chatglm系列模型

1.ChatGLM

1.1 背景

主流的预训练框架主要有三种：

1. **autoregressive自回归模型（AR模型）**：代表作GPT。本质上是一个left-to-right的语言模型。**通常用于生成式任务**，在长文本生成方面取得了巨大的成功，比如自然语言生成（NLG）领域的任务：摘要、翻译或抽象问答。当扩展到十亿级别参数时，表现出了少样本学习能力。缺点是单向注意力机制，在NLU任务中，无法完全捕捉上下文的依赖关系。
2. **autoencoding自编码模型（AE模型）**：代表作BERT。是**通过某个降噪目标（比如MLM）训练的双向文本编码器。**编码器会产出适用于NLU任务的上下文表示，但无法直接用于文本生成。
3. **encoder-decoder（Seq2seq模型）**：代表作T5。采用双向注意力机制，**通常用于条件生成任务**，比如文本摘要、机器翻译等。

三种预训练框架各有利弊，没有一种框架在以下三种领域的表现最佳：自然语言理解（NLU）、无条件生成以及条件生成。T5曾经尝试使用MTL的方式统一上述框架，然而自编码和自回归目标天然存在差异，简单的融合自然无法继承各个框架的优点。在这个天下三分的僵持局面下，GLM诞生了。**GLM模型基于autoregressive blank infilling方法，结合了上述三种预训练模型的思想。**

1.2 GLM预训练框架

GLM特点：

1.自编码思想：在输入文本中，随机删除连续的tokens。

2.自回归思想：顺序重建连续tokens。在使用自回归方式预测缺失tokens时，模型既可以访问corrupted文本，又可以访问之前已经被预测的spans。

3.span shuffling + 二维位置编码技术。

4.通过改变缺失spans的数量和长度，自回归空格填充目标可以为条件生成以及无条件生成任务预训练语言模型。

（1）自回归空格填充任务

给定一个输入文本 $x = [x_1, \ldots, x_n]$，可以采样得到多个文本spans$\{s_1, \ldots, s_m\}$。为了充分捕捉各spans之间的相互依赖关系，可以对spans的顺序进行随机排列，得到所有可能的排列集合 Z_m，对于每一个排列 z，我们定义：$S_{z<i} = [s_{z_1}, \ldots, s_{z_{i-1}}]$。所以预训练目标很清晰是最大化以下期望对数似然：

$$\max_{\theta} \mathbb{E}_{z \sim Z_m} \left[\sum_{i=1}^{m} \log p_\theta(s_{z_i} \mid x_{\text{corrupt}}, S_{z<i}) \right]$$

GLM 的“自回归空格填充”任务，目的是让模型从一个**被遮掩（mask）过的文本中**，逐步生成那些被遮住的内容。这些被遮住的部分叫做 **spans（片段）**，模型需要一个一个地预测它们。简单来说，就是让模型根据当前上下文和前面已经生成的内容片段来预测下一个内容片段的内容。

GLM自回归空格填充任务的技术细节：

1.输入 x 可以被分成两部分：Part A是**可见部分（corrupt 后的文本）** x_{corrupt}，Part B由masked spans组成。
假设原始输入文本是 $[x_1, x_2, x_3, x_4, x_5, x_6]$。采样的两个文本片段是$x_3$以及$[x_5, x_6]$,那么mask后的文本序列是：$x_1, x_2, [M], x_4, [M]$即Part A；同时我们需要对Part B的片段进行随机打乱（shuffle）。每个片段使用[S]填充在开头作为输入，使用[E]填充在末尾作为输出。

2.二维位置编码：Transformer使用位置编码来标记tokens中的绝对和相对位置。在GLM中，使用二维位置编码，第一个位置id用来标记Part A中的位置，第二个位置id用来表示跨度内部的相对位置。这两个位置id会通过embedding表被投影为两个向量，最终都会被加入到输入token的embedding表达中。

