baby-llama2-chinese 实践笔记

baby-llama2-chinese 项目，演示了如何从零训练一个参数量 500M-1B 的 LLM。该项目完整包含了训练、SFT指令微调、奖励模型等过程，是一个很好的教学项目。

环境搭建：首先从 GitHub 上 clone 项目。相关依赖包可参见 requirements.txt 。

在本文中，记录了我对该项目的自学过程。该过程同时参考了《jjchen1 》的高质量笔记。

> Note
>
> 文中提到一些预训练语料，我统一收集至笔记《大模型数据集收集》中。

下载运行预训练微调模型

我先从预训练模型下手。作者同时提供了 base 和微调模型，我选择 Llama2-Chinese-218M-v1-NormalChat 。将下载好的模型放置到项目的 out 目录下。

修改 eval.py 中的 ckpt_path：

```
ckpt_path = "out/baby-llama2-chinese/Llama2-Chinese-218M-v2/sft_model_norm_epoch5.pth"
```

注意参数要对上：

```
max_seq_len=1024
dim=1024
n_layers=12
n_heads=8
```

运行 eval.py ：

```
(.venv) [maxiee@archlinux baby-llama2-chinese]$ python eval.py
[prompt]: 最近我在办公室坐久了会感到头晕，请问这是什么原因?有什么缓解办法吗？
[answer]:  我无法确定您所描述的问题，因此无法回答。是一种完成的工作或任务，您需要提供更多信息或采取任何可能蛋白质时间或建议来帮助您处理。如果您需要我提供信息，请告诉我您需要答案指出了哪些原因。
---------------
[prompt]: 前列腺囊肿的症状是什么？
[answer]:  您是否对您所推荐购买的决策具体建议？如果可以的话，请提供更多信息。
---------------
[prompt]: 请问，世界上最大的动物是什么？
```

[answer]: 世界上最大的动物是蓝鲸。蓝鲸段头鲸，也称为蓝鲸河见鲸，是雨海、海洋中的垃圾袋鲸，是大型哺乳动物，身长约为40米，体重超过30000公斤。

2 jjchen1 笔记

我是从 jjchen1 大佬的笔记中得知 baby-llama2-chinese 项目的。jjchen1 的笔记是一个为期 30 天的 LLM 的探索，这里摘录与本文项目相关的文章，其他文章同样值得推荐。

Day 6 - Baby LLama2 ：

預訓練語料的預處理採用GPT的通用做法，對語料進行提前分詞，並將所有訓練語料拼接成一個數組以二進制格式存儲到磁盤上。
SFT樣本的構建需要注意質量，並且需要花費時間來清洗數據，以獲得更好的SFT效果。
預訓練和SFT的腳本已提供，並可以根據自身的算力來調整參數，控制模型的計算量和參數量。
Day 7 - Baby LLama2 Chinese
使用 medical dataset 进行语料准备
使用 pretrain.py 进行训练：修改網路的大小，把sampler改成非分散式
Day 8 - Baby LLama2 Chinese (2)
Day 9 - Baby LLama2 Chinese (3)
完成预训练，练了 10 个 epoch
进行 Supervised Fine Tuning (SFT)
出现 OOM，缩小 batch_size
Day 10 - Baby LLama2 Chinese (4)
第一版模型 Bug：eval 样例与 SFT 重复，高估问答能力。加大模型后效果逐步提升。
完成了7個epoch的sft訓練以後，結果差的慘不忍睹，开始 Debug
pretrain 做的事情非常簡單，就是純粹訓練語言模型，把一段文字喂給模型，一直要求他預測位移一個字以後的tokens。
以這個階段做的任務來看，我認為只是讓模型學會認字，理論上應該對SFT後的成果影響不會這麼大才對(雖然我看一篇文章寫pretrain佔了整個訓練總運算量的98%以上)
sft的訓練跟前一階段很類似，一樣是用語言模型的方式訓練，只是這個階段的輸入文章由 問題 與 答案 兩個段落組成。
Day 11 - Baby LLama2 Chinese (5)
medical dataset语料：son檔中每個 line 都是一個json物件，key 和 text
pretrain資料前處理流程：
讀取一個 .json 檔，
把其中每一個樣本的text轉成對應的tokens，
在tokens結尾加上一個 <eos> 的token，
之後全部串起來成為一個很長的陣列，
最後把陣列型態轉換成uin16(因為字典長度64793 < 2^16)，
寫入到一個 .bin 檔。
限制每個樣本的token總數 > 5，否則丟棄

