

赞同 159

分享

从头预训练一只超迷你 LLaMA 3——复现 Tiny Stories



Xode

没啥想法，到处看看

已关注

159 人赞同了该文章

我的代码：

[git@github.com:Mxoder/LLM-from-scratch.git](https://github.com/Mxoder/LLM-from-scratch.git)

这次打算用 Hugging Face 的 API 来写一份预训练大（小）模型的代码，也就是用 Trainer 来做预训练。由于只是想练习一下，因此打算选一个极小模型 + 小数据集。为了贴近主流，于是打算预训练一个 LLaMA 3——不过是超迷你版本，大小仅不到 20M。

想起来曾经看到过的微软的工作 TinyStories，探索的是语言模型在多小的情况下还能流利地讲故事，工作非常直白、有趣，刚好也契合我的练习想法，于是这次来复现一下。

代码放在这里了：[GitHub - Mxoder/LLM-from-scratch: 一些 LLM 的从零复现笔记。](#)

2024.06.09：更新了第二篇文章 [PyTorch 从零复现 LoRA](#)，代码在上面的仓库。

1. 前期准备

让我们先来想一想大概需要做什么。

首先是**模型架构**的选择。原工作用的是 GPT Neo 架构（可以看他们的 [config](#)），这个算是很老的模型了，最初是 EleutherAI 用来复现追踪 GPT-3 的工作的，现在用的也比较少了。我打算选用 LLaMA 架构，也算是符合研究主流、便于推广。LLaMA 3 主要多了个 GQA，也是现在模型的主流，我这里也用一下。

其次是**数据**的选择。既然是复现，就直接贯彻拿来主义，用原工作开源的数据集（主要是从头生成要花不少 api 费用）。原工作第一版的时候用的是 GPT-3.5 生成的数据，后面社区有人更新了**第二版**，是用 GPT-4 生成的，比原数据更好，就用它了。

最后是**训练**。其实我手上就两张 3060 12G 和 4060 Ti 16G，训这个确实是绰绰有余，但我还是不想在桌前吵我自己，于是继续用 Colab。现在 Colab 可以直接看到剩余使用时长了，虽然已经被砍到只有 3h 左右的用卡时间，但至少心里有个底，况且 3h 训我们这个也完全够了。

我们这次用到的 Hugging Face 的库如下：



```
transformers
accelerate
datasets
```



理论上比较新的版本都没问题，但如果你很久没更新了，最好用 `pip install -U` 来升级一下。

我这里的用到的库版本如下，供参考：

```
torch==2.2.1
transformers==4.40.0
accelerate==0.29.3
datasets==2.18.0
```

另外，接下来的步骤讲解主要是以 jupyter notebook 的形式展开的，并不是 `.py` 文件的形式，也就是说前面执行的变量会在中间储存下来。

2. 原工作简介

虽然是练习，但既然打着复现工作的名头，还是来简要回顾一下原工作究竟做了些什么吧。

原工作探索的问题是语言模型（LM）在文本连贯性上的表现。像早期的一些语言模型如 GPT-2，即使在一些 Common Crawl 这样的语料库上大量预训练后，也很难生成长的、连贯的文本。比如前几年有一种 AI 玩具类型是做文本续写，例如彩云小梦，可以写写作文、小说什么的，如果大家玩过就知道效果其实一言难尽，和今天的大模型完全没法比，其实这就是 GPT-2 level 的续写能力。

作者就在想，会不会是因为训练的语料库太多、太宽泛，需要学习各种语法元素、词汇、知识、推理等等，才导致小语言模型（SLM）没法有一个很好的表现。作者决定专注于一个任务——短篇故事续写，来探索一下 LM 的性能边界。

作者用 GPT-4 和 GPT-3.5 构建了一个英文短篇小说数据集 TinyStories，将内容限制在三四岁儿童也能轻松理解的程度，并且使用不同的关键词来让故事的主题足够丰富。此外，他们还加入了额外的关键词，来控制故事有更曲折的走向、不同的结局等等。

作者用的模型基座架构是 GPT Neo，词表大小约为 50k，并且他们尝试了不同的模型参数，调整了隐藏层维度（`hidden_size`）、隐藏层数（`num_hidden_layers`）等，来探索不同参数对于模型性能的影响。

作者的评估方式是经典的 GPT-4 监督打分模式，就是让不同的 SLM 根据提示生成故事，然后 GPT-4 从设定好的不同维度来打分，主要有 Creativity、Grammar、Consistency 三项，分别代表**创造性**、**语法正确性**、**上下文一致性**。此外，作者额外加入了一套 TinyStories-Instruct 数据集，来训练一批指令微调的 SLM，并测试他们的**指令跟随能力**，也就是第四项 Instruct。

作者主要和 GPT-Neo 以及 GPT-2 的小中杯进行了对比。

3. 模型初始化

让我们正式开始复现！

3.1 决定模型的参数

首先是定义我们自己的模型。由于 LLaMA 3 的架构早就集成于 transformers 库中，因此我们可以直接用 `AutoConfig` 初始化一个模型配置，传入参数 `model_type="llama"` 即可。