3.注意力掩码设计（Attention Mask）：观察GLM中自定义attention mask的设计，非常巧妙：

- Part A 内部 tokens 彼此可见，但不可见 Part B 的任意 token。
- Part B 中的 tokens 可以看到 Part A 的所有 token。
- Part B 中的 tokens 只能看到之前已经生成的 span，不能看到未来的 span。

4.采样方式：文本片段的采样遵循泊松分布，重复采样，直到原始tokens中有15%被mask。

5.总结：模型可以自动学习双向encoder（Part A）以及单向decoder（Part B）。**既让模型学会了像 BERT 一样理解上下文（双向 encoder），又让它具备了像 GPT 一样的逐步生成能力（单向 decoder）**

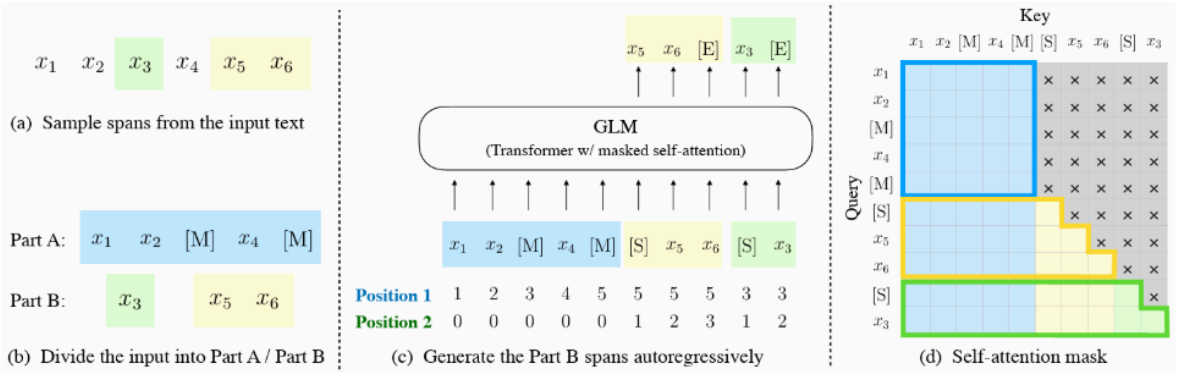

Figure 2: GLM pretraining. (a) The original text is $[x_1, x_2, x_3, x_4, x_5, x_6]$. Two spans $[x_3]$ and $[x_5, x_6]$ are sampled (b) Replace the sampled spans with [M] in Part A, and shuffle the spans in Part B. (c) GLM autoregressively generates Part B. Each span is prepended with [S] as input and appended with [E] as output. 2D positional encoding represents inter- and intra-span positions. (d) Self-attention mask. Grey areas are masked out. Part A tokens can attend to themselves (blue frame) but not B. Part B tokens can attend to A and their antecedents in B (yellow and green frames correspond to the two spans). [M] := [MASK], [S] := [START], and [E] := [END].

（2）多目标预训练

上述方法适合于NLU任务。作者希望可以训练一个既可以解决NLU任务，又具备文本生成能力的模型。因此除了空格填充目标之外，还需要增加一个生成长文本目标的任务。具体包含以下两个目标：

1. **文档级别**：该目标旨在提升模型在**长文本生成** 和**全局语义建模** 方面的能力。具体做法是从一篇完整文档中随机采样一个连续文本片段进行遮蔽（mask），要求被遮蔽的片段长度占整个文档长度的 **50%～100%**。随后，模型需要根据未被遮蔽的部分，逐步重建被遮蔽的大段内容。
   这一目标的设计有助于模型学习：**长距离依赖关系和全局语义连贯性以及上下文逻辑一致性**，适用于如文章续写、故事生成、知识密集型问答等需要整体把握文本结构的任务。
2. **句子级别**：专注于强化模型对**完整句子或段落级生成** 的能力。其核心思想是：**强制要求被遮蔽的 span 必须是一个完整的句子**，并且多个被遮蔽的句子总共应覆盖原始输入 token 的 **15%** 左右。

