

ChatGLM两代的部署/微调/实现：从基座GLM、ChatGLM的LoRA/P-Tuning微调、6B源码解读到ChatGLM2的微调与实现

前言

随着『GPT4 多模态 /Microsoft 365 Copilot/Github Copilot X/ChatGPT插件』的推出，绝大部分公司的技术 产品 服务，以及绝大部分人的工作都将被革新一遍

- 类似iPhone的诞生 大家面向iOS编程 有了App Store
- 现在有了ChatGPT插件/GPT应用商店，以后很多公司 很多人面向GPT编程(很快技术人员分两种，一种懂GPT，一种不懂GPT)

然ChatGPT/GPT4基本不可能开源了，而通过上篇文章《

LLaMA的解读与其微调：Alpaca-LoRA/Vicuna/BELLE/中文LLaMA/姜子牙/LLaMA 2》可知，国内外各大公司、研究者推出了很多类ChatGPT开源项目，比如LLaMA、BLOOM

第一部分 国内的GLM框架与类ChatGPT项目ChatGLM-6B

1.1 GLM: General Language Model Pretraining with Autoregressive Blank Infilling

1.1.1 GLM结构：微改transformer block且通过自定义attention mask兼容GPT BERT T5三种结构

在2022年上半年，当时主流的预训练框架可以分为三种：

- autoregressive，自回归模型的代表是单向的**GPT**，本质上是一个从左到右的语言模型，常用于无条件生成任务（unconditional generation），缺点是无法利用到下文的信息
- autoencoding，自编码模型是通过某个降噪目标(如掩码语言模型，简单理解就是通过挖洞，训练模型做完形填空的能力)训练的语言编码器，如双向的**BERT**、ALBERT、RoBERTa、DeBERTa
自编码模型擅长自然语言理解任务（natural language understanding tasks），常被用来生成句子的上下文表示，缺点是不适合生成任务
- encoder-decoder，则是一个完整的Transformer结构，包含一个编码器和一个解码器，以**T5**、**BART**为代表，常用于有条件的生成任务（conditional generation）
细致来说，T5的编码器中的注意力是双向，解码器中的注意力是单向的，因此可同时应用于自然语言理解任务和生成任务。但T5为了达到和RoBERTa和DeBERTa相似的性能，往往需要更多的参数量

这三种预训练模型各自称霸一方，那么问题来了，可否结合三种预训练模型，以成天下之一统？这便是2022年5月发表的这篇论文《GLM: General Language Model Pretraining with Autoregressive Blank Infilling》的出发点，它提出了GLM架构

首先，GLM框架在整体基于Transformer基础上，做了以下三点微小改动

1. 论文中说的是，重新排列了层归一化和残差连接的顺序
we rearrange the order of layer normalization and the residual connection, which has been shown critical for large-scale language models to avoid numerical errors (Shoeybi et al., 2019)
但实际实现时，GLM用的post deepNorm，可以认为是对原始transformer用的post-norm的改进(*GPT1和原始transformer都是先self-attention再LN，或先feed forward再LN，可称为post-norm，至于GPT2和GPT3等绝大部分模型则是LN层被放置在self-attention层和feed forward层之前，可称为pre-norm*)，如下图所示(图源：A Survey of Large Language Models 第17页)

Model	Category	Size	Normalization	PE	Activation	Bias	#L	#H	d_{model}	MCL
GPT3 [55]	Causal decoder	175B	Pre LayerNorm	Learned	GeLU	✓	96	96	12288	2048
PanGU- α [75]	Causal decoder	207B	Pre LayerNorm	Learned	GeLU	✓	64	128	16384	1024
OPT [81]	Causal decoder	175B	Pre LayerNorm	Learned	ReLU	✓	96	96	12288	2048
PaLM [56]	Causal decoder	540B	Pre LayerNorm	RoPE	SwiGLU	×	118	48	18432	2048
BLOOM [69]	Causal decoder	176B	Pre LayerNorm	ALiBi	GeLU	✓	70	112	14336	2048
MT-NLG [97]	Causal decoder	530B	-	-	-	-	105	128	20480	2048
Gopher [59]	Causal decoder	280B	Pre RMSNorm	Relative	-	-	80	128	16384	2048
Chinchilla [34]	Causal decoder	70B	Pre RMSNorm	Relative	-	-	80	64	8192	-
Galactica [35]	Causal decoder	120B	Pre LayerNorm	Learned	GeLU	×	96	80	10240	2048
LaMDA [63]	Causal decoder	137B	-	Relative	GeLU	-	64	128	8192	-
Jurassic-1 [91]	Causal decoder	178B	Pre LayerNorm	Learned	GeLU	✓	76	96	13824	2048
LLaMA [57]	Causal decoder	65B	Pre RMSNorm	RoPE	SwiGLU	✓	80	64	8192	2048
GLM-130B [83]	Prefix decoder	130B	Post DeepNorm	RoPE	GeLU	✓	70	96	12288	2048
T5 [73]	Encoder-decoder	11B	Pre RMSNorm	Relative	ReLU	×	24	128	1024	512

2. 针对token的输出预测使用单一线性层
3. 用GeLU替换ReLU激活函数

此外，关于GLM的结构，这个视频也可以看下：从GLM-130B到ChatGLM：大模型预训练与微调

另，考虑到我讲的ChatGPT技术原理解析课群内，有同学对这块有疑问，所以再重点说下

- 本质上，一个GLMblock其实就是在在一个transformer block的基础上做了下结构上的微小改动而已
至于实际模型时，这个block的数量或层数可以独立设置，比如设置24层(具体见下述代码第48行) GLM/arguments.py at 4b65db165ad323e28f91129a0ec053228d10566 · THUOM/GLM · GitHub

```
group.add_argument('--num-layers', type=int, default=24,
```

- 比如，基于GLM框架的类ChatGPT开源项目「ChatGLM」使用了28个GLMBlock，类似gpt2 用的12-48个decoder-transformer block，BERT用的12-24个encoder-transformer block

```
[12]: model
[12]: ChatGLMForConditionalGeneration(
  (transformer): ChatGLMModel(
    (word_embeddings): Embedding(130528, 4096)
    (layers): ModuleList(
      (0-27): 28 x GLMBlock(
        (input_layernorm): LayerNorm((4096,), eps=1e-05, elementwise_affine=True)
        (attention): SelfAttention(
          (rotary_emb): RotaryEmbedding()
          (query_key_value): Linear(in_features=4096, out_features=12288, bias=True)
          (dense): Linear(in_features=4096, out_features=4096, bias=True)
        )
        (post_attention_layernorm): LayerNorm((4096,), eps=1e-05, elementwise_affine=True)
        (mlp): GLU(
          (dense_h_to_4h): Linear(in_features=4096, out_features=16384, bias=True)
          (dense_4h_to_h): Linear(in_features=16384, out_features=4096, bias=True)
        )
      )
    )
    (final_layernorm): LayerNorm((4096,), eps=1e-05, elementwise_affine=True)
  )
  (lm_head): Linear(in_features=4096, out_features=130528, bias=False)
)
```