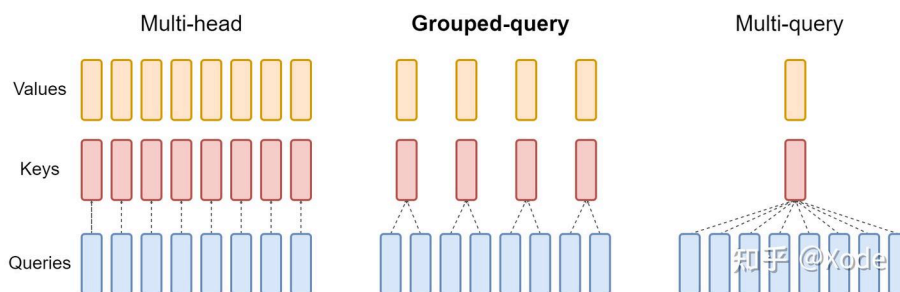
架构确定了，那么现在来探讨一下模型具体参数，比如隐藏层大小、隐藏层数等等。我们先来看看 TinyStories 原工作的实验结果：

Hidden size	Layer		Eval loss	Creativity	Grammar	Consistency	Instruct	Plot
64	12		2.02	4.84/0.36	6.19/0.42	4.75/0.31	4.34/0.23	4.39/0.20
64	8		2.08	4.68/0.33	6.14/0.41	4.45/0.27	4.34/0.23	4.40/0.21
64	4		2.26	3.97/0.20	5.31/0.22	3.77/0.18	3.79/0.14	3.71/0.06
64	2		2.38	2.94/0.00	4.33/0.00	2.41/0.00	2.86/0.00	3.40/0.00
128	12		1.62	6.02/0.58	7.25/0.66	7.20/0.64	6.94/0.63	6.58/0.65
128	8		1.65	5.97/0.57	7.23/0.66	7.10/0.62	6.87/0.62	6.16/0.57
128	4		1.78	5.70/0.52	6.91/0.58	6.60/0.56	6.00/0.49	5.53/0.44
128	2		1.92	4.90/0.37	6.43/0.48	4.75/0.31	5.23/0.37	4.89/0.31
256	12		1.34	6.66/0.71	7.80/0.79	8.38/0.79	7.68/0.75	7.18/0.78
256	8		1.38	6.54/0.68	7.72/0.77	8.02/0.75	7.92/0.78	7.23/0.79
256	4		1.47	6.32/0.64	7.64/0.75	7.76/0.71	8.07/0.81	7.18/0.78
256	2		1.60	6.23/0.62	7.50/0.72	7.20/0.64	7.23/0.68	6.50/0.64
512	12		1.19	6.90/0.75	8.46/0.93	9.11/0.89	8.21/0.83	7.37/0.82
512	8		1.20	6.85/0.74	8.34/0.91	8.95/0.87	8.05/0.80	7.26/0.79
512	4		1.27	6.75/0.72	8.35/0.91	8.50/0.81	8.34/0.85	7.36/0.81
512	2		1.39	6.40/0.66	7.72/0.77	7.90/0.73	7.76/0.76	7.13/0.77
768	12		1.18	7.00/0.77	8.30/0.90	9.20/0.90	8.23/0.83	7.47/0.84
768	8		1.18	7.02/0.77	8.62/0.97	9.34/0.92	8.36/0.85	7.34/0.81
768	4		1.20	6.89/0.75	8.43/0.93	9.01/0.88	8.44/0.87	7.52/0.85
768	2		1.31	6.68/0.71	8.01/0.83	8.42/0.80	7.97/0.79	7.34/0.81
768	1		1.54	6.00/0.58	7.35/0.68	7.25/0.64	5.81/0.46	6.44/0.63
1024	12		1.22	7.05/0.78	8.43/0.93	8.98/0.87	8.18/0.82	7.29/0.80
1024	8		1.20	7.13/0.80	8.25/0.89	8.92/0.87	8.47/0.87	7.47/0.84
1024	4		1.21	7.04/0.78	8.32/0.90	8.93/0.87	8.34/0.85	7.47/0.84
1024	2		1.27	6.68/0.71	8.22/0.88	8.52/0.81	8.04/0.80	7.24/0.79
1024	1		1.49	6.36/0.65	7.77/0.78	7.47/0.67	6.09/0.50	6.42/0.62
GPT-Neo (125M)	-	-	-	3.34/0.08	5.27/0.21	4.22/0.24	-	-
GPT-2-small (125M)	-	-	-	3.70/0.14	5.40/0.24	4.32/0.25	-	-
GPT-2-med (355M)	-	-	-	4.22/0.24	6.27/0.44	5.34/0.39	-	-
GPT-2-large (774M)	-	-	-	4.30/0.26	6.43/0.48	6.04/0.48	-	-
GPT-4	-	-	-	8.21/1.00	8.75/1.00	9.93/1.00	9.31/1.00	8.26/1.00