   通过这种方式，模型可以在序列到序列（seq2seq）框架下更好地学习：**句子的语法结构和语义完整性以及上下句之间的衔接逻辑**，该目标特别适合用于诸如摘要生成、机器翻译、对话回复生成等典型的 seq2seq 任务。

三个预训练目标对比总结：

预训练目标	描述	覆盖比例	特点	适合任务
空格填充（Blank Filling）	随机采样任意长度的 span，打乱顺序生成	~15% tokens	局部上下文建模强	NLU、小段文本生成
文档级别（Document-Level）	mask 占文档长度 50%~100% 的大 span	50%100%	建模长距离依赖	长文本生成、文档理解
句子级别（Sentence-Level）	mask 完整句子，总 cover 15% 的 token	~15%（句子级）	强化句子结构理解	Seq2Seq、摘要、对话

（3）模型结构

GLM在原始single Transformer的基础上进行了一些修改：

1. 重组了LN和残差连接的顺序；使用 Pre-LN 结构，有助于缓解梯度消失、提升训练稳定性。
2. 使用单个线性层对输出token进行预测；在模型的最后一层，不使用复杂的分类头（如两层 FFN），而是直接通过一个线性变换将 hidden state 映射为词表上的概率分布，从而决定下一个 token 是什么。
3. 激活函数从ReLU换成了GeLU。

但我觉得这部分的修改比较简单常见。核心和亮点还是空格填充任务的设计。

（4）GLM微调

对于下游NLU任务来说，通常会将预训练模型产出的序列或tokens表达作为输入，使用线性分类器预测label。所以预训练与微调之间存在天然不一致。GLM 在微调阶段采用了“Prompt-based”方法，**把 NLU 任务转化为类似预训练阶段的“空白填充生成任务”，从而实现预训练与微调目标的一致性。**虽然这个做法在一定程度上有些“强行转化”，但它确实能让模型更好地发挥其在预训练中学到的语言理解和生成能力。

作者按照PET的方式，将下游NLU任务重新表述为空白填充的生成任务。具体来说，比如给定一个已标注样本(x, y)，将输入的文本x转换成一个包含mask token的完形填空问题。比如，情感分类任务可以表述为："{SENTENCE}. It's really [MASK]"。输出label y也同样会被映射到完形填空的答案中。“positive” 和 “negative” 对应的标签就是“good” 和 “bad。

其实，预训练时，对较长的文本片段进行mask，以确保GLM的文本生成能力。但是在微调的时候，相当于将NLU任务也转换成了生成任务，这样其实是为了适应预训练的目标。

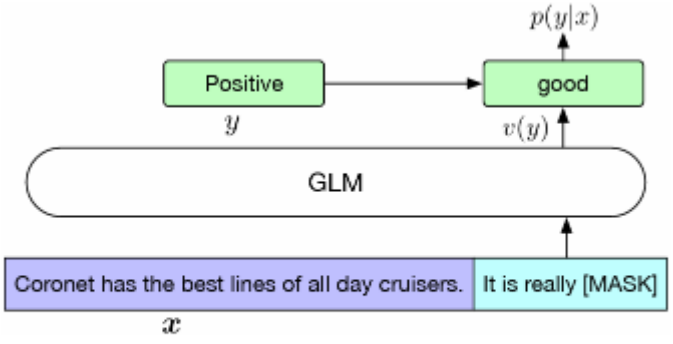

Figure 3: Formulation of the sentiment classification task as blank infilling with GLM.