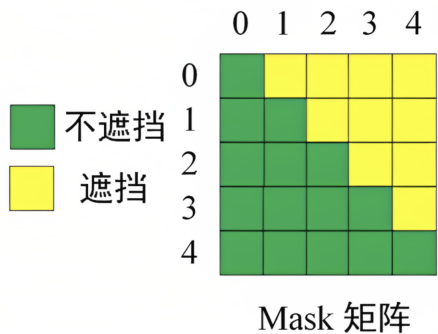
- 有些文章 包括我那篇Transformer笔记，为举例，使用的N=6的示例，相当于编码器模块 用的6个encoder-transformer block，解码器模块 也用的6个decoder-transformer block

其次，考虑到三类预训练模型的训练目标

- GPT的训练目标是从左到右的文本生成
- BERT的训练目标是对文本进行随机掩码，然后预测被掩码的词
- T5则是接受一段文本，从左到右的生成另一段文本

为了大一统，我们必须在结构和训练目标上兼容这三种预训练模型。如何实现呢？文章给出的解决方法是结构上，只需要GLM中同时存在单向注意力和双向注意力即可因为在原本的Transformer模型中，这两种注意力机制是通过修改attention mask实现的

- 当attention_mask是全1矩阵的时候，这时注意力是双向的
- 当attention_mask是三角矩阵的时候（如下图），注意力就是单向

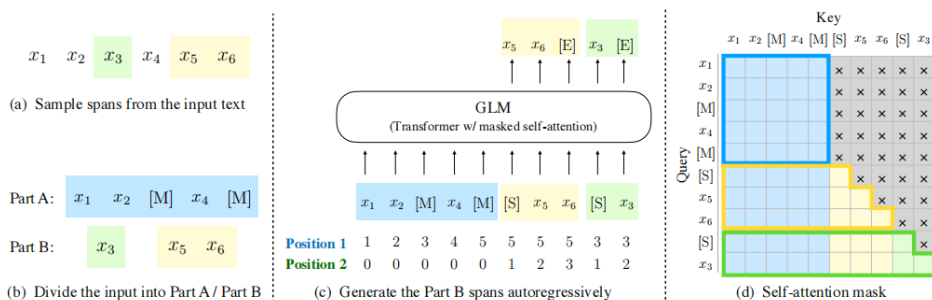


类似地，GLM可以在只使用Transformer编码器的情况下，自定义attention mask来兼容三种模型结构，使得

- 前半部分互相之间能看到，等效于编码器(BERT)的效果，侧重于信息提炼
- 后半部分只能看到自身之前的，等效于解码器(GPT)的效果，侧重于生成

这样综合起来实现的效果就是，将提炼信息作为条件，进行有条件地生成(有条件生成就是编解码模型)

其实，所谓编码解码，其本质就是mask的遮盖设计。举个例子，假设原始的文本序列为 $x_1, x_2, x_3, x_4, x_5, x_6$ ，采样的两个文本片段为 x_3 和 x_5, x_6 ，那么掩码后的文本序列为 $x_1, x_2, [M], x_4, [M]$ （以下简称Part A），如上图所示，拆解图中的三块分别可得



- 我们要根据第一个 $[M]$ 解码出 x_3 ，根据第二个 $[M]$ 依次解码出 x_5, x_6 ，那怎么从 $[M]$ 处解码出变长的序列吗？这就需要用到开始标记 $[S]$ 和结束标记 $[E]$ 了
- 我们从开始标记 $[S]$ 开始依次解码出被掩码的文本片段，直至结束标记 $[E]$ 。通过本博客内的Transformer笔记可知，Transformer中的位置信息是通过位置向量来记录的

在GLM中，位置向量有两个，一个 用来记录Part A中的相对顺序，一个 用来记录被掩码的文本片段（简称为Part B）中的相对顺序

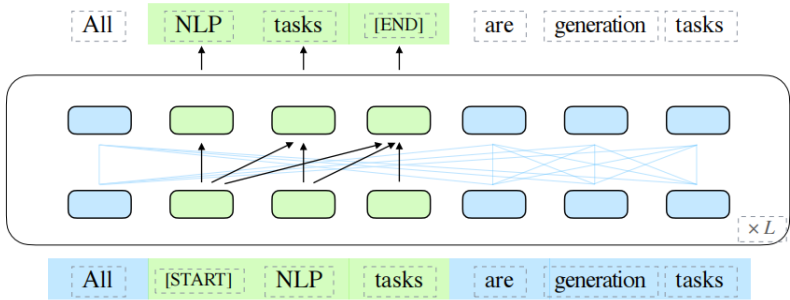
- 此外，还需要通过自定义自注意掩码(attention mask)来达到以下目的：
 - 双向编码器Part A中的词彼此可见，即图(d)中蓝色框中的区域
 - 单向解码器Part B中的词单向可见，即图(d)黄色框的区域
 - Part B可见Part A
 - 其余不可见，即图(d)中灰色的区域

需要说明的是，Part B包含所有被掩码的文本片段，但是文本片段的相对顺序是随机打乱的

1.1.2 GLM的预训练和微调

训练目标上，GLM论文提出一个自回归空格填充的任务(Autoregressive Blank Infilling)，来兼容三种预训练目标

自回归填充有些类似掩码语言模型，首先采样输入文本中部分片段，将其替换为[MASK]标记，然后预测[MASK]所对应的文本片段，与掩码语言模型不同的是，预测的过程是采用自回归的方式



具体来说

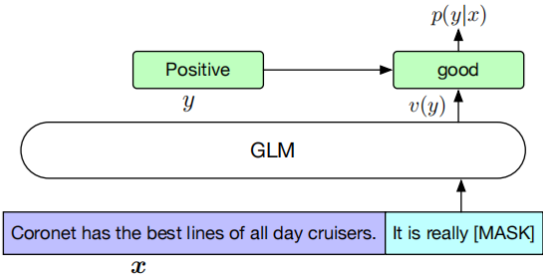
- 当被掩码的片段长度为1的时候，空格填充任务等价于掩码语言建模，类似BERT
- 当将文本1和文本2拼接在一起，然后将文本2整体掩码掉，空格填充任务就等价于条件语言生成任务，类似T5/BART
- 当全部的文本都被掩码时，空格填充任务就等价于无条件语言生成任务，类似GPT