不同隐藏层大小和层数对结果的影响评估（结果的格式为 a / b, a 表示原始分数, b 表示归一化分数）

可以看到，隐藏层维度从 64 增长到 256 时的收益是比较大的，往后收益就逐渐放缓了。而层数的影响并不如隐藏层维度那么大，大而浅的网络也能有不错的表现（例如 hidden_size=1024, num_hidden_layers=1 的模型）。综合考虑，我这里选择 hidden_size=256 和 num_hidden_layers=4。

其他参数方面，我们遵循现在主流的研究表现，将 FFN 的维度从传统的 4 倍隐藏层维度设为 8/3 倍（按 128 向上取整）。头的数目我们设为 16，并应用 GQA 机制。GQA 的实现在 transformers 中非常简单，只需要配置 num_key_value_heads 即可。num_key_value_heads 取值和 num_attention_heads 相同时即为 MHA 机制，取值为 1 时即为 MQA 机制。



MHA、GQA 和 MQA，来源：<https://arxiv.org/abs/2305.13245>

综上，我们的配置如下：

```
# 模型配置
from transformers import AutoConfig

hidden_size = 256
# 中间层取 8/3 倍，按 128 向上取整
intermediate_size = (int(hidden_size * 8/3 / 128) + 1) * 128

# 只改动我们需要调整的参数，其余保持不变
config = AutoConfig.for_model(
    model_type="llama",
    hidden_size=hidden_size,
    intermediate_size=intermediate_size,
    num_attention_heads=16,
    num_hidden_layers=4,
    num_key_value_heads=8
)
# 分为 8 组
# 即两个query共享一个key
```

```

"""
LlamaConfig {
    "attention_bias": false,                # 不使用注意力偏置
    "attention_dropout": 0.0,              # 注意力层的 dropout 比例
    "bos_token_id": 1,                     # bos_token (begin of sentence) 的 i
    "eos_token_id": 2,                     # eos_token (end of sentence) 的 id
    "hidden_act": "silu",                  # 隐藏层激活函数类型, silu 即 SwiGLU
    "hidden_size": 256,                    # 隐藏层维度大小
    "initializer_range": 0.02,             # 权重初始化范围, 会被后面的 Kaiming 初始
    "intermediate_size": 768,              # 中间层大小, 采用 8/3 倍而非 4 倍
    "max_position_embeddings": 2048,
    "model_type": "llama",
    "num_attention_heads": 16,
    "num_hidden_layers": 4,
    "num_key_value_heads": 8,
    "pretraining_tp": 1,
    "rms_norm_eps": 1e-06,
    "rope_scaling": null,
    "rope_theta": 10000.0,
    "tie_word_embeddings": false,          # 头尾 embedding 和 lm_head 是否共享权重
    "transformers_version": "4.40.0",      # 注意: 此处并不共享权重
    "use_cache": true,
    "vocab_size": 32000
}
"""

```

3.2 分词器 Tokenizer

我这里选用 LLaMA 2 的分词器，因为二代的词表比较小（32k），LLaMA 3 的词表太大了（128k），在 SLM 中会占用太多的参数比重，并且这只是个专有任务数据训练，没必要用太大的词表。

```

# 分词器
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('NousResearch/Llama-2-7b-hf')

"""
LlamaTokenizerFast(name_or_path='NousResearch/Llama-2-7b-hf', vocab_size=32000,
    0: AddedToken("<unk>", rstrip=False, lstrip=False, single_word=False, norma
    1: AddedToken("<s>", rstrip=False, lstrip=False, single_word=False, normali
    2: AddedToken("</s>", rstrip=False, lstrip=False, single_word=False, normal
}
"""

```

另外注意这里 padding_side='left'，如果不是的话需要设置 `tokenizer.padding_side='left'`，即批量填充的时候从左边开始填充，这对于 decoder-only 的模型做生成任务是必要的，因为我们本质上做的是 next token prediction，如果 pad 挡在了生成序列的右边，会影响到模型生成。

```

# 假设 pad_token 就是 eos_token(</s>)
# 从右边填充
Once upon a time </s></s></s></s>...
# 从左边填充
</s></s></s></s>Once upon a time ...

```

3.3 模型实例化

接下来就是实例化模型，这里就不用从预训练模型加载 `from_pretrained()` 了，而是从配置加载 `from_config()`：

```

# 模型
import torch
from transformers import AutoModelForCausalLM

# 能用 cuda 就用 cuda
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 从配置加载模型
model = AutoModelForCausalLM.from_config(
    config,
    torch_dtype=torch.float32 # 全精度训练
).to(device) # 迁移到 device 上

"""
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(32000, 256)
    (layers): ModuleList(
      (0-3): 4 x LlamaDecoderLayer(
        (self_attn): LlamaSdpaAttention(
          (q_proj): Linear(in_features=256, out_features=256, bias=False)
          (k_proj): Linear(in_features=256, out_features=128, bias=False)
          (v_proj): Linear(in_features=256, out_features=128, bias=False)
          (o_proj): Linear(in_features=256, out_features=256, bias=False)
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=256, out_features=768, bias=False)
          (up_proj): Linear(in_features=256, out_features=768, bias=False)
          (down_proj): Linear(in_features=768, out_features=256, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=256, out_features=32000, bias=False)
)
"""

