` 或 `"./my_model_directory/"`

2. cache_dir

- **作用**：指定模型和配置文件的缓存目录。如果未指定，将使用默认的缓存目录。
- **示例**：`"./my_cache_directory/"`

3. force_download

- **作用**：即使在缓存中已有模型的情况下，也强制重新下载模型。
- **类型**：布尔值，如 `True` 或 `False`

4. resume_download

- **作用**：如果下载被中断，则从中断的地方继续下载。
- **类型**：布尔值，如 `True` 或 `False`

5. proxies

- **作用**：定义代理配置以使用代理服务器进行下载。
- **示例**：`{"http": "http://10.10.1.10:3128", "https": "https://10.10.1.10:1080"}`

6. revision

- **作用**：指定模型的特定修订版，主要用于模型版本控制。
- **示例**：`"main"` 或某个特定的 git 哈希值。

7. use_auth_token

- **作用**：如果你需要通过认证的方式访问私有模型或者使用额外的 API 令牌，可以使用这个参数。
- **示例**：可以是一个字符串或 `True`，如果是 `True`，则会使用 Hugging Face 库配置的 token。

8. local_files_only

- **作用**：只从本地文件系统加载模型，不尝试从互联网下载。
- **类型**：布尔值，如 `True` 或 `False`

9. torch_dtype

- **作用**：设置加载模型时使用的 PyTorch dtype。这对于模型精度和性能调优特别有用。
- **示例**：`torch.float64`、`torch.float32`、`torch.float16`、`torch.int8`、`torch.int4`

10. device_map

- **作用**：`device_map` 参数用于控制模型的计算图如何分配到不同的设备（如GPU、CPU）上，尤其在多个GPU环境下，用来优化显存利用和加速模型推理或训练。
  - `"sequential"`：模型按顺序加载到多个GPU上，并依次在这些设备之间传递数据。这种方式适合在多个GPU内存有限的情况下，将不同层分配到不同的GPU上。例如 `device_map="sequential"``
  - `"auto"`：自动检测系统中的设备并根据负载情况在设备之间进行最优分配。它会智能地将模型的一部分加载到不同的GPU或CPU中，尽可能平衡资源。例如`device_map="auto"`
  - "balanced"：尽可能均匀地将模型加载到所有可用的设备上，适合在多个设备之间平衡负载。
  - "balanced_low_0"：与balanced`类似，但会优先使用设备0上更多的内存（通常是主GPU），然后在其他设备之间平衡。

- 手动指定：device_map={"": "cuda:0"}；指定在gpu0上加载模型
- 指定多个GPU：

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

def load_model(model_path):
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        torch_dtype=torch.float16,  # 使用 float16 来减少内存使用
        device_map="auto",
        max_memory={0: "24GiB", 1: "24GiB"},  # 根据你的GPU内存大小调整
        low_cpu_mem_usage=True
    )

    tokenizer = AutoTokenizer.from_pretrained(model_path)

    # 使用 GPU 0 作为主设备
    main_device = 'cuda:0'
    return main_device, tokenizer, model

def chat_qwen(main_device, tokenizer, model, prompt):
    messages = (
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt}
    )
    text = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    model_inputs = tokenizer(text, return_tensors="pt").to(main_device)

    generated_ids = model.generate(
        model_inputs.input_ids,
        max_new_tokens=512
    )

    generated_ids = generated_ids[:, model_inputs.input_ids.shape[-1]:]
    response = tokenizer.batch_decode(generated_ids,
skip_special_tokens=True)[0]
    print(response)