最终，作者使用了两个预训练目标来优化GLM，两个目标交替进行：

- 文档级别的预测/生成：从文档中随机采样一个文本片段进行掩码，片段的长度为文档长度的50%-100%
- 句子级别的预测/生成：从文档中随机掩码若干文本片段，每个文本片段必须为完整的句子，被掩码的词数量为整个文档长度的15%

尽管GLM是BERT、GPT、T5三者的结合，但是在预训练时，为了适应预训练的目标，作者还是选择掩码较长的文本片段，以确保GLM的文本生成能力，并在微调的时候将自然语言理解任务也转化为生成任务，如情感分类任务转化为填充空白的任务

输入：{Sentence}, prompt: It is really $[M]$ ，对应的标签为good和bad



1.2 GLM-130B：国内为数不多的可比肩GPT3的大模型之一

2022年8月，清华背景的智谱AI基于GLM框架，正式推出拥有1300亿参数的中英双语稠密模型 GLM-130B(论文地址、代码地址，论文解读之一， GLM-130B is trained on a cluster of 96 DGX-A100 GPU (8×40G) servers with a 60-day，可以较好的支持2048个token的上下文窗口)

其在一些任务上的表现优于GPT3-175B，是国内与2020年5月的GPT3在综合能力上差不多的模型之一(即便放到23年年初也并不多)，这是它的一些重要特点

	基础架构	训练方式	量化	加速	跨平台能力
GPT3-175B	GPT	自监督预训练	-	-	NVIDIA
BLOOM-176B	GPT	自监督预训练	INT8	Megatron	NVIDIA
GLM-130B	GLM	自监督预训练 多任务预训练	INT8/INT4	Faster Transformer	NVIDIA 海光 DCU 昇腾910 申威
对比优势	高精度: · Big-bench-lite: +5.2% · LAMBADA: +2.3% · CLUE: +24.3% · FewCLUE: +12.8%		普惠推理: 节省75%内存 可单台3090 (4) 或单台2080 (8) 进行无损推理	高速推理: 比Pytorch 提速7-8.4倍 Megatron 提速2.5倍	跨平台: 支持更多不同的 大规模语言模型 的适配和应用

1.3 ChatGLM-6B的训练框架与部署步骤

1.3.1 ChatGLM-6B的训练框架

ChatGLM-6B(介绍页面、代码地址), 是智谱 AI 开源、支持中英双语的对话语言模型, 其

- 基于General Language Model(GLM)架构, 具有62亿参数, 无量化下占用显存13G
INT8量化级别下支持在单张11G显存的 2080Ti 上进行推理使用(因为INT8下占用显存8G)
而INT4量化级别下部署的话最低只需 6GB显存(另基于 P-Tuning v2 的高效参数微调方法的话, 在INT4 下最低只需 7GB 显存即可启动微调)

量化等级	最低 GPU 显存（部署/推理）	最低 GPU 显存（高效参数微调）
FP16（无量化）	13 GB	14 GB
INT8	8 GB	9 GB
INT4	6 GB	7 GB

这里需要解释下的是, INT8量化是一种将深度学习模型中的权重和激活值从32位浮点数（FP32）减少到8位整数（INT8）的技术。这种技术可以降低模型的内存占用和计算复杂度, 从而减少计算资源需求, 提高推理速度, 同时降低能耗

量化的过程通常包括以下几个步骤:

- 量化范围选择: 确定权重和激活值的最小值和最大值
- 量化映射: 根据范围将32位浮点数映射到8位整数
- 反量化: 将8位整数转换回浮点数, 用于计算

- ChatGLM-6B参考了 ChatGPT 的训练思路, 在千亿基座模型GLM-130B中注入了代码预训练, 通过监督微调(Supervised Fine-Tuning)、反馈自助(Feedback Bootstrap)、人类反馈强化学习(Reinforcement Learning from Human Feedback)等方式等技术实现人类意图对齐, 并针对中文问答和对话进行优化
- 最终经过约 1T 标识符的中英双语训练, 生成符合人类偏好的回答

虽尚有很多不足(比如因为6B的大小限制, 导致模型的记忆能力、编码、推理能力皆有限), 但在6B这个参数量级下不错了, 部署也非常简单, 我七月在线的同事朝阳花了一两个小时即部署好了(主要时间花在模型下载上, 实际的部署操作很快)



1.3.2 ChatGLM-6B的部署步骤

以下是具体的部署过程

1. 硬件配置

本次实验用的七月的GPU服务器(专门为七月集/高/论文/VIP学员配置的), 显存大小为16G的P100, 具体配置如下:
CPU&内存: 28核(vCPU)112 GB
操作系统: Ubuntu_64
GPU: NVIDIA Tesla P100
显存: 16G

2. 配置环境

建议最好自己新建一个conda环境
pip install -r requirements.txt
(ChatGLM-6B/requirements.txt at main · THUDM/ChatGLM-6B · GitHub)

特别注意torch版本不低于1.10（这里安装的1.10），transformers为4.27.1

torch的安装命令可以参考pytorch官网：<https://pytorch.org/>

这里使用的pip命令安装的，命令如下

```
pip install torch==1.10.0+cu102 torchvision==0.11.0+cu102 torchaudio==0
```









3. 下载项目仓库

```
git clone https://github.com/THUDM/ChatGLM-6B
```






```
cd ChatGLM-6B
```





4. 下载ChatGLM-6B模型文件

具体而言，较大的8个模型文件可以从这里下载（下载速度快）：[清华大学云盘](#)

	pytorch_model-00001-of-00008.bin	1.9 GB
	pytorch_model-00002-of-00008.bin	1.9 GB
	pytorch_model-00003-of-00008.bin	2.0 GB
	pytorch_model-00004-of-00008.bin	1.9 GB
	pytorch_model-00005-of-00008.bin	1.9 GB
	pytorch_model-00006-of-00008.bin	1.9 GB
	pytorch_model-00007-of-00008.bin	1.1 GB
	pytorch_model-00008-of-00008.bin	1.2 GB

其他的小文件可以从这里下载（点击红框的下载按钮即可）：[THUDM/chatglm-6b · Hugging Face](#)

	config.json	722 Bytes ↓	Add pad_token_id in config.
	configuration_chatglm.py	4.12 kB ↓	Add support for loading quantized
	ice_text.model	2.7 MB  ↓	add ice_text.model
	modeling_chatglm.py	56.5 kB ↓	Fix position ids expand

	pytorch_model.bin.index...	33.4 kB ↓	Add chatglm-6b
	quantization.py	15.1 kB ↓	Add support for loading quantized
	tokenization_chatglm.py	17.7 kB ↓	Fix generate
	tokenizer_config.json	416 Bytes ↓	Add chatglm-6b