讀取 datasets
讀取多個.bin檔內預處理好的uint16陣列doc_ids
data=多個.bin檔內的doc_ids串在一起
刪除陣列尾部資料，確保 len(data) % max_length = 0
將超長的陣列data切割成多個訓練樣本
pretrain 階段 Debug
在pretain.py中印出X,Y發現竟然有一些是亂碼，可能是在前處理的地方出了問題。
Day 12 - Baby LLama2 Chinese (6)
data_process.py 处理百度数据集时，内存不够，主要是 json（注：可改用python使用ijson读取大文件json文件 - 知乎 ）
Day 13 - Baby LLama2 Chinese (7)
medical_book_zh.json 乱码来自语料
Day 14 - Baby LLama2 Chinese (8)
SFT 语料里面有两种格式，A Instruction + Output，B Instruction + Input + Output
代码中统一为 prompt 與 answer 。
Day 15 - Baby LLama2 Chinese (9)
除了 eos token以外SFT額外需要一個 bos token來將 prompt 與 answer 切分開；並且在讀取訓練資料時，必須要檢查每筆資料的長度，如果 prompt 跟 answer 分別超過對應的長度限制，必須將多出來的字去除。作者：這邊刪除超出長度資料的方法我覺得有些疑惑
loss_mask 的index為什麼是對應 X 而不是 Y
Day 16 - Baby LLama2 Chinese (10)
SFT訓練與Loss計算
Day 17 - Baby LLama2 Chinese (11)
模型訓練進展很慢
每經過一定的iteration就儲存一次最後的 checkpoint ，否則模型訓練一個epoch實在太久，必須要能夠resume上次的訓練
Day 18 - Baby LLama2 Chinese (12)
Day 19 - Baby LLama2 Chinese (13)
把 sft.py 也改成了可以resume training
可以歸納出 pretrain 與 sft 在訓練的程式上的不同點：
Load Data
Dataset類別 PretrainDataset vs SFTDataset
前處理檔案 ./data/pretrain_data.bin vs ./data/sft_data.csv
dataloader讀取資料 (X,Y) vs (X,Y,loss_mask)
Loss計算
pretrain直接使用 raw_model.last_loss
sft額外使用 loss_mask 計算loss
Day 20 - Baby Llama2 Chinese (13)

读完笔记后，我相当地佩服 jjchen1 大佬！

4 开始预训练！

接下来，该我出场了，在大佬们的指点下，我也准备走一遍这个流程。

baby-llama2-chinese 作者提供了加工好的语料，共计634亿Tokens的数据量，文件大小为118.18GB。我直接复用这个语料。

为了加快训练速度，减小模型参数：

max_seq_len=512
dim=512
n_layers=4
n_heads=4

相较于 92M 模型，层数由8减为4，多头注意力由8减为4。

修改了一下语料加载代码，一次性加载目录下所有 .bin：

```
data_root_path = "/mnt/ssd2/home/maxiee/LLM/baby-llama2-chinese/pretrain_data_bin"
# 搜索 data_root_path 下所有 .bin 文件，保存至 data_path_list
data_path_list = []
for root, dirs, files in os.walk(data_root_path):
    for file in files:
        if file.endswith(".bin"):
            print(os.path.join(root, file))
            data_path_list.append(os.path.join(root, file))
```

jjchen1 笔记中有提到： memmap=False ，防止内存干爆，我实测了以下：

如果为 False，会将语料全部加载到内存中，因为我用的是完整的 118.18GB，导致开头部分全在读取语料
改为 True 后，会立刻开始预训练，

```
train_ds = PretrainDataset(data_path_list, max_length=max_seq_len, memmap=False)
```