```

可以看到，k_proj 和 v_proj 的 out_features 从 256 变为了 128，这即是 GQA 机制。

此时，模型已经初始化了，让我们来打印一下看看参数：

```

# 打印模型的每一层及其参数大小
def print_model_parameters(model):
    print("Layer Name & Parameters")
    print("-----")
    total_params = 0
    for name, parameter in model.named_parameters():
        param_size = parameter.size()
        param_count = torch.prod(torch.tensor(param_size)).item()
        total_params += param_count
        print(f"{name:50} | Size: {str(param_size):30} | Count: {str(param_count):30}")
    print("-----")
    print(f"Total Parameters: {total_params} ({total_params / 1000000:.1f} M)")

print_model_parameters(model)

```

得到结果如下：

Layer Name & Parameters	
model.embed_tokens.weight	Size: torch.Size([32000, 256])
model.layers.0.self_attn.q_proj.weight	Size: torch.Size([256, 256])
model.layers.0.self_attn.k_proj.weight	Size: torch.Size([128, 256])
model.layers.0.self_attn.v_proj.weight	Size: torch.Size([128, 256])
model.layers.0.self_attn.o_proj.weight	Size: torch.Size([256, 256])
model.layers.0.mlp.gate_proj.weight	Size: torch.Size([768, 256])
model.layers.0.mlp.up_proj.weight	Size: torch.Size([768, 256])
model.layers.0.mlp.down_proj.weight	Size: torch.Size([256, 768])
中间省略...	
model.layers.3.input_layernorm.weight	Size: torch.Size([256])
model.layers.3.post_attention_layernorm.weight	Size: torch.Size([256])
model.norm.weight	Size: torch.Size([256])
lm_head.weight	Size: torch.Size([32000, 256])

Total Parameters: 19532032 (19.5 M) 20M个参数

可以看到，我们的模型只有不到 20M！非常非常小，并且其中 Embedding 占了大头。

尽管模型还没有训练，但我们仍然可以测试一下推理：

```
def inference(
    model: AutoModelForCausalLM,
    tokenizer: AutoTokenizer,
    input_text: str = "Once upon a time, ",
    max_new_tokens: int = 16
):
    inputs = tokenizer(input_text, return_tensors="pt").to(device)
    outputs = model.generate(
        **inputs,
        pad_token_id=tokenizer.eos_token_id,
        max_new_tokens=max_new_tokens,
        do_sample=True,
        top_k=40,
        top_p=0.95,
        temperature=0.8
    )
    generated_text = tokenizer.decode(
        outputs[0],
        skip_special_tokens=True
    )
    # print(outputs)
    print(generated_text)

inference(model, tokenizer)

"""
Once upon a time, Hostu crimeine /\ könnenlinewidth measurementresol perfectly
"""
```

嗯，的确是胡言乱语呢，不过可以正常推理，说明模型没问题！

但现在模型是随机初始化的，为了让模型更好地收敛，我们最好给模型一个更好的初始化方法，我这里选用 Kaiming 初始化，比较适用于 ReLU 类的激活，当然也可以选用高斯初始化、Xavier 初始化等等。

```
# Kaiming 初始化
def kaiming_initialization(model):
    for name, param in model.named_parameters():
        if 'weight' in name and param.dim() > 1:
            torch.nn.init.kaiming_uniform_(param, mode='fan_in', nonlinearity='')
```

```

        elif 'bias' in name:
            # 一般偏置项可以初始化为 0
            torch.nn.init.constant_(param, 0)

kaiming_initialization(model)

```



现在，我们的模型真正初始化完成了！如果你愿意，可以先将这个初始化好的模型保存到本地，用 `save_pretrained()` 即可。

4. 数据集

让我们继续！

4.1 加载数据集

我们接下来需要从 Hugging Face 加载数据集，我这里是建立在网络畅通的基础上的，如果你没有用 Colab 或者网络无法直连 Hugging Face，那么也可以先下载到本地某个文件夹中，

`load_dataset` 也可以直接读取本地文件夹。我们要用的数据集路径如下：

[noanabeshima/TinyStoriesV2 · Datasets at Hugging Face](#)

```

# 加载数据集
from datasets import load_dataset

dataset_name_or_path = "noanabeshima/TinyStoriesV2" # 可以替换为本地文件夹

# ds_train = load_dataset(dataset_name_or_path, split='train') # 取全部数
ds_train = load_dataset(dataset_name_or_path, split='train[:10%]') # 只取前 1
ds_val = load_dataset(dataset_name_or_path, split='validation')

print(ds_train)
print(ds_val)

"""
Dataset({
  features: ['text'],
  num_rows: 271769
})
Dataset({
  features: ['text'],
  num_rows: 27629
})
"""

```

我们来看看数据长什么样子：

```

# 查看前两条
print(ds_train[:2])

"""
{'text': ['Once upon a time, there was a reliable otter named Ollie. He lived i
'One day, a little boy named Tim went to the park. He saw a big tiger. The ti
"""