# 载入模型
model_path = './qwen/Qwen2___5-7B-Instruct'
main_device, tokenizer, model = load_model(model_path)

# 打印模型的设备分布
print(model.hf_device_map)

# 进行测试
chat_qwen(main_device, tokenizer, model, '你好，请你介绍一下自己')
```

1.1.1.2. `AutoTokenizer`

`AutoTokenizer` 同样是一个自动类，它用于加载与给定的预训练模型相对应的分词器。分词器负责文本的预处理，包括分词、添加必要的特殊标记（如起始、结束标记），以及将文本转换为模型可以理解的数字格式（通常是 token ID）。由于不同的模型架构可能需要不同格式的输入，因此每个模型通常都有与之配套的分词器。预训练的大模型通常与特定的分词器配套使用。这是因为模型在训练过程中使用了特定的分词方式，如果使用不同的分词器，可能会导致模型性能下降或者完全无法理解输入的数据。以下是一些主要参数及其作用：

1. pretrained_model_name_or_path

- **作用**：指定预训练分词器的名称或路径。可以是 Hugging Face Model Hub 上的模型标识符，也可以是本地文件系统上的路径。
- **示例**：`"bert-base-uncased"` 或 `"./my_tokenizer_directory/"`

2. cache_dir

- **作用**：指定分词器和配置文件的缓存目录。如果未指定，将使用默认的缓存目录。
- **示例**：`"./my_cache_directory/"`

3. force_download

- **作用**：即使在缓存中已有分词器的情况下，也强制重新下载分词器。
- **类型**：布尔值，如 `True` 或 `False`

4. resume_download

- **作用**：如果下载被中断，则从中断的地方继续下载。
- **类型**：布尔值，如 `True` 或 `False`

5. proxies

- **作用**：定义代理配置以使用代理服务器进行下载。
- **示例**：`{"http": "http://10.10.1.10:3128", "https": "https://10.10.1.10:1080"}`

6. revision

- **作用**：指定分词器的特定修订版，主要用于版本控制。
- **示例**：`"main"` 或某个特定的 git 哈希值。

7. use_auth_token

- **作用**：如果你需要通过认证的方式访问私有分词器或者使用额外的 API 令牌，可以使用这个参数。
- **示例**：可以是一个字符串或 `True`，如果是 `True`，则会使用 Hugging Face 库配置的 token。

8. local_files_only

- **作用**：只从本地文件系统加载分词器，不尝试从互联网下载。
- **类型**：布尔值，如 `True` 或 `False`

9. use_fast

- **作用**：是否使用分词器的快速实现（如果有的话）。快速分词器是用 Rust 编写的，提供更高的性能。
- **类型**：布尔值，如 `True` 或 `False`（默认通常是 `True`，如果提供了快速实现）

10. from_pt

- **作用**：如果原始分词器是基于 PyTorch 的，则设置此参数以确保正确加载。
- **类型**：布尔值，如 `True` 或 `False`

11. padding_side="left"

- **作用**：指定填充（padding）的方向。对于某些模型（特别是用于生成任务的模型），可能需要将填充令牌添加到序列的开始处，以保持重要的信息靠近序列的结尾。
- **示例**：`"left"` 或 `"right"`。默认值通常是 `"right"`，表示填充令牌被添加到序列的末尾。

12. add_eos_token=True

- **作用**：指示分词器在序列的末尾自动添加一个结束符（End-Of-Sequence, EOS）令牌。这在自回归语言模型中非常重要，因为 EOS 令牌标识着序列的结束，对于模型生成句子的完整性和结构性至关重要。
- **类型**：布尔值，如 `True` 或 `False`。设置为 `True` 时，分词器将自动添加 EOS 令牌。

13. add_bos_token=True

- **作用**：指示分词器在序列的开始自动添加一个起始符（Begin-Of-Sequence, BOS）令牌。这同样适用于需要明确标识序列开始的场景，如在文本生成中，确保模型从一个清晰的起点开始生成文本。
- **类型**：布尔值，如 `True` 或 `False`。设置为 `True` 时，分词器将自动添加 BOS 令牌。

14. use_fast=False

- **作用**：决定是否使用分词器的快速实现版本。快速分词器基于 Rust 编写，提供了更优的性能和更低的内存消耗，特别是在处理大规模数据时。
- **类型**：布尔值，如 `True` 或 `False`。默认情况下，如果分词器有快速实现，`use_fast` 通常设为 `True`。设为 `False` 可能是因为需要确保与 Python 版本的行为完全一致或解决快速版本的特定问题。

特殊token：

在自然语言处理中，尤其是在使用语言模型如BERT、GPT等时，通常有几种特殊的令牌用于特定目的：

- **EOS (End-Of-Sequence) Token**：用于标记序列的结束。在生成任务中，这帮助模型识别何时停止生成更多的内容。
- **PAD Token**：用于在批处理多个序列时，填充较短的序列以匹配批次中最长序列的长度，确保所有序列具有相同的长度以便并行处理。

1.1.2. 定义输入和格式化