对比 Prompt-based 微调 与 传统 NLU 微调

对比维度	传统 NLU 方法（如 BERT）	GLM 的 Prompt-based 微调
微调目标形式	分类任务（classification）	生成任务（generation / blank filling）
模型输出方式	使用 [CLS] 向量 + 线性分类器	使用生成机制预测 mask token
训练目标一致性	与 MLM 预训练目标不一致	与 GLM 预训练目标高度一致

模型比较：

BERT	XLNet	T5	UniLM
1、无法捕捉mask tokens的相互依赖性。2、不能准确填充多个连续的tokens。为了推断长度为l的答案概率，BERT需要执行l次连续预测。	与GLM相同，使用自回归目标预训练。1、使用文本mask之前的原始位置编码，推理过程中，需要事先知晓或枚举答案长度，与BERT的问题相同。2、双向自注意力机制，使预训练时间成本增加了一倍。	使用类似的空格填充目标预训练encoder-decoder Transformer。在编码和解码阶段使用独立的位置编码，使用多个哨兵token来区分mask片段。而在下游任务中，仅使用一个哨兵token，造成模型能力的浪费以及预训练-微调的不一致。	通过改变双向、单向以及交叉注意力之间的注意力mask，统一不同的预训练目标。1、总是使用[mask] token替代mask片段，限制了它对mask片段及其上下文的依赖关系进行建模的能力。2、在下游任务微调时，自编码比自回归更加低效。

2.ChatGLM-2

2.1 主要创新

1. **更长的上下文**：**基于 FlashAttention 技术**，将基座模型的上下文长度（Context Length）由 ChatGLM-6B 的 **2K 扩展到了 32K**，并在对话阶段使用 8K 的上下文长度训练。对于更长的上下文，发布了 ChatGLM2-6B-32K 模型。LongBench 的测评结果表明，在等量级的开源模型中，ChatGLM2-6B-32K 有着较为明显的竞争优势。
2. **更强大的性能**：基于 ChatGLM 初代模型的开发经验，全面升级了 ChatGLM2-6B 的基座模型。ChatGLM2-6B **使用了GLM 的混合目标函数**，经过了 1.4T 中英标识符的预训练与人类偏好对齐训练，评测结果显示，相比于初代模型，ChatGLM2-6B 在 MMLU（+23%）、CEval（+33%）、GSM8K（+571%） 、BBH（+60%）等数据集上的性能取得了大幅度的提升，在同尺寸开源模型中具有较强的竞争力。
3. **更高效的推理**：基于 Multi-Query Attention 技术，ChatGLM2-6B 有更高效的推理速度和更低的显存占用：在官方的模型实现下，推理速度相比初代提升了 42%，INT4 量化下，6G 显存支持的对话长度由 1K 提升到了 8K。
4. **更开放的协议**：ChatGLM2-6B 权重对学术研究**完全开放**，在填写问卷进行登记后**亦允许免费商业使用。**

2.2 与ChatGLM的变化

1. **使用了RoPE替换二维位置编码。**这也是GLM中提出的亮点设计之一。但是目前大部分主流的LLMs都在使用RoPE，所以大势所趋。当前版本仍然采用了最初的RoPE设计，事实上现在的RoPE经过了xPOS→线性内插→NTK-Aware Scaled RoPE→...若干次进化。
2. **Multi-Query Attention**：这是一种共享机制的Attention，相比Multi-Head Attention，其Query部分没有区别，Key和Value可以只用一个Head。计算时，对Key和Value进行expand或者repeat操作，使它们填充到与Query一样的维度，后续计算就与Multi-Head Attention没区别。
3. **Attention Mask**: V1的attention mask分了2部分，Part A和Part B，Part A部分是双向Attention（代码中的prefix_attention_mask），Part B部分是Causal Attention(原代码文件中的get_masks函数)。在V2版本，全部换成了Causal Attention，不再区分是Part A还是Part B，**完全变成了decoder-only的架构。**
4. **多目标任务**：Chat版本主要还是用的gMask生成式任务，但是在V1版本的代码还能看到mask、gMask等字样，V2已经摒弃了这些特殊token，原因与Attention Mask一致，均因为变成了decoder-only的架构，不再需要区分Part A和Part B。