注意这里都下载在了/data/chatglm-6b下，在后面执行代码的时候需要将文件中的模型文件路径改为自己的

5. 推理与部署

可运行的方式有多种

→ 如果在本地运行，可以直接执行代码，或者使用命令行方式运行

→ 如果想让别人公网访问，可以用下面两种方式：一种是基于Gradio，一种是基于streamlit

特别注意：运行代码前请检查模型文件路径是否正确，这里均改为了/data/chatglm-6b

代码运行demo

```
1 from transformers import AutoTokenizer, AutoModel
2 tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm-6b", trust_remote_code=True)
3 model = AutoModel.from_pretrained("/data/chatglm-6b", trust_remote_code=True).half().cuda()
4 model = model.eval()
5 response, history = model.chat(tokenizer, "你好", history=[])
6 print(response)
7 response, history = model.chat(tokenizer, "晚上睡不着应该怎么办", history=history)
8 print(response)
```

命令行 Demo

运行仓库中 cli_demo.py:

```
python cli_demo.py
```

程序会在命令行中进行交互式的对话，在命令行中输入指示并回车即可生成回复，输入 clear 可以清空对话历史，输入 stop 终止程序

基于Gradio的网页版demo

运行web_demo.py即可（注意可以设置share=True，便于公网访问）：python web_demo.py（注意运行前确认下模型文件路径）

用户：晚上睡不着怎么办？

ChatGLM-6B：以下方法是一些有助于在晚上入睡的方法：

1. 建立规律的睡眠时间表：每天在相同的时间上床和起床可以帮助身体调整自己的内部时钟，更容易入睡。
 2. 创造舒适的睡眠环境：保持房间安静、黑暗、凉爽，使用舒适的床垫和枕头，这些都有助于创造一个舒适的睡眠环境。
 3. 放松身心：使用深呼吸、渐进性肌肉松弛等放松技巧可以帮助放松身心，减轻压力和焦虑，从而更容易入睡。
 4. 避免刺激性食物和饮料：例如咖啡、茶、可乐等刺激性饮料和食物，应该在睡前几个小时内避免食用。
 5. 练习放松技巧：例如瑜伽或冥想等放松技巧，可以帮助放松身心，减轻压力和焦虑，从而更容易入睡。
 6. 避免躺在床上翻来覆去：如果躺在床上超过20分钟还不能入睡，不要继续躺在床上，而是起床去做一些放松的活动，例如阅读或听轻柔的音乐，直到感到困倦为止。
- 如果这些方法都没有帮助入睡，可以考虑咨询医生或睡眠专家，了解更多可能需要采取的治疗措施。

晚上睡不着怎么办？

Maximum length

2048

Top P

0.7

Temperature

0.95

Generate

基于streamlit网页版 Demo

```
pip install streamlit
pip install streamlit-chat
streamlit run web_demo2.py --server.port 6006（可以将6006端口放出，便于公网访问）
```

默认情况下，模型以 FP16 精度加载，运行上述代码需要大概 13GB 显存。如果显存有限，还可以考虑模型量化，目前支持4/8 bit 量化

此外，据介绍，GLM团队正在内测130B参数的ChatGLM，相信从6B到130B，效果应该能提升很多

1.4 微调ChatGLM-6B：针对各种数据集通过LoRA或P-Tuning v2

1.4.1 通过Stanford Alpaca的52K数据集基于LoRA(PEFT库)微调ChatGLM-6B

从上文可知，Stanford Alpaca的52K数据集是通过Self Instruct方式提示GPT3对应的API产生的指令数据，然后通过这批指令数据微调Meta的LLaMA 7B

而GitHub上的这个微调ChatGLM-6B项目(作者：mymusise)，则基于Stanford Alpaca的52K数据集通过LoRA(low-rank adaptation)的方式微调ChatGLM-6B

如上一篇文章所说，Huggingface公司推出的PEFT(Parameter-Efficient Fine-Tuning)库便封装了LoRA这个方法，具体而言，通过PEFT-LoRA微调ChatGLM-6B的具体步骤如下

- 第一步，配置环境与准备
先下载项目仓库
git clone https://github.com/mymusise/ChatGLM-Tuning.git

```
创建一个python3.8的环境
conda create -n torch1.13 python==3.8
conda activate torch1.13
```

```
根据requirements.txt配置环境
pip install bitsandbytes==0.37.1
```

```
安装1.13, cuda11.6（torch官网命令）
pip install torch==1.13.1+cu116 torchvision==0.14.1+cu116 torchaudio==0.13.1 --extra-index-url https://download.pytorch.org/whl/cu116
```

安装其他的包

```
1 | pip install accelerate==0.17.1
2 | pip install tensorboard==2.10
3 | pip install protobuf==3.19.5
4 | pip install transformers==4.27.1
5 | pip install icetk
6 | pip install cpm_kernels==1.0.11
7 | pip install datasets==2.10.1
8 | pip install git+https://github.com/huggingface/peft.git # 最新版本 >=0.3.0.dev0
```

遇到冲突问题：icetk 0.0.5 has requirement protobuf<3.19, but you have protobuf 3.19.5.
最后装了3.18.3的protobuf，发现没有问题

模型文件准备
模型文件在前面基于ChatGLM-6B的部署中已经准备好了，注意路径修改正确即可

- 第二步，数据准备
项目中提供了数据，数据来源为 Stanford Alpaca 项目的用于微调模型的52K数据，数据生成过程可详见：https://github.com/tatsu-lab/stanford_alpaca#data-release
alpaca_data.json，包含用于微调羊驼模型的 52K 指令数据，这个 JSON 文件是一个字典列表，每个字典包含以下字段：

instruction: str, 描述了模型应该执行的任务, 52K 条指令中的每一条都是唯一的

input: str, 任务的可选上下文或输入。例如, 当指令是“总结以下文章”时, 输入就是文章, 大约 40% 的示例有输入

output: str, 由 *text-davinci-003* 生成的指令的答案

示例如下:

```
1 | [  
2 |   {  
3 |     "instruction": "Give three tips for staying healthy.",  
4 |     "input": "",  
5 |     "output": "1.Eat a balanced diet and make sure to include plenty of fruits and vegetables. \n2. Exercise regularly to keep y  
6 |   },  
7 |   {  
8 |     "instruction": "What are the three primary colors?",  
9 |     "input": "",  
10 |    "output": "The three primary colors are red, blue, and yellow."  
11 |  },  
12 |  ...  
13 | ]
```