```
def chat_qwen(device, tokenizer, model, prompt):
    messages = (
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt}
    )
    # 使用分词器的 apply_chat_template 方法将消息格式化为模型可理解的输入格式
    text = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    model_inputs = tokenizer((text), return_tensors="pt").to(device)
```

在Transformers库中,`apply_chat_template`是一个tokenizer的方法,用于将聊天格式的消息列表转换为模型可以处理的格式。这个方法特别适用于聊天或对话场景,其中消息通常以角色(如"system"、"user"、"assistant")和相应的内容组成。

这里我们先使用tokenizer.apply_chat_template构建模版，再通过model_inputs = tokenizer((text), return_tensors="pt").to(device)实现转化。这样相对灵活，并且方便我们检查模型的输入。

1.1.2.1. apply_chat_template

`apply_chat_template`方法的签名如下:

```
def apply_chat_template(
    self,
    messages,
    tokenize=True,
    separator_style="two",
    add_role_type_strings=True,
    add_generation_prompt=True,
):
```

参数解释:

- `messages`: 一个列表,包含表示聊天消息的字典。每个字典应该有两个键:"role"和"content",分别表示消息的角色和内容。
- `tokenize`: 一个布尔值,指示是否对结果进行tokenization。默认为True。
- `separator_style`: 一个字符串,指定用于分隔不同消息的分隔符样式。可选值为"one"或"two"。默认为"two"。
- `add_role_type_strings`: 一个布尔值,指示是否在每个消息前添加角色类型字符串。默认为True。
- `add_generation_prompt`: 一个布尔值,指示是否在结果末尾添加生成提示符。默认为True。

除了`apply_chat_template`,Transformers库还提供了其他一些模板方法,用于将不同格式的输入转换为模型可以处理的格式。以下是一些常见的模板方法:

1. `apply_text_template`: 用于将简单的文本输入转换为模型可以处理的格式。
2. `apply_question_answering_template`: 用于将问答格式的输入(包含问题和上下文)转换为模型可以处理的格式。

3. `apply_sequence_classification_template`: 用于将序列分类格式的输入(包含文本和标签)转换为模型可以处理的格式。
4. `apply_token_classification_template`: 用于将标记分类格式的输入(包含文本和对应的标签)转换为模型可以处理的格式。

这些模板方法可以根据不同的任务和输入格式进行选择和使用。它们提供了一种方便的方式来预处理和转换输入数据,使其与模型的预期格式相匹配。你可以根据具体的任务需求选择适当的模板方法,并将其与相应的模型和tokenizer一起使用,以实现对输入数据的预处理和转换。

我们对比一下使用apply_chat_template前后的prompt

使用前:

```
[{'role': 'system', 'content': 'You are a helpful assistant.'}, {'role': 'user',
'content': '你是谁？'}]
```

使用后:

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
你是谁？<|im_end|>
<|im_start|>assistant
```

1.1.2.2. 准备模型输入

```
model_inputs = tokenizer([text], return_tensors="pt").to(device)
```

这里将处理后的文本通过分词器转换为模型需要的输入格式，`return_tensors="pt"`表示返回PyTorch张量格式，并将输入传送到前面指定的设备上（GPU）。

当你使用 Hugging Face 的 `transformers` 库中的 `AutoTokenizer` 对文本进行处理，并调用 `return_tensors="pt"`，你实际上是在为模型准备输入数据，并且这些数据是以 PyTorch 张量的形式返回的。这行代码做了什么？

1. **分词** (`tokenizer`): 这个函数调用首先使用分词器将传入的文本（`[text]`）进行分词。分词是将连续的文本字符串转换成模型可以理解的离散单元（tokens）。
2. **转换为张量** (`return_tensors="pt"`): 分词后的数据被转换为张量（tensor），这里的 `"pt"` 表示生成的是 PyTorch 张量。这意味着分词器会输出一个包含多个键（key）的字典，其中每个键对应一种不同的输入数据类型，例如 `input_ids`、`attention_mask` 等。
3. **转移到指定设备** (`to(device)`): `.to(device)` 会将张量从 CPU 移动到指定的设备上，这里的 `device` 是一个字符串，可以是 `"cuda"`（GPU）或 `"cpu"`，这取决于你之前如何定义 `device` 变量。这一步是为了在适当的硬件上