```

这里需要注意，datasets 加载后的数据是 `Dict[str, List[str]]` 的形式的，并非 `List[Dict[str, str]]`。

```

# 数据长这样
{

```

```

        'text': [
            <text_1>, <text_2>, <text_3>, ...
        ]
    }
}

```

不是下面这样

```

[
    {'text': <text_1>},
    {'text': <text_2>},
    {'text': <text_3>},
    ...
]

```



4.2 数据预处理

接下来，我们要将数据预处理一下，也就是用 tokenizer 进行 tokenize。让我们来写一个处理函数：

```

from typing import Dict, List

def process_func(
    examples: Dict[str, List]
) -> Dict[str, List]:
    max_token = 2048    # 设置最长 token 数目，对于我们当前任务，2048 绝对不会超

    encoded_texts = tokenizer(examples['text'], add_special_tokens=False)
    input_ids_list = encoded_texts['input_ids']

    new_input_ids_list, new_attn_mask_list = [], []
    for input_ids in input_ids_list:
        temp = input_ids[-max_token+1:] + [tokenizer.eos_token_id]
        new_input_ids_list.append(temp)
        new_attn_mask_list.append([1] * len(temp))
    return {
        "input_ids": new_input_ids_list,
        "attention_mask": new_attn_mask_list
    }

```

我们来解析一下其中的一些点：

tokenizer 的 encode

```
encoded_texts = tokenizer(examples['text'], add_special_tokens=False)
```

根据前面的示例，我们知道这里 examples['text'] 其实是一个 List[str]，当一个 List 传入 tokenizer() 时，tokenizer 会自动进行 batch encode，得到的是 {'input_ids': List[int], 'attention_mask': List[int]}（当然，如果设置了 return_tensors='pt' 就会得到 Tensor）。

add_special_tokens=False 则是让 tokenizer 不要加上特殊 token，在 LLaMA 中就是不会在句首加上 bos_token <s>。

```

text = 'Hello, world!'

tokenizer(text)
# {'input_ids': [1, 15043, 29892, 3186, 29991], 'attention_mask': [1, 1, 1, 1, 1]}
tokenizer(text, add_special_tokens=False)
# {'input_ids': [15043, 29892, 3186, 29991], 'attention_mask': [1, 1, 1, 1]}
# 上面多了一个 1，即 tokenizer.bos_token_id，在 LLaMA 中对应的就是 <s>

```



填充还是截断？

```
max_token=2048
temp = input_ids[-max_token+1:] + [tokenizer.eos_token_id]
new_input_ids_list.append(temp)
new_attn_mask_list.append([1] * len(temp))
```

截取后max_token-1位

在这里，我采用直接截断的方式，最大截取当前输入序列的后 (max_token - 1) 位，再加上一个 eos_token_id，组成总长度不超过 max_token 的序列。attention_mask 的长度保持一致，全为 1。

这里利用到了 list 的切片特性，input_ids[-max_token+1:] 可以获取 min(max_token, len(input_ids)) - 1 的序列。

当然，也可以采取将超出长度部分再按照 max_token 来分块，重新组装。

应用在所有数据上

接下来，我们用 map() 函数，来将 process_func() 应用到 ds_train 和 ds_val 中的每个样本：

```
num_proc = 8 # 处理数据时所用的线程数

ds_train = ds_train.shuffle() # 训练集打乱一下

ds_train = ds_train.map(
    process_func,
    batched=True,
    num_proc=num_proc,
    remove_columns=ds_train.column_names,
    desc='Running tokenizer on train_set: '
)
ds_val = ds_val.map(
    process_func,
    batched=True,
    num_proc=num_proc,
    remove_columns=ds_val.column_names,
    desc='Running tokenizer on val_set: '
)

print(ds_train)
print(ds_val)

"""
Dataset({
  features: ['input_ids', 'attention_mask'],
  num_rows: 271769
})
Dataset({
  features: ['input_ids', 'attention_mask'],
  num_rows: 27629
})
"""
```

数据预处理成功！

4.3 数据批处理——DataCollator

4.3.1 两行代码

我们在训练的时候往往不会一条一条训练，而是成批次地训练，那么我们就需要对数据做批处理。因此我们需要用到 transformers 中的一个工具系列——[DataCollator](#)。

既然是预训练，那么就是让模型在语料上做无监督学习，也就是我们熟知的 next token prediction，即根据前面的所有输入来预测下一个 token，然后把新的 token 拼接在已有输入上作为下一输入，如此往复，直到触发停止设定（例如触发 `max_new_tokens`）。