执行预训练，我是单卡：

```
torchrun --standalone pretrain.py
```

效果赏析（挑选一些看起来好一点的）：

```
Enter a question: 信念不是到处去
[prompt]: 信念不是到处去
[answer]: 打工，反而能像我一样学习
```

```
[prompt]: 什么叫作失败?失败是
[answer]: 必须的。
```
你为什么认为胜利呢？我为什么要提出胜利或失败，你为什么觉得败，你为什么要失败。成功不是为了失败和失败，但当一个人遇到成功时，那人可能不是他自己的命运，而是当一个人去。结果就失败了。结果就是失败了。他不能坚持失败。于是又把胜利归告了。这样，失败就了。结果就败了。。

```
[prompt]: 得到智慧的惟一办法，就是用
[answer]: 我们的智慧和智慧去智慧。。主要产品：《智障》系列图书简介：【系列构成】智障系列由北京智障科技股份有限公司、深圳智障科技发展有限公司倾力打造，是中国智障科技旗下专业研究人员，是智障科技旗下公司运营的专业研究团队。也是公司所有咨询专家和顾问。智障人从咨询、咨询、咨询到技术咨询；到品牌规划：提供、指导、执行，帮助客户实现营销策略。。编辑推荐：
```

```
[prompt]: 我从来不怕得罪朋友，因为
[answer]: 我是大人物。
```

预训练代码阅读

训练大模型的第一步是数据处理（ data_process.py ），第二步是正式预训练（ pretrain.py)。本节中阅读正式预训练（ pretrain.py)部分的代码。

首先，有众多超参数：

参数	说明	理解
out_dir	输出目录，用于保存训练过程中生成的模型检查点和日志。	
max_epoch	最大训练轮次。每个轮次都会遍历一次整个训练数据集。	
eval_interval	评估间隔，每训练 eval_interval 轮后，模型会在验证集上进行评估。	
log_interval	日志间隔，每训练 log_interval 轮后，训练的进度和性能指标会被记录下来。	
save_interval	保存间隔，每训练 save_interval 轮后，模型的当前状态会被保存下来。	
eval_iters	评估迭代次数，定义了在每次评估过程中，模型需要处理的数据批次数量。	
eval_only	如果为True，脚本在第一次评估后就会退出。	
always_save_checkpoint	如果为True，每次评估后都会保存一个模型检查点。	
init_from	模型初始化方式，可以从头开始（'scratch'），从上次的检查点恢复（'resume'），或者从预训练的GPT2模型开始（'gpt2*'）。	

参数	说明	理解
gradient_accumulation_steps	梯度累积步数，用于模拟更大的批次大小。	
batch_size	批次大小，如果 gradient_accumulation_steps 大于1，这就是微批次的大小。	
max_seq_len	序列的最大长度，即模型可以处理的输入序列的最大长度。	
dim	模型的维度，即模型内部表示的大小。	
n_layers	Transformer模型的层数。	
n_heads	Transformer模型的注意力头的数量。	
multiple_of	用于确保序列长度是某个数的倍数，这对某些硬件加速器（如GPU）的性能优化很重要。	
dropout	Dropout率，用于防止模型过拟合。在预训练阶段，通常设置为0，在微调阶段，可以尝试设置为0.1或更高。	
bias	是否在LayerNorm和Linear层中使用偏置。	
learning_rate	学习率，决定了模型参数在每次更新时的调整幅度。如果学习率过大，训练可能会不稳定；如果学习率过小，训练可能会过慢。	
weight_decay	权重衰减，用于防止模型过拟合。它通过在优化器的更新规则中添加一个与当前参数值成比例的项来实现。	
beta1， beta2	这是Adam优化器的超参数，用于计算梯度和梯度平方的移动平均值。	
grad_clip	梯度裁剪值，用于防止梯度爆炸。如果梯度的范数超过这个值，就会被裁剪。	
decay_lr warmup_iters lr_decay_iters min_lr	这些变量用于配置学习率衰减策略。 decay_lr 决定是否使用学习率衰减， warmup_iters 是预热步数，在这些步数内，学习率会线性增加到 learning_rate ， lr_decay_iters 是学习率开始衰减的步数， min_lr 是学习率的最小值。	
backend	分布式数据并行（DDP）的后端，可以是'nccl'，'gloo'等。	
device	训练设备，可以是'cpu'，'cuda'，'cuda:0'，'cuda:1'等。	
dtype	dtype : 数据类型，可以是'float32'，'bfloat16'，或'float16'。	如果选择'float16'，将自动实现梯度缩放，这对于防止浮点数精度问题很有用。

参数	说明	理解
compile	是否使用PyTorch 2.0编译模型以提高速度。	

该脚本首先判断是否运行于分布式环境下（由 ddp 变量标识），这里我主要看单 GPU 的场景。

接下来计算每次迭代处理的 token 数量：

```
tokens_per_iter =
   # 梯度累积步数。
   gradient_accumulation_steps \
   # 分布式数据并行世界大小
   * ddp_world_size \
   # 批次大小
   * batch_size \
   # 序列的最大长度
   * max_seq_len
```