• 第三步, 数据处理

运行 `cover_alpaca2jsonl.py` 文件

`python cover_alpaca2jsonl.py \ --data_path data/alpaca_data.jsonl \ --save_path data/alpaca_data.jsonl \`

处理后的文件示例如下:

```
1 | {  
2 |   "text": "### Instruction:\nGive three tips for staying healthy.\n\n### Response:\n1.Eat a balanced diet and make sure to include pl  
3 |   "text": "### Instruction:\nWhat are the three primary colors?\n\n### Response:\nThe three primary colors are red, blue, and yellow.
```

运行 `tokenize_dataset_rows.py` 文件, 注意: 修改`tokenize_dataset_rows`中的`model_name`为自己的文件路径: `/data/chatglm-6b`

```
1 | python tokenize_dataset_rows.py \  
2 |   --jsonl_path data/alpaca_data.jsonl \  
3 |   --save_path data/alpaca \  
4 |   --max_seq_length 200 \  
5 |   --skip_overlength \  
6 |
```

• 第四步, 微调过程

注意: 运行前修改下`finetune.py` 文件中模型路径: `/data/chatglm-6b`

```
1 | python finetune.py \  
2 |   --dataset_path data/alpaca \  
3 |   --lora_rank 8 \  
4 |   --per_device_train_batch_size 6 \  
5 |   --gradient_accumulation_steps 1 \  
6 |   --max_steps 52000 \  
7 |   --save_steps 1000 \  
8 |   --save_total_limit 2 \  
9 |   --learning_rate 1e-4 \  
10 |  --fp16 \  
11 |  --remove_unused_columns false \  
12 |  --logging_steps 50 \  
13 |  --output_dir output;
```

这个`finetune`长啥样呢?

my musise / ChatGLM-Tuning

<> Code Issues 169 Pull requests 8 Actions Projects Security Insights

Code

master

Go to file

- > data
- > examples
- > output
- .gitignore
- LICENSE
- README.md
- cover_alpaca2jsonl.py
- finetune.py
- infer.ipynb
- requirements.txt
- tokenize_dataset_rows.py

ChatGLM-Tuning / finetune.py

my musise It looks like the training issue with the official code has been fixe...

Code Blame 118 lines (98 loc) · 3.54 KB

```
1 from transformers.integrations import TensorBoardCallback
2 from torch.utils.tensorboard import SummaryWriter
3 from transformers import TrainingArguments
4 from transformers import Trainer, HfArgumentParser
5 from transformers import AutoTokenizer, AutoModel
6 import torch
7 import torch.nn as nn
8 from peft import get_peft_model, LoraConfig, TaskType
9 from dataclasses import dataclass, field
10 import datasets
11 import os
12
13
14 tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm-6b", trust_remote_code=True)
15
16
```

其对应的完整代码为

```
1 # 导入所需的库和模块
2 from transformers.integrations import TensorBoardCallback
3 from torch.utils.tensorboard import SummaryWriter
4 from transformers import TrainingArguments
5 from transformers import Trainer, HfArgumentParser
6 from transformers import AutoTokenizer, AutoModel
7 import torch
8 import torch.nn as nn
9 from peft import get_peft_model, LoraConfig, TaskType
10 from dataclasses import dataclass, field
11 import datasets
12 import os
13
14 # 从预训练模型加载tokenizer
15 tokenizer = AutoTokenizer.from_pretrained("THUDM/chatglm-6b", trust_remote_code=True)
16
17 # 定义FinetuneArguments数据类, 用于存储微调的参数
18 @dataclass
19 class FinetuneArguments:
20     dataset_path: str = field(default="data/alpaca") # 数据集路径
21     model_path: str = field(default="output") # 模型保存路径
22     lora_rank: int = field(default=8) # Lora排名, 用于peft模型的设置
23
24 # 自定义CastOutputToFloat类, 继承自nn.Sequential, 用于将输出转换为float32类型
25 class CastOutputToFloat(nn.Sequential):
26     def forward(self, x):
27         return super().forward(x).to(torch.float32)
28
29 # 数据处理函数data_collator, 用于将输入数据按照最长序列长度进行padding
30 def data_collator(features: list) -> dict:
31     len_ids = [len(feature["input_ids"]) for feature in features]
32     longest = max(len_ids)
33     input_ids = []
34     labels_list = []
35     for ids_l, feature in sorted(zip(len_ids, features), key=lambda x: -x[0]):
36         ids = feature["input_ids"]
37         seq_len = feature["seq_len"]
38         labels = (
39             [-100] * (seq_len - 1) + ids[(seq_len - 1) :] + [-100] * (longest - ids_l)
40         )
41         ids = ids + [tokenizer.pad_token_id] * (longest - ids_l)
42         _ids = torch.LongTensor(ids)
43         labels_list.append(torch.LongTensor(labels))
44         input_ids.append(_ids)
45     input_ids = torch.stack(input_ids)
46     labels = torch.stack(labels_list)
47     return {
48         "input_ids": input_ids,
49         "labels": labels,
50     }
51
```



```

52 # 自定义ModifiedTrainer类, 继承自Trainer, 用于微调训练, 并对模型保存进行了自定义 53 | class ModifiedTrainer(Trainer):
54     def compute_loss(self, model, inputs, return_outputs=False):
55         return model(
56             input_ids=inputs["input_ids"],
57             labels=inputs["labels"],
58         ).loss
59
60     def save_model(self, output_dir=None, _internal_call=False):
61         from transformers.trainer import TRAINING_ARGS_NAME
62
63         os.makedirs(output_dir, exist_ok=True)
64         torch.save(self.args, os.path.join(output_dir, TRAINING_ARGS_NAME))
65         saved_params = {
66             k: v.to("cpu") for k, v in self.model.named_parameters() if v.requires_grad
67         }
68         torch.save(saved_params, os.path.join(output_dir, "adapter_model.bin"))
69
70 # 主函数main()
71 def main():
72     # 创建TensorBoard的SummaryWriter, 用于记录训练过程的日志
73     writer = SummaryWriter()
74
75     # 使用HfArgumentParser解析命令行参数并存储为FinetuneArguments和TrainingArguments两个数据类的实例
76     finetune_args, training_args = HfArgumentParser(
77         (FinetuneArguments, TrainingArguments)
78     ).parse_args_into_dataclasses()
79
80     # 初始化模型, 从预训练模型加载微调模型
81     model = AutoModel.from_pretrained(
82         "THUDM/chatglm-6b", load_in_8bit=True, trust_remote_code=True, device_map="auto"
83     )
84     model.gradient_checkpointing_enable() # 开启梯度检查点
85     model.enable_input_require_grads() # 开启输入的梯度计算
86     model.is_parallelizable = True # 模型可并行计算
87     model.model_parallel = True # 使用模型并行计算
88     model.lm_head = CastOutputToFloat(model.lm_head) # 将输出转换为float32类型
89     model.config.use_cache = (
90         False # 关闭缓存以减少内存占用, 但在推断时需要重新开启
91     )
92
93     # 设置peft模型, 设置LoraConfig, 用于构造peft模型
94     peft_config = LoraConfig(
95         task_type=TaskType.CAUSAL_LM,
96         inference_mode=False,
97         r=finetune_args.lora_rank,
98         lora_alpha=32,
99         lora_dropout=0.1,
100     )
101     # 加载peft模型
102     model = get_peft_model(model, peft_config)
103
104     # 从磁盘加载数据集
105     dataset = datasets.load_from_disk(finetune_args.dataset_path) print(f"\n{len(dataset)=}\n") # 打印数据集的样本数量
106
107     # 开始训练
108     trainer = ModifiedTrainer(
109         model=model,
110         train_dataset=dataset,
111         args=training_args,
112         callbacks=[TensorBoardCallback(writer)], # 添加TensorBoard的回调函数, 用于记录训练过程的日志
113         data_collator=data_collator,
114     )
115     trainer.train() # 执行训练
116     writer.close() # 关闭TensorBoard的SummaryWriter
117     # 保存模型
118     model.save_pretrained(training_args.output_dir) # 保存微调后的模型
119
120 # 程序入口
121 if __name__ == "__main__":
122     main() # 调用主函数main()