3.ChatGLM-3

ChatGLM2与ChatGLM3模型架构是完全一致的，ChatGLM与后继者结构不同。可见ChatGLM3相对于ChatGLM2没有模型架构上的改进。

相对于ChatGLM，ChatGLM2、ChatGLM3模型上的变化：

1. 词表大小从 150528 缩减至 65024，显著提升了模型加载速度并增强了训练稳定性。
2. 位置编码由每个 GLM Block 独立计算改为全局共享一份，减少了重复计算，提高了推理效率。
3. 在 Self-Attention 后的前馈网络（FFN）中，激活函数从 GELU 改为 Swish-1（即 SiLU），并采用了类似 GLU 的门控结构。这种结构会将输入分别映射为“数据路径”和“门控路径”，其中一条路径用于控制另一条路径的信息流动。因此，虽然 FFN 的输出维度从 27392 下降至 13696，看似信息压缩，实则是将一半参数用于门控机制，属于门控激活函数的标准做法，不仅没有损失表达能力，反而增强了模型对信息流的控制力，有助于提升整体性能。

4.模型架构比较

```
tokenizer = AutoTokenizer.from_pretrained(model_path, trust_remote_code=True)
model = AutoModel.from_pretrained(model_path,
trust_remote_code=True).float().to('mps')
# 多显卡支持，使用下面两行代替上面一行，将num_gpus改为你实际的显卡数量
# from utils import load_model_on_gpus
# model = load_model_on_gpus("THUDM/chatglm3-6b", num_gpus=2)
model = model.eval()

print(model)
```

ChatGLM的模型结构：

```
ChatGLMForConditionalGeneration(
  (transformer): ChatGLMModel(
    (word_embeddings): Embedding(150528, 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=150528, bias=False)
)
```

ChatGLM2的模型结构：

```
ChatGLMForConditionalGeneration(
  (transformer): ChatGLMModel(
    (embedding): Embedding(
      (word_embeddings): Embedding(65024, 4096)
    )
    (rotary_pos_emb): RotaryEmbedding()
    (encoder): GLMTransformer(
      (layers): ModuleList(
        (0-27): 28 x GLMBlock(
          (input_layernorm): RMSNorm()
          (self_attention): SelfAttention(
            (query_key_value): Linear(in_features=4096, out_features=4608,
bias=True)
            (core_attention): CoreAttention(
              (attention_dropout): Dropout(p=0.0, inplace=False)
            )
            (dense): Linear(in_features=4096, out_features=4096, bias=False)
          )
          (post_attention_layernorm): RMSNorm()
          (mlp): MLP(
            (dense_h_to_4h): Linear(in_features=4096, out_features=27392,
bias=False)
            (dense_4h_to_h): Linear(in_features=13696, out_features=4096,
bias=False)
          )
        )
      )
      (final_layernorm): RMSNorm()
    )
    (output_layer): Linear(in_features=4096, out_features=65024, bias=False)
  )
)
```

ChatGLM3的模型结构：

```
ChatGLMForConditionalGeneration(
  (transformer): ChatGLMModel(
    (embedding): Embedding(
      (word_embeddings): Embedding(65024, 4096)
    )
    (rotary_pos_emb): RotaryEmbedding()
    (encoder): GLMTransformer(
      (layers): ModuleList(
        (0-27): 28 x GLMBlock(
          (input_layernorm): RMSNorm()
          (self_attention): SelfAttention(
            (query_key_value): Linear(in_features=4096, out_features=4608,
bias=True)
            (core_attention): CoreAttention(
              (attention_dropout): Dropout(p=0.0, inplace=False)
            )
            (dense): Linear(in_features=4096, out_features=4096, bias=False)
          )
          (post_attention_layernorm): RMSNorm()
          (mlp): MLP(
            (dense_h_to_4h): Linear(in_features=4096, out_features=27392,
bias=False)
            (dense_4h_to_h): Linear(in_features=13696, out_features=4096,
bias=False)
          )
        )
      )
      (final_layernorm): RMSNorm()
    )
    (output_layer): Linear(in_features=4096, out_features=65024, bias=False)
  )
)
```