所以我们的训练目标——或者说是 label——显而易见，就是把输入偏移一位当作预测目标，我们计算的就是输出和这个目标之间的 loss：

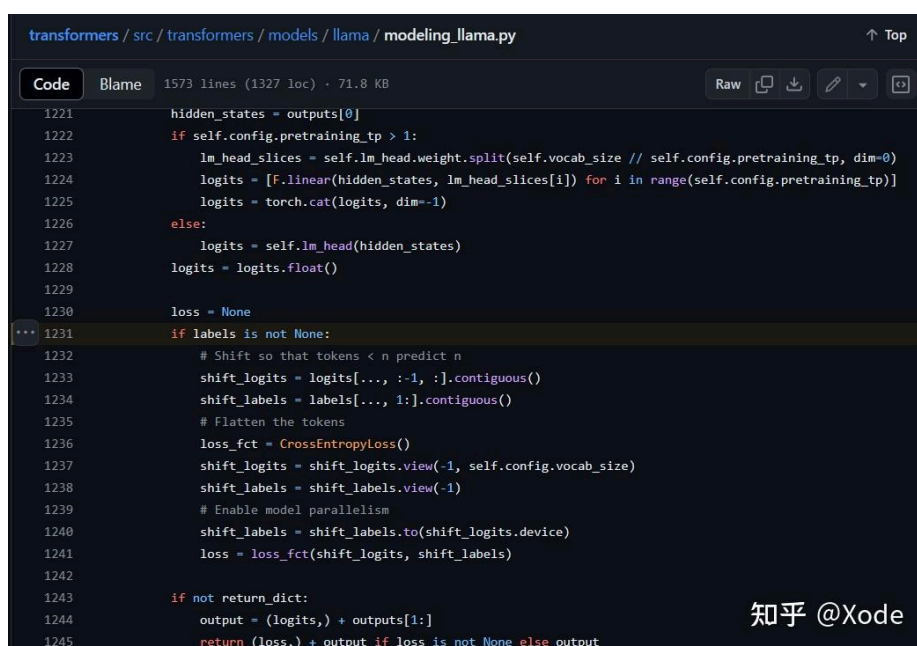
```
... once upon a time there was ...      # input
once upon a time there was ... ..      # label

# label 错开一位，是 input 的下一预测目标，计算的就是 input 和 label 之间的 loss
```

所以我们只需要把 `input_ids` 复制一份、再偏移一位，就可以作为 labels 了。

……再等等，让我们看看这条问题：[Shifting ids to the right when training GPT-2 on text generation? - Beginners - Hugging Face Forums](#)：

这里 `sgugger` 提到，在 Hugging Face 的实现里，training 时已经实现好了偏移一位的逻辑，不需要我们再手动实现了。我们也可以在 transformers 的源码里看到这一点，例如 [LLaMA 的实现](#)。

The image shows a snippet of Python code from the transformers library, specifically from the file `transformers/src/transformers/models/llama/modeling_llama.py`. The code is displayed in a dark-themed editor with line numbers on the left. The relevant section starts at line 1231, where an `if labels is not None:` block is entered. Inside this block, the code shifts the logits and labels to the right by one position using `shift_logits = logits[..., :-1, :].contiguous()` and `shift_labels = labels[..., 1:].contiguous()`. It then flattens the tokens and calculates the loss using `CrossEntropyLoss()`. The final line of the block is `return (loss,) + output if loss is not None else output`. The code is well-commented, explaining the purpose of each step. The file name and line numbers are visible at the top of the code block.

```
transformers / src / transformers / models / llama / modeling_llama.py
Code Blame 1573 lines (1327 loc) · 71.8 KB
1221 hidden_states = outputs[0]
1222 if self.config.pretraining_tp > 1:
1223     lm_head_slices = self.lm_head.weight.split(self.vocab_size // self.config.pretraining_tp, dim=0)
1224     logits = [F.linear(hidden_states, lm_head_slices[i]) for i in range(self.config.pretraining_tp)]
1225     logits = torch.cat(logits, dim=-1)
1226 else:
1227     logits = self.lm_head(hidden_states)
1228     logits = logits.float()
1229
1230 loss = None
1231 if labels is not None:
1232     # Shift so that tokens < n predict n
1233     shift_logits = logits[..., :-1, :].contiguous()
1234     shift_labels = labels[..., 1:].contiguous()
1235     # Flatten the tokens
1236     loss_fct = CrossEntropyLoss()
1237     shift_logits = shift_logits.view(-1, self.config.vocab_size)
1238     shift_labels = shift_labels.view(-1)
1239     # Enable model parallelism
1240     shift_labels = shift_labels.to(shift_logits.device)
1241     loss = loss_fct(shift_logits, shift_labels)
1242
1243 if not return_dict:
1244     output = (logits,) + outputs[1:]
1245     return (loss,) + output if loss is not None else output
```

transformers 中 LLaMA 的实现，已经处理好了 labels 的偏移

所以，我们只需要将 `input_ids` 直接复制一份作为 labels 即可。

那么怎么做呢？我们可以用 `DataCollatorForLanguageModeling`，并设置 `mlm=False`：

```
from transformers import DataCollatorForLanguageModeling

data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)
```

两行代码，非常简单！

不过我们可以稍微多讲一点，这个 data collator 是如何发挥作用的？为什么选的是它而不是在微调中更常见的 `DataCollatorForSeq2Seq`？

4.3.2 More things……

实际上，就像 `mlm` 这个参数所显示的一样，`DataCollatorForLanguageModeling` 一开始其实是设计给 Bert 的 MLM 任务的。MLM 任务就是 Masked Language Modeling，掩码语言建模，