```

如遇Nvidia驱动报错 (如没有可忽略)

说明Nvidia驱动太老, 需要更新驱动

UserWarning: CUDA initialization: The NVIDIA driver on your system is too old (found version 10020). Please update your GPU driver by downloading and installing a new version from the URL: <http://www.nvidia.com/Download/index.aspx> Alternatively, go to: <https://pytorch.org> to install a PyTorch version that has been compiled with your version of the CUDA driver. (Triggered internally at ./c10/cuda/CUDAFuncions.cpp:109.)

解决：更新驱动即可，参考：Ubuntu 18.04 安装 NVIDIA 显卡驱动 - 知乎

BUG REPORT报错

参考：因为peft原因，cuda10.2报错 · Issue #108 · mymusise/ChatGLM-Tuning · GitHub

CUDA SETUP: CUDA version lower than 11 are currently not supported for LLM.int8()

考虑安装11以上的cudatoolkit，参考下面链接，安装cudatoolkit11.3（因为Ubuntu系统版本的原因，不能装11.6的）

Ubuntu16.04 安装cuda11.3+cudnn8.2.1 - 知乎

cudatoolkit下载地址：

CUDA Toolkit 11.3 Downloads | NVIDIA 开发者

运行代码前先执行下面命令：

```
1 export LD_LIBRARY_PATH=/usr/local/cuda-11.3/lib64:$LD_LIBRARY_PATH
2 export CUDA_HOME=/usr/local/cuda-11.3:$CUDA_HOME
3 export PATH=/usr/local/cuda-11.3/bin:$PATH
```

内存不够，考虑将per_device_train_batch_size设为1

```
1 python finetune.py \
2     --dataset_path data/alpaca \
3     --lora_rank 8 \
4     --per_device_train_batch_size 1 \
5     --gradient_accumulation_steps 1 \
6     --max_steps 52000 \
7     --save_steps 1000 \
8     --save_total_limit 2 \
9     --learning_rate 1e-4 \
10    --fp16 \
11    --remove_unused_columns false \
12    --logging_steps 50 \
13    --output_dir output;
```

报错：RuntimeError: expected scalar type Half but found Float

<https://github.com/mymusise/ChatGLM-Tuning/issues?q=is%3Aissue+is%3Aopen+RuntimeError%3A+expected+scalar+type+Half+but+found+Float>

解决方法：

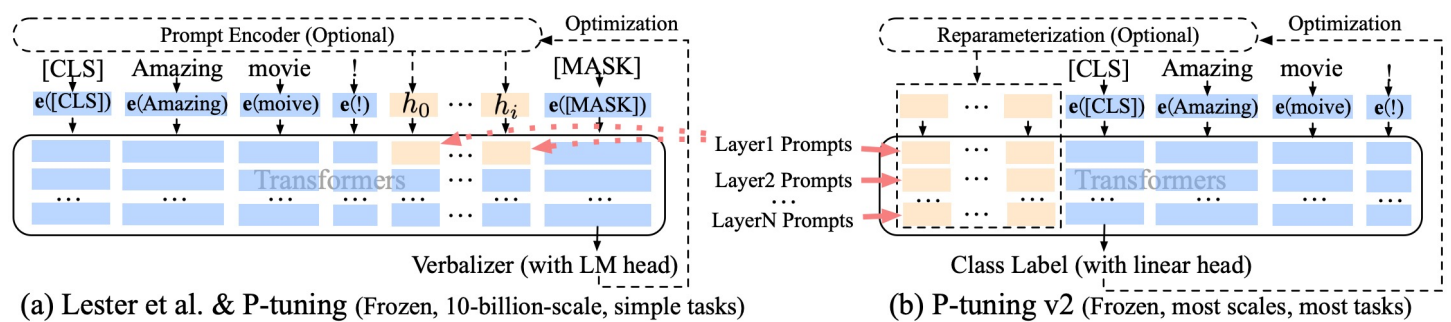
一种是，不启用fp16，load_in_8bit设为True，可以运行，但loss为0；

一种是，启用fp16，load_in_8bit设为False，不行，应该还是显存不够的问题，至少需要24G左右的显存

1.4.2 ChatGLM团队：通过ADGEN数据集基于P-Tuning v2微调ChatGLM-6B

此外，ChatGLM团队自身也出了一个基于P-Tuning v2的方式微调ChatGLM-6B的项目：[ChatGLM-6B 模型基于 P-Tuning v2 的微调](#)

P-Tuning v2(代码地址，论文地址)意义在于：将需要微调的参数数量减少到原来的 0.1%，再通过模型量化、Gradient Checkpoint 等方法，最低只需要 7GB 显存即可运行



那具体怎么通过P-Tuning v2微调ChatGLM-6B呢，具体步骤如下：

• 第一步，配置环境与准备

地址：[ChatGLM-6B/ptuning at main · THUDM/ChatGLM-6B · GitHub](#)

安装以下包即可，这里直接在torch1.13的conda环境下安装的

```
pip install rouge_chinese nltk jieba datasets
```