其中：

gradient_accumulation_steps : 梯度累积步数。在每个步骤中，模型都会处理一个微批次的数据，并计算出梯度。然后，这些梯度会被累积起来，直到处理
完 gradient_accumulation_steps 个微批次后，才会用来更新模型的参数。这样可以模拟更大的批次大小，而不需要增加内存消耗。（本文中未开启）
ddp_world_size : 分布式数据并行（Distributed Data Parallel，简称 DDP）的世界大小，即参与训练的进程数量。在 DDP 中，每个进程都会处理一部分数据，并计算出梯度。然后，这些梯度会被平均，得到最终用于更新模型参数的梯度。（本文中未开启）
batch_size : 批次大小，即每个进程在每个步骤中处理的数据数量。
max_seq_len : 序列的最大长度，即模型可以处理的输入序列的最大长度。

在我的场景下，就是 max_seq_len * batch_size ，也就是一次喂给 GPU 的 Token 数量，即迭代（iterator）。

继续进行设置：

```
# 设置了 PyTorch 的随机数生成器的种子。
torch.manual_seed(1337 + seed_offset)

# 启用了 TensorFlow 32位浮点数（TF32）格式。
# TF32 是 NVIDIA 在其 Ampere 架构的 GPU 中引入的一种新的浮点数格式
# 它可以在不牺牲太多精度的情况下提高计算速度。
torch.backends.cuda.matmul.allow_tf32 = True  # allow tf32 on matmul
torch.backends.cudnn.allow_tf32 = True  # allow tf32 on cudnn
```

```
# 设置了设备类型为 "cuda"，表示将在 GPU 上进行计算。
device_type = "cuda"

# 根据 `dtype` 的值选择了一个 PyTorch 的数据类型。
# 如果 `dtype` 是 "float32"，`ptdtype` 就会被设置为 `torch.float32`；
# 如果 `dtype` 是 "bfloat16"，`ptdtype` 就会被设置为 `torch.bfloat16`；
# 如果 `dtype` 是 "float16"，`ptdtype` 就会被设置为 `torch.float16`。
# note: float16 data type will automatically use a GradScaler
ptdtype = {"float32": torch.float32, "bfloat16": torch.bfloat16, "float16": torch.float16}[dtype]

# 如果 `device_type` 是 "cuda"，`ctx` 就会被设置为 `torch.cuda.amp.autocast()`
# 这是一个自动混合精度（Automatic Mixed Precision，简称 AMP）的上下文管理器
# 它可以自动选择合适的数据类型进行计算，以提高计算速度和减少内存消耗。
ctx = (
    nullcontext()
    if device_type == "cpu"
    else torch.cuda.amp.autocast()
)
# 用于存储最佳的验证损失
# 初始值被设置为一个非常大的数
# 这样在训练过程中只要遇到一个比这个数小的验证损失，就会更新
best_val_loss = 1e9
```

下面是加载处理后的 Token 语料，我对源代码做了修改，能够自动遍历目录下所有语料文件（.bin）：

```
data_root_path = "/mnt/ssd2/home/maxiee/LLM/baby-llama2-chinese/pretrain_data_bin"
# 搜索 data_root_path 下所有 .bin 文件，保存至 data_path_list
data_path_list = []
for root, dirs, files in os.walk(data_root_path):
    for file in files:
        if file.endswith(".bin"):
            print(os.path.join(root, file))
            data_path_list.append(os.path.join(root, file))
```

接下来创建预训练数据集类，以及训练采样器：

```
# 自定义的数据集类，它从 `data_path_list` 中的文件中加载数据
# 每个数据样本的最大长度为 `max_seq_len`。
# 如果 `memmap` 参数为 `True`，则使用内存映射文件来加载数据，这可以减少内存消耗。
train_ds = PretrainDataset(data_path_list, max_length=max_seq_len, memmap=True)

# DistributedSampler 是 PyTorch 的一个工具，用于在分布式训练中对数据进行采样。
train_sampler = torch.utils.data.distributed.DistributedSampler(train_ds)
```

```
# DataLoader 是 PyTorch 的一个工具，用于在训练过程中加载数据。
# 它可以自动地将数据分成多个批次，并在需要时加载每个批次的数据。
# 这里的参数设置如下：
train_loader = torch.utils.data.DataLoader(
    train_ds,
    batch_size=batch_size,
    pin_memory=False,
    drop_last=False,
    shuffle=False,
    num_workers=0 if os.name == 'nt' else 4,
    sampler=train_sampler
)
```