就是在一整个序列中挑选一部分 token 用 [mask] 给盖住，让模型去根据上下文预测被盖住的是什么 token。

```
# MLM 任务示意
今天早上下 [mask] 了      # input
今天早上下 雨 了         # label
```



可以看到，这样建立的 label 就是原来的 input 的 copy，它是将 input 中随机 mask 一部分，label 不变。

这几乎就是我们想要的——只是不需要 mask。所以，我们设置 `mlm=False` 后，就可以直接得到 `input_ids` 的 copy 了。

让我们继续看看 `DataCollatorForLanguageModeling` 怎么作用的：

```
# DataCollatorForLanguageModeling
# 这里的 tokenizer 选用的是 Qwen1.5 的，并非 LLaMA 的，只是做一个示意
dc = DataCollatorForLanguageModeling(tokenizer, mlm=False)
data = ['南京', '南京市', '南京市长江']

raw_tokens = [tokenizer(text) for text in data]

print(f'tokenizer.pad_token_id: {tokenizer.pad_token_id}\n')
print(dc(raw_tokens))

"""
tokenizer.pad_token_id: 151643

{
  'input_ids': tensor([[151643, 151643, 102034],
                      [151643, 151643, 112891],
                      [102034, 102975, 69177]]),
  'attention_mask': tensor([[0, 0, 1],
                           [0, 0, 1],
                           [1, 1, 1]]),
  'labels': tensor([[ -100,   -100, 102034],
                   [ -100,   -100, 112891],
                   [102034, 102975, 69177]])
}
"""
```

可以看到：

- 由于长度不一，所以 data collator 做了 padding，padding 的方向就是我们 3.2 中提到的 `tokenizer.padding_size`
- labels 确实是 input_ids 的原位复制，区别在于 input_ids 里用 `pad_token_id` 来填充，labels 里对应的是 -100、表示不计算 loss

那么微调里常用的 `DataCollatorForSeq2Seq` 又是如何作用的呢？我们仍然用刚刚的数据例子：

```
# DataCollatorForSeq2Seq
# 这里的 tokenizer 选用的是 Qwen1.5 的，并非 LLaMA 的，只是做一个示意
dc = DataCollatorForSeq2Seq(tokenizer)
data = ['南京', '南京市', '南京市长江']

raw_tokens = [tokenizer(text) for text in data]

print(f'tokenizer.pad_token_id: {tokenizer.pad_token_id}\n')
print(dc(raw_tokens))

"""
tokenizer.pad_token_id: 151643

{
```

```

        'input_ids': tensor([[151643, 151643, 102034],
                             [151643, 151643, 112891],
                             [102034, 102975, 69177]]),
        'attention_mask': tensor([[0, 0, 1],
                                   [0, 0, 1],
                                   [1, 1, 1]])

    # 注意没有 labels 字段
}
"""

```



可以发现，`DataCollatorForSeq2Seq` 和 `DataCollatorForLanguageModeling` 一样，做了批处理和 padding，但是没有标签 labels。原因是：`DataCollatorForSeq2Seq` 设计之初用于的任务和它的名字一样，是序列到序列（seq2seq）任务，放到文本任务上，就是要有两个 seq：输入 text 和输出 label，比如下面的例子：

```

# DataCollatorForSeq2Seq
# 这里的 tokenizer 选用的是 Qwen1.5 的，并非 LLaMA 的，只是做一个示意
dc = DataCollatorForSeq2Seq(tokenizer, padding=True)
data = [('南京', '市长江大桥'), ('南京市', '长江大桥'), ('南京市长江', '大桥')]

features = []
for text, label in data:
    feature = tokenizer(text)
    feature['labels'] = tokenizer(label)['input_ids']
    features.append(feature)

print(f'tokenizer.pad_token_id: {tokenizer.pad_token_id}\n')
print(dc(features))

"""
tokenizer.pad_token_id: 151643

{
  'input_ids': tensor([[151643, 151643, 102034],
                       [151643, 151643, 112891],
                       [102034, 102975, 69177]]),
  'attention_mask': tensor([[0, 0, 1],
                            [0, 0, 1],
                            [1, 1, 1]]),
  'labels': tensor([[102975, 69177, 106936],
                    [ -100, 104924, 106936],
                    [ -100,  -100, 106936]])
}
"""

```

可以看到：

- `DataCollatorForSeq2Seq` 需要指定当前输入的文本和后面需要生成的文本，即 text 和 label，如果像 `DataCollatorForLanguageModeling` 那样处理会得不到 labels 字段
- 因此，`DataCollatorForSeq2Seq` 适合有监督微调（SFT），输入是 text，输出是 label，非常合理

5. 超迷你 LLaMA，启动！

5.1 配置训练参数

我们需要用到 transformers 的 `TrainingArguments` 来配置训练参数，具体参数说明可以[看这里](#)。

```

from transformers import TrainingArguments

training_args = TrainingArguments(