• 第二步，模型文件准备

模型文件在前面基于ChatGLM-6B的部署中已经准备好了，注意路径修改正确即可

特别注意：如果你是之前下载的可能会报错，下面有详细的错误及说明

• 第三步，数据准备

ADGEN数据集的任务为根据输入（content）生成一段广告词（summary）

```
{
```

```

"content": "类型#上衣*版型#宽松*版型#显瘦*图案#线条*衣样式#衬衫*衣袖型#泡泡袖*衣款式#抽绳",
"summary": "这件衬衫的款式非常的宽松，利落的线条可以很好的隐藏身材上的小缺点，穿在身上有着很好的显瘦效果。领口装饰了一个可爱的抽绳，漂亮的绳结展
现出了十足的个性，配合时尚的泡泡袖型，尽显女性甜美可爱的气息。"
}

```

从[Google Drive](#) 或者 [Tsinghua Cloud](#) 下载处理好的 ADGEN数据集，将解压后的AdvertiseGen目录放到本 ptuning 目录下即可

• 第四步，微调过程

修改train.sh文件

去掉最后的 --quantization_bit 4

注意修改模型路径，THUDM/chatglm-6b修改为/data/chatglm-6b

如果你也是在云服务器上运行，建议可以加上nohup后台命令，以免断网引起训练中断的情况修改后train.sh文件如下：

```

1 PRE_SEQ_LEN=8
2 LR=1e-2
3
4 CUDA_VISIBLE_DEVICES=0 nohup python -u main.py \
5     --do_train \
6     --train_file AdvertiseGen/train.json \
7     --validation_file AdvertiseGen/dev.json \
8     --prompt_column content \
9     --response_column summary \
10    --overwrite_cache \
11    --model_name_or_path /data/chatglm-6b \
12    --output_dir output/adgen-chatglm-6b-pt-$PRE_SEQ_LEN-$LR \
13    --overwrite_output_dir \
14    --max_source_length 64 \
15    --max_target_length 64 \
16    --per_device_train_batch_size 1 \
17    --per_device_eval_batch_size 1 \
18    --gradient_accumulation_steps 16 \
19    --predict_with_generate \
20    --max_steps 3000 \
21    --logging_steps 10 \
22    --save_steps 1000 \
23    --learning_rate $LR \
24    --pre_seq_len $PRE_SEQ_LEN \
25    >> log.out 2>&1 &

```



执行命令，开始微调

bash train.sh

如果报错：'ChatGLMModel' object has no attribute 'prefix_encoder'（如没有可忽略）

解决方案：需要更新 THUDM/chatglm-6b at main 里面的几个py文件(重新下载下这几个文件就可以了)

tokenization_chatglm.py

quantization.py

modeling_chatglm.py

config.json

configuration_chatglm.py

微调过程占用大约13G的显存

NVIDIA-SMI 510.108.03 Driver Version: 510.108.03 CUDA Version: 11.6									
GPU	Name	Persistence-M		Bus-Id	Disp.A	Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	MIG M.	
0	Tesla P100-PCIE...	Off	00000000:00:0A.0	Off					
N/A	63C	P0	178W / 250W		12993MiB / 16384MiB	89%	Default	N/A	
Processes:									
GPU	GI	CI	PID	Type	Process name	GPU Memory			
	ID	ID				Usage			
0	N/A	N/A	4663	C	python3	12991MiB			

微调过程loss变化情况

```
{'loss': 7.595, 'learning_rate': 0.009966666666666667, 'epoch': 0.0}
{'loss': 7.7457, 'learning_rate': 0.009933333333333334, 'epoch': 0.0}
{'loss': 7.4689, 'learning_rate': 0.0099, 'epoch': 0.0}
{'loss': 7.365, 'learning_rate': 0.009866666666666668, 'epoch': 0.01}
{'loss': 6.9716, 'learning_rate': 0.009833333333333333, 'epoch': 0.01}
{'loss': 6.7287, 'learning_rate': 0.0098, 'epoch': 0.01}
{'loss': 6.6212, 'learning_rate': 0.009766666666666667, 'epoch': 0.01}
{'loss': 6.3539, 'learning_rate': 0.009733333333333333, 'epoch': 0.01}
{'loss': 6.2145, 'learning_rate': 0.0097, 'epoch': 0.01}
{'loss': 6.1978, 'learning_rate': 0.009666666666666667, 'epoch': 0.01}
{'loss': 6.0319, 'learning_rate': 0.009633333333333334, 'epoch': 0.02}
{'loss': 5.6662, 'learning_rate': 0.0096, 'epoch': 0.02}
{'loss': 5.1955, 'learning_rate': 0.009566666666666666, 'epoch': 0.02}
4%|██████████| 130/3000 [28:20<10:20:27, 12.97s/it]
```

微调完成后，output/adgen-chatglm-6b-pt-8-1e-2路径下会生成对应的模型文件，如下（这里生成了3个）：

checkpoint-1000 checkpoint-2000 checkpoint-3000 runs

• 第五步，推理过程

只需要在加载模型的位置修改成微调后的路径即可

将 evaluate.sh 中的 CHECKPOINT 更改为训练时保存的 checkpoint 名称，运行以下指令进行模型推理和评测：

改这一行即可：--model_name_or_path ./output/\$CHECKPOINT/checkpoint-3000

bash evaluate.sh

```
100%|██████████| 1069/1070 [1:01:19<00:03, 3.35s/it]
[INFO|configuration_utils.py:575] 2023-04-05 15:35:06,224 >> Generate config GenerationConfig {
  "from_model_config": true,
  "bos_token_id": 150004,
  "eos_token_id": 150005,
  "pad_token_id": 20003,
  "transformers_version": "4.27.1"
}
```

评测指标为中文 Rouge score 和 BLEU-4，生成的结果保存在

./output/adgen-chatglm-6b-pt-8-1e-2/generated_predictions.txt

```
100%|██████████| 1070/1070 [1:01:29<00:00, 3.45s/it]
***** predict metrics *****
predict_bleu-4          =      5.8412
predict_rouge-1         =     30.5592
predict_rouge-2         =      6.7782
predict_rouge-l         =     24.7899
predict_runtime         =    1:01:33.28
predict_samples         =      1070
predict_samples_per_second =      0.29
predict_steps_per_second  =      0.29
```

我们可以对比下微调前后的效果

以命令行 Demo 为例，只需修改cli_demo.py中的模型路径为：ptuning/out/adgen-chatglm-6b-pt-8-1e-2/checkpoint-3000，运行 cli_demo.py即可：

python cli_demo.py

用以下数据为例：

Input: 类型#上衣*材质#牛仔布*颜色#白色*风格#简约*图案#刺绣*衣样式#外套*衣款式#破洞 Label: 简约而不简单的牛仔外套,白色的衣身十分百搭。衣身多处有做旧破洞设计,打破单调乏味,增加一丝造型看点。衣身后背处有趣味刺绣装饰,丰富层次感,彰显别样时尚。这件上衣的材质是牛仔布,颜色是白色,风格是简约,图案是刺绣,衣样式是外套,衣款式是破洞。