其中，在 PretrainDataset 部分，jjchen1 在 Day 13 - Baby LLama2 Chinese (7) 中有提及：原因是， data_path_list 是个列表，允许传入多个语料文件。但是，当采用 memmap=True 时，PretrainDataset 只会用 np.memmap 加载第一个文件。这里，根据源作者提供的处理后语料能够看出，他是将所有原始语料集拼接为一个语料文件进行训练。

当 memmap=False 时，需要将所有语料 .bin 都加载到内存中，这是非常占用内存的。并且在语料集非常大时，这一步耗时也很长，需要等待一段时间才能开始训练。

shuffle 被设置为 False ，这意味着数据不会在每个 epoch 开始时被随机打乱。

下面创建模型实例：

```
model=init_model()
model.to(device)
```

如下是 init_model 的具体实现。通过 init_from 变量判断模型的 3 种模式：scratch 从头训练、resume 增量预训练。

```
def init_model():
    # 模型参数
    model_args = dict(
        dim=dim,          # 维度
        n_layers=n_layers, # 层数
        n_heads=n_heads,   # 头数
        n_kv_heads=n_heads,# 头数
        vocab_size=64793,  # 词汇表大小
        multiple_of=multiple_of,
        max_seq_len=max_seq_len, # 序列的最大长度
        dropout=dropout,
    )  # start with model_args from command line
    if init_from == "scratch":
        # 从头训练
```

```
        # init a new model from scratch
        print("Initializing a new model from scratch")
        gptconf = ModelArgs(**model_args)
        model = Transformer(gptconf)
    elif init_from == "resume":
        # 增量预训练
        print(f"Resuming training from {out_dir}")
        # resume training from a checkpoint.
        ckpt_path = os.path.join(out_dir, "ckpt.pt")
        # 加载检查点文件
        checkpoint = torch.load(ckpt_path, map_location=device)
        checkpoint_model_args = checkpoint["model_args"]
        # force these config attributes to be equal otherwise we can't even resume training
        # the rest of the attributes (e.g. dropout) can stay as desired from command line
        for k in ["dim", "n_layers", "n_heads", "n_kv_heads", "vocab_size", "multiple_of",
"max_seq_len"]:
            model_args[k] = checkpoint_model_args[k]
        # create the model
        gptconf = ModelArgs(**model_args)
        model = Transformer(gptconf)
        state_dict = checkpoint["model"]
        # fix the keys of the state dictionary :(
        # honestly no idea how checkpoints sometimes get this prefix, have to debug more
        unwanted_prefix = "_orig_mod."
        for k, v in list(state_dict.items()):
            if k.startswith(unwanted_prefix):
                state_dict[k[len(unwanted_prefix) :]] = state_dict.pop(k)
        model.load_state_dict(state_dict)
        iter_num = checkpoint["iter_num"]
        best_val_loss = checkpoint["best_val_loss"]
    return model
```

接下来创建优化器并开始训练：

```
# initialize a GradScaler. If enabled=False scaler is a no-op
scaler = torch.cuda.amp.GradScaler(enabled=(dtype == 'float16'))
# optimizer
optimizer = model.configure_optimizers(weight_decay, learning_rate, (beta1, beta2), device_type)

# ...

raw_model = model.module if ddp else model # unwrap DDP container if needed
# training loop

# 计算了每个训练周期（epoch）中的迭代次数（iteration）
# 返回的是训练数据加载器 `train_loader` 中的批次数量。
```

```
# 在 PyTorch 中，`DataLoader` 对象是一个可迭代的对象，它将数据集分成多个批次，
# 每个批次包含多个数据样本。
# 当我们对 `DataLoader` 对象使用 `len()` 函数时，返回的是批次的数量
# 也就是每个训练周期需要进行的迭代次数。
iter_per_epoch=len(train_loader)
for epoch in range(max_epoch):
    train_epoch(epoch)

# 保存 epoch 训练结果
torch.save(raw_model.state_dict(),'{}/epoch_{}.pth'.format(save_dir,epoch))
    else:
        torch.save(raw_model.state_dict(),'{}/epoch_{}.pth'.format(save_dir,epoch))
```