什么是 `model_inputs.input_ids`？

- `input_ids`：这是上述字典中的一个键。`input_ids` 是一个二维的 PyTorch 张量，每一行代表输入序列的一个示例（在这个情况下只有一个示例）。这些值是分词后的词汇表索引，模型将使用这些索引来查找相应的词嵌入，从而进行进一步的处理和理解。

`input_ids` 是模型理解和生成回复的基础，因为这些索引直接对应于模型训练时使用的词汇表中的词语。这些索引构成了模型输入的主要部分，是后续所有操作（如前向传播）的基础。

这样的处理流程使得文本数据能够被模型以数学形式理解和操作，从而进行任务如问答、文本生成等的处理。

1.1.3. 使用模型生成输出

```
generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in
zip(model_inputs.input_ids, generated_ids)
]
```

使用模型的`generate`方法生成响应。`max_new_tokens=512`限制生成的最大新词数。

生成的文本以ID列表形式返回，并通过列表推导式截取生成部分。这里由于input和output都是只是1个问题，1个输出，实际上只有1对回答。需要注意的是output_ids[len(input_ids):] 这一部分，其实是因为输出时，包含了输入，因此把输入部分的长度截掉，就只剩下实际的输出。

1.1.3.1. qwen1.5的bug

在使用qwen1.5模型时，生成输出的代码为：

```
generated_ids =
self.model.generate(encoded_input.input_ids,max_new_tokens=512,eos_token_id=15164
5,pad_token_id=151645)
# 检查默认的eos_token_id
# print(self.tokenizer.eos_token_id,self.tokenizer.pad_token_id)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in
zip(encoded_input.input_ids, generated_ids)
    ]
```

此处手动设置`pad_token_id` to `eos_token_id`，原因在于qwen中可能这里面两个ID有bug，因此不设置会导致自问自答的问题

如上图所示，qwen模型输入问题后，一直自问自答，同时我们也看到模型提示

“Setting `pad_token_id` to `eos_token_id`:151643 for open-end generation.”

这实际上是结束符ID不一致的问题。

解决方案是生成模型中设置：

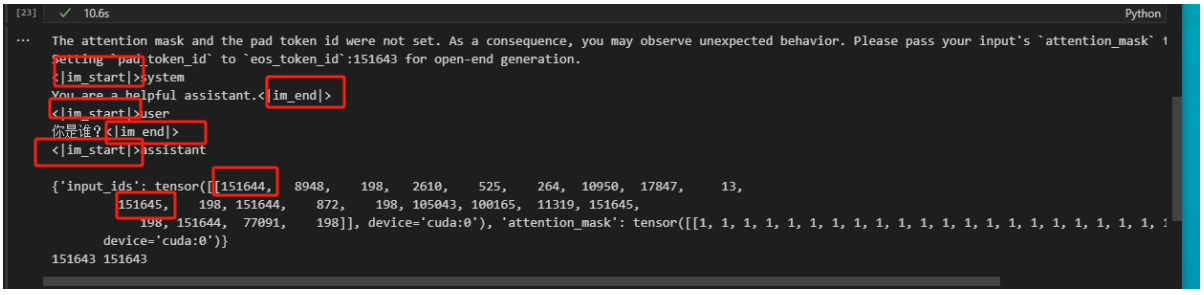
```
generated_ids =
self.model.generate(encoded_input.input_ids,max_new_tokens=512,eos_token_id=15164
5,pad_token_id=151645)
```

在不设置eos_token_id=151645的情况下，我们观察一下：

```
print(self.tokenizer.eos_token_id,self.tokenizer.pad_token_id)
```

输出的是151643

然后我们观察把问题输入后，tokenizer后的编码是什么：



可以发现起始的编码（<|im_start|>s）是151644，结束编码<|im_end|>是151645。和实际不符合。因此我们需要像1.3.1.2中一样，生成模型时，手动设置结束编码：

1.1.4. 解码输出

```
response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

最后使用分词器的 `batch_decode` 方法将ID转换回文本，并通过 `skip_special_tokens=True` 去除任何特殊符号或编码，得到最终的文本响应。