```

```

output_dir='saves',
overwrite_output_dir=True,
do_train=True,
do_eval=True,
eval_steps=1000,
per_device_train_batch_size=4,
gradient_accumulation_steps=1,
learning_rate=1e-4,
lr_scheduler_type='cosine',
bf16=torch.cuda.is_bf16_supported(),
fp16=not torch.cuda.is_bf16_supported(),
logging_steps=50,
report_to=None,
num_train_epochs=2,
save_steps=1000,
save_total_limit=2,
seed=3407
)
# 输出路径, 包括模型检查点、中间文件:
# 是否覆写 output_dir
# 是否做训练
# 是否做评估
# 评估步骤间隔
# 每设备批次
# 梯度累计步大小, 省显存, 但小模型没:
# 学习率大小
# 学习率调度策略, LLM 训练一般都用余
# 尝试配置 bf16
# bf16 不行就上 fp16
# 打印步骤间隔
# 日志输出目标, 不想用 wandb 可以设
# 训练轮数, 2 ~ 3 即可
# 检查点保存步骤间隔
# output_dir 内留存的检查点最大数E
# 随机种子

```

如果你之前用了 wandb , 现在想禁用掉, 可以设置环境变量:

```

import os

os.environ['WANDB_DISABLED'] = 'true'

```

5.2 配置 Trainer

同样地, 具体参数说明可以[看这里](#)。

```

from transformers import Trainer

trainer = Trainer(
    model=model,                # 模型实例
    args=training_args,        # 训练参数
    train_dataset=ds_train,     # 训练集
    eval_dataset=ds_val,        # 验证集 (评估集)
    tokenizer=tokenizer,        # 分词器
    data_collator=data_collator, # data collator
)

```

5.3 训练与保存

配置好 Trainer 后, 通过下列代码即可启动训练:

```
trainer.train()
```

接下来只需要等待训练完成。我用一个半小时训练了 2 epochs, loss 达到了 1.6 左右。

训练完成后, 如果用的是 jupyter notebook, 那么此时 model 已经是训练好的状态了。我们可以再次推理试试看:

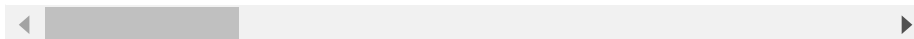
```

inference(
    model,
    tokenizer,
    "Once upon a time, in a beautiful garden, there lived a little rabbit named
    max_new_tokens=256
)

```

得到如下结果:

Once upon a time, in a beautiful garden, there lived a little rabbit named Pete. One day, Robby saw a big box in his yard. He was curious and wanted to know what Timmy and Hopper went to find the big box. They found a key under a tree. They



可以看到：

- 20M 模型确实能够流畅续写故事了
- 20M 模型写出的故事的语法、流畅度都不错，但是一致性欠佳，特别是故事主题、人名的前后连贯性不高

总之，这个超迷你 LLaMA 3 确实训练完成了！我们可以将它保存到本地：

```
model_path = '...'

model.save_pretrained(model_path)
```

也可以推送到 Hugging Face：

```
from huggingface_hub import notebook_login

repo_name = 'TinyStories-LLaMA2-20M-256h-4l-GQA'

notebook_login()    # 输入 Access Tokens

model.push_to_hub(repo_name)
tokenizer.push_to_hub(repo_name)
```

6. 结尾

这次尝试用 Trainer 来做一个模型的预训练，以往都是用 Trainer 来做微调，这次也算是学习了一下吧。TinyStories 这个工作之前就关注过，但一直没顾上来复现一下，这次也算是简单复现了个小模型出来，和原工作的丰富度确实是比不了，但也算完成一个 todo。

后面这个小模型可以继续做 SFT，也就是做指令微调，可以和原工作一样，给定故事背景、关键词、开头让小模型续写，也可以迁移到别的任务。不过由于我们的预训练任务只针对了讲短篇故事这一类任务，加上参数又特别少，如果直接迁移其它指令任务估计表现不会很好。

本篇文章也是希望尽可能写得详细一点，欢迎大家交流。

致谢 - Reference

1. [LLM大模型之Trainer以及训练参数 - 知乎 \(zhihu.com\)](#)：详细说明了 Trainer 与 TrainingArguments 的参数，这位博主的其他文章也非常好，值得拜读
2. [NLP（九十四）transformers模块中的DataCollator - 知乎 \(zhihu.com\)](#)：详细介绍了 DataCollator，并配上了示例，通俗易懂
3. [stanleyslxl/llms_tool: 一个基于HuggingFace开发的大语言模型训练、测试工具。支持各模型的webui、终端预测，低参数量及全参数模型训练\(预训练、SFT、RM、PPO、DPO\)和融合、量化。\(github.com\)](#)：借鉴了里面采用 DataCollatorForLanguageModeling 来做预训练 data collator 的思路，这个仓库也写得非常工整，值得细读

编辑于 2024-06-09 01:08 · IP 属地北京

内容所属专栏



LLM from scratch
LLM相关的从零实现的笔记

订阅专栏

[LLM（大型语言模型）](#) [llama](#) [预训练](#)