看一个周期内是如何训练的 train_epoch ：

```
def train_epoch(epoch):
    start_time=time.time()
    # 取出一个批次，根据 X 预测 Y
    for step, (X, Y) in enumerate(train_loader):
        # 数据放到 GPU
        X=X.to(device)
        Y=Y.to(device)
        lr = get_lr(epoch*iter_per_epoch+step) if decay_lr else learning_rate
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
        # and using the GradScaler if data type is float16
        # for micro_step in range(gradient_accumulation_steps):
        # ……
        with ctx:
            # 生成预测
            logits = model(X, Y)
            loss = raw_model.last_loss
            loss = loss / gradient_accumulation_steps
        # immediately async prefetch next batch while model is doing the forward pass on the GPU
        # backward pass, with gradient scaling if training in fp16
        scaler.scale(loss).backward()
        #
        if (step + 1) % gradient_accumulation_steps == 0:
            # clip the gradient
            if grad_clip != 0.0:
                scaler.unscale_(optimizer)
                torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)
            # step the optimizer and scaler if training in fp16
            scaler.step(optimizer)
            scaler.update()
            # flush the gradients as soon as we can, no need for this memory anymore
```

```
            optimizer.zero_grad(set_to_none=True)
        # 打印日志
        if step % log_interval == 0:
            # 打印训练过程日志
        #
        if step % save_interval == 0:
            # 保存中间过程
            # 将模型切换到评估模式（evaluation mode）。
            # 在评估模式下，某些类型的层（如 Dropout 和 BatchNorm）会改变它们的行为。
            # 例如，Dropout 层在训练模式下会随机丢弃一部分神经元
            # 而在评估模式下则不会丢弃任何神经元。
            model.eval()
            # 保存模型
            torch.save(
                model.module.state_dict(),
                '{}/iter_{}.pth'.format(
                    save_dir,
                    int(step+epoch*iter_per_epoch)))
            # 这行代码将模型切换回训练模式。
            model.train()
```

6 Transformer 模型代码阅读

在本节中，阅读项目中使用到的 Transformer 模型代码。其中，我用 karpathy/minGPT 作为对照，来对比两者的差异。

TransformerBlock

一个 Transformer 块，Transformer 是由多层组成的（由层数 n_layers ）控制，而每一层都是由 TransformerBlock 叠加而成。

TransformerBlock 块的实现，首先，输入数据 x 通过 self.attention_norm 进行归一化处理，然后传入 self.attention 层。注意力层的输出与原始输入 x 相加，得到 h 。 h 通过 self.ffn_norm 进行归一化处理，然后传入 self.feed_forward 层。前馈神经网络层的输出与 h 相加，得到最终的输出 out 。具体代码如下：

```
class TransformerBlock(nn.Module):
    def __init__(self, layer_id: int, args: ModelArgs):
        super().__init__()
        self.n_heads = args.n_heads
        self.dim = args.dim
        self.head_dim = args.dim // args.n_heads
        self.attention = Attention(args)
        self.feed_forward = FeedForward(
            dim=args.dim,
            hidden_dim=4 * args.dim,
```

```
            multiple_of=args.multiple_of,
            dropout=args.dropout,
        )
        self.layer_id = layer_id
        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)

    def forward(self, x, freqs_cos, freqs_sin):
        h = x + self.attention.forward(self.attention_norm(x), freqs_cos, freqs_sin)
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

对比与 minGPT 的代码，Block 实现基本是一致的。FeedForward 也基本一致，但是存在少许差异。

需要注意，FeedForward 层的隐藏状态维度是 dim 的 4 倍，

FeedForward 是一个前馈神经网络它包含三个线性层和一个 dropout 层。初始化过程：