用户：根据输入生成一段广告词，输入为：类型#上衣*材质#牛仔布*颜色#白色*风格#简约*图案#刺绣*衣样式#外套*衣款式#破洞。

Output[微调前]:

```
欢迎使用 ChatGLM-6B 模型，输入内容即可进行对话，clear 清空对话历史，stop 终止程序

用户：根据输入生成一段广告词，输入为：类型#上衣*材质#牛仔布*颜色#白色*风格#简约*图案#刺绣*衣样式#外套*衣款式#破洞。

ChatGLM-6B: 时尚简约，个性破洞！这款牛仔上衣采用优质牛仔布，颜色为白色，风格为简约，图案为刺绣。它还可以根据自己的喜好选择衣样式，包括外套和衣款式。无论是搭配牛仔褲还是其他褲子，都能够展现出独特的时尚感。此外，破洞的设计更是增添了一份随意感和休闲感，让你的穿着更加舒适自然。不要错过这款时尚的牛仔上衣，赶快试试看吧！
```

Output[微调后]:

欢迎使用 ChatGLM-6B 模型，输入内容即可进行对话，**clear** 清空对话历史，**stop** 终止程序

用户：根据输入生成一段广告词，输入为：类型#上衣*材质#牛仔布*颜色#白色*风格#简约*图案#刺绣*衣样式#外套*衣款式#破洞。

ChatGLM-6B：简约风格外套，采用牛仔布材质，穿着舒适有型，搭配白色刺绣图案，展现甜美俏皮气质。经典的破洞设计，轻松穿出时尚范。

总结：建议使用官方提供的基于P-Tuning v2微调ChatGLM-6B的方式对自己的数据进行微调

此外，[此文](#)还介绍了如何通过Cursor一步一步生成一份微调ChatGLM的示例代码

第二部分 ChatGLM-6B的代码架构与逐行实现

ChatGLM-6B([介绍页面](#))，是智谱 AI 开源、支持中英双语的对话语言模型。

话不多说，直接干，虽然6B的版本相比GPT3 175B 不算大，但毕竟不是一个小工程，本文就不一一贴所有代码了，更多针对某个文件夹下或某个链接下的代码进行整体分析/说明，以帮助大家更好、更快的理解ChatGLM-6B，从而加速大家的类ChatGPT复现之路

- 其对应的Hugging Face上([THUDM/chatglm-6b](#) · [Hugging Face](#)，详见下文的2.1 2.2 2.3节)

存放的是chatglm-6b的模型文件，包含权重、配置文件、模型信息等等

main chatglm-6b		7 contributors		History: 79 commits	
zxdu20	Update slack link	0829959			about 8 hours ago
.gitattributes		1.48 kB	↓	Add chatglm-6b	about 1 month ago
LICENSE		11.3 kB	↓	Add chatglm-6b	about 1 month ago
MODEL_LICENSE		2.35 kB	↓	Add chatglm-6b	about 1 month ago
README.md		5.82 kB	↓	Update slack link	about 8 hours ago
config.json		773 Bytes	↓	Update code for slim	11 days ago
configuration_chatglm.py		4.28 kB	↓	Update code for slim	11 days ago
ice_text.model		2.71 MB	🔥 LFS ↓	Drop icetk dependency	11 days ago
modeling_chatglm.py		57.6 kB	↓	Change mask positions to batch	3 days ago
pytorch_model-00001-of-00008.bin	📁 pickle	1.74 GB	🔥 LFS ↓	Update slim checkpoint (#28)	11 days ago
pytorch_model-00002-of-00008.bin	📁 pickle	1.88 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00003-of-00008.bin	📁 pickle	1.98 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00004-of-00008.bin	📁 pickle	1.91 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00005-of-00008.bin	📁 pickle	1.88 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00006-of-00008.bin	📁 pickle	1.88 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00007-of-00008.bin	📁 pickle	1.07 GB	🔥 LFS ↓	Add model file	about 1 month ago
pytorch_model-00008-of-00008.bin	📁 pickle	1.07 GB	🔥 LFS ↓	Update slim checkpoint	11 days ago
pytorch_model.bin.index.json		33.4 kB	↓	Add chatglm-6b	about 1 month ago
quantization.py		15.1 kB	↓	Add support for loading quantized model	18 days ago
tokenization_chatglm.py		16.7 kB	↓	Always add gmask in token ids	3 days ago
tokenizer_config.json		441 Bytes	↓	Fix eos token in tokenizer	6 days ago

其中，pytorch_model-00001-of-00008.bin 到 pytorch_model-00008-of-00008.bin: 这些文件是PyTorch模型的权重文件，相当于一个大模型被分割成多个部分以方便下载和使用

- 其对应的GitHub上([GitHub - THUDM/ChatGLM-6B: ChatGLM-6B: An Open Bilingual Dialogue Language Model](#) | 开源双语对话语言模型，详见下文的2.4节)主要是ChatGLM-6B/ptuning，涉及对chatglm-6b进行推理、部署、微调的代码(就是如何使用的相关代码)

2.1 模型的核心实现： chatglm-6b/modeling_chatglm.py

2.1.1 导入相关库、编码器、GELU、旋转位置编码(第1-239行)

- 首先，代码导入了许多需要的库，如torch、torch.nn.functional等，它们为模型实现提供了基本的功能。
脚本中设置了一些标志，以便在运行时启用JIT（Just-In-Time）编译功能
- 定义了InvalidScoreLogitsProcessor类，它继承自LogitsProcessor。该类用于处理可能出现的NaN和inf值，通过将它们替换为零来确保计算的稳定性
- load_tf_weights_in_chatglm_6b函数，用于从TensorFlow检查点加载权重到PyTorch模型中。这对于迁移学习和在PyTorch中使用预训练模型非常有用
- PrefixEncoder类是一个编码器，用于对输入的前缀进行编码。它根据配置使用一个两层的MLP(多层感知器)或者直接进行嵌入，输出维度为(batch_size, prefix_length, 2 * layers * hidden)
- gelu_impl函数是一个GELU(高斯误差线性单元)激活函数的实现，这是一个常用的激活函数，尤其在Transformer模型中

```
1 # 使用PyTorch的JIT编译器，将Python函数转换为Torch脚本，以便优化和加速执行
2 @torch.jit.script
3 # 定义名为gelu_impl的函数，接受一个参数x
4 def gelu_impl(x):
```