```
class FeedForward(nn.Module):
    def __init__(self, dim: int, hidden_dim: int, multiple_of: int, dropout: float):
        super().__init__()
        # 将 `hidden_dim` 的值变为原来的 2/3。这是为了在后续的线性层中减少参数的数量。
        hidden_dim = int(2 * hidden_dim / 3)
        # multiple_of 可以控制让隐藏状态变大
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
        # 这是第一个线性层，它将输入的维度从 dim 变为 hidden_dim。
        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        # 这是第二个线性层，它将输入的维度从 `hidden_dim` 变回 `dim`。
        self.w2 = nn.Linear(hidden_dim, dim, bias=False)
        # 这是第三个线性层，它将输入的维度从 `dim` 变为 `hidden_dim`。
        self.w3 = nn.Linear(dim, hidden_dim, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.dropout(self.w2(F.silu(self.w1(x)) * self.w3(x)))
```

forward 步骤拆解：

. self.w1(x) : 输入数据 x 首先通过一个线性层 self.w1 。这个线性层会对 x 进行线性变换。
. F.silu(...) : 然后，线性层的输出通过 Sigmoid Linear Unit (SiLU) 激活函数。SiLU 函数也被称为 Swish 函数，它的公式是 silu(x) = x * sigmoid(x) ，其中 sigmoid(x) 是 logistic sigmoid 函数。SiLU 函数可以增加模型的非线性，使得模型能够学习更复杂的模式。
. ... * self.w3(x) : 接着，SiLU 函数的输出与 self.w3(x) 的结果相乘。这是一种称为门控（gating）的技术，它可以控制信息的流动。

. self.w2(...) : 然后，乘法的结果通过另一个线性层 self.w2 。这个线性层会对数据进行另一次线性变换。
. self.dropout(...) : 最后，线性层的输出通过一个 dropout 层。在训练过程中，dropout 层会随机丢弃一部分数据，这可以防止模型过拟合。

与 minGPT 对比，两者使用的激活函数不同，两者对比：

特性	本文	minGPT
隐藏层维度动态计算	输入维度 * 4	
激活函数	SiLU (Swish)	NewGELU (GELU 变体)

SiLU (Swish) 和 NewGELU (GELU 变体) 的对比:

SiLU 和 GELU 都是非线性激活函数，可以替代传统的 ReLU 和 Sigmoid 函数。
SiLU 的计算公式为 x * sigmoid(x) ，GELU 的计算公式为 x * P(x) ，其中 P(x) 是一个累积分布函数 (CDF)。
SiLU 和 GELU 都具有平滑的梯度，可以缓解梯度消失问题。
SiLU 的计算速度比 GELU 快，但 GELU 的性能通常比 SiLU 好。

总结:

两个实现的主要区别在于隐藏层维度和激活函数的选择。
第一个实现使用动态计算的隐藏层维度和 SiLU 激活函数，具有较快的计算速度。
第二个实现使用固定比例的隐藏层维度和 NewGELU 激活函数，通常具有更好的性能。

建议:

在实际应用中，可以根据具体需求选择合适的激活函数。
如果需要较快的计算速度，可以选择 SiLU 激活函数。
如果需要更好的性能，可以选择 NewGELU 激活函数。

参考资料:

SiLU论文: https://arxiv.org/abs/1702.07867
GELU论文: https://arxiv.org/abs/1606.08415
NewGELU实现: https://github.com/huggingface/transformers/blob/main/src/transformers/activations.py

Attention

在 forward 方法中，定义了模型的前向传播过程。首先，输入数据 x 被分解为查询、键和值。然后，这些查询和键被应用了旋转位置嵌入（Rotary Position Embedding，RoPE）。接着，键和值被复制以匹配查询的数量。然后，查询、键和值被重塑以便进行注意力计算。

具体实现如下：

```
def forward(
    self,
    x: torch.Tensor,
    freqs_cos: torch.Tensor,
    freqs_sin: torch.Tensor,
):
    bsz, seqlen, _ = x.shape

    # QKV
    # 首先，输入数据 `x` 被分解为查询、键和值。
    xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)
    xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
    xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
    xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)

    # RoPE relative positional embeddings
    # 然后，这些查询和键被应用了旋转位置嵌入（Rotary Position Embedding，RoPE）。
    xq, xk = apply_rotary_emb(xq, xk, freqs_cos, freqs_sin)

    # grouped multiquery attention: expand out keys and values
    # ...

    # make heads into a batch dimension
    xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
    xk = xk.transpose(1, 2)
    xv = xv.transpose(1, 2)

    # flash implementation
    if self.flash:
        output = torch.nn.functional.scaled_dot_product_attention(xq, xk, xv, attn_mask=None,
dropout_p=self.dropout if self.training else 0.0, is_causal=True)
    else:
        # manual implementation
        scores = torch.matmul(xq, xk.transpose(2, 3)) / math.sqrt(self.head_dim)
        assert hasattr(self, 'mask')
        scores = scores + self.mask[:, :, :seqlen, :seqlen]   # (bs, n_local_heads, seqlen, cache_len +
seqlen)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        scores = self.attn_dropout(scores)
        output = torch.matmul(scores, xv)  # (bs, n_local_heads, seqlen, head_dim)

    # restore time as batch dimension and concat heads
    output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)

    # final projection into the residual stream
```

```
    output = self.wo(output)
    output = self.resid_dropout(output)
    return output
```

其中，在预训练时， n_kv_heads 我设置为与 n_heads 相等。

当 self.flash 为真时，代码使用PyTorch的 torch.nn.functional.scaled_dot_product_attention 函数来执行缩放点积注意力计算。这是一种更快、更简洁的实现方式，通常用于优化性能和计算效率。

当 self.flash 为假时，则采用手动实现的注意力机制，这涉及到更详细的步骤，包括计算查询（xq）和键（xk）之间的得分，应用掩码（mask），执行softmax操作，然后计算最终的输出。这种实现方式提供了更多的自定义性和控制性，但可能不如 torch.nn.functional.scaled_dot_product_attention 函数高效。

flash 的判断是根据 PyTorch 中是否有 'scaled_dot_product_attention' 函数来判断的：

```
# use flash attention or a manual implementation?
self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
if not self.flash:
    print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
    mask = torch.full((1, 1, args.max_seq_len, args.max_seq_len), float("-inf"))
    mask = torch.triu(mask, diagonal=1)
    self.register_buffer("mask", mask)
```

Transformer

configure_optimizers 中有一个 use_fused 属性：

use_fused 是一个布尔值，用于决定是否使用融合版本的 AdamW 优化器。融合版本的优化器在某些操作上进行了优化，以提高计算效率和性能。

如何估算大模型的参数

参数介绍：

词表大小为 vocab_size
序列长度： max_seq_len
模型维度： dim
模型层数： n_layers
注意力头数： n_heads

LLM 模型的参数主要来自于以下几个部分：

Embedding 层：
该层将每个词映射为一个固定长度的向量。

通常，模型的嵌入层会有一个与模型维度相等的参数量，用于将输入词标（tokens）映射到高维空间。
因此，嵌入层的参数数量大约为 vocab_size * dim 。
Transformer 层：
Transformer 层是 LLM 模型的核心部分，它由多个 Self-Attention 模块组成。
自注意力层（self-attention）:
包括三个权重矩阵（Query, Key, Value）各自有 dim * dim 个参数
以及一个输出投影层也有 dim * dim 个参数。
由于有 n_heads 个注意力头，但参数通常在所有头之间共享，所以基本上是 n_heads * dim * dim ，但实际上因为维度通常被分割成多头，所以总量可能更接近 3 * dim * dim + dim * dim 。
前馈网络（FFN）: 通常有两层，每层有 dim * dim 参数（假设隐藏层大小等于模型维度），因此是 2 * dim * dim 。
输出层：输出层将 Transformer 层的输出映射为最终的预测结果。假设输出类别数为 num_classes ，则输出层的参数量为 dim * num_classes 。

参数量 = vocab_size * dim + n_layers * dim * dim * n_heads + dim * num_classes

以 Llama2-Chinese-218M-v1 为例：max_seq_len=1024，dim=1024，n_layers=12，n_heads=8。

参数量 = 64793 * 1024 + 12 * 1024 * 1024 * 8 + 1024 * 100

数据集下载

该模型采用了一些开放数据集，非常有价值，我也一并下载之。

Hugging Face 数据集下载

参见此文：下载Hugging Face的数据集并离线使用的方法 - 知乎 ：

```
# 下载
import datasets
dataset = datasets.load_dataset("dataset_name")
dataset.save_to_disk('your_path')

# 加载
import datasets
dataset = load_from_disk("your_path")
```

在《Downloading datasets 》提到，数据集本身就是 Git LFS Repo，直接 Clone 就行，这样更加方便。

我将收集到的数据集记录到。

网络资源

DLLXW/baby-llama2-chinese: 用于从头预训练+SFT一个小参数量的中文LLaMa2的仓库；24G单卡即可运行得到一个具备简单中文问答能力的chat-llama2.