CS336-assignment1

一、BPE Tokenizer Training

要求：编写一个字节级的 BPE Tokenizer 函数，能够接收 special_tokens 列表以及词表大小，返回词表和 merges 的过程。

函数的主要流程如下：

阶段 1：读取数据文件并进行分块

- 定义分块的数量等于 4 倍预分词的进程数量；
- 在分块的时候注意 special_tokens 的处理，不能切分 special_tokens；
- 分块后得到的是各个 chunk 的起始和结束下标列表。

阶段 2：预分词处理

- 使用 joblib.Parallel 进行多进程预分词处理；
- 在预分词的过程中，如果遇到 special_tokens 则保留，否则使用正则化表达式预分词。

阶段3：构建频率统计与初始化

此阶段的目标是完成所有合并前的准备工作，包括计算初始频率和构建加速后续迭代所需的数据结构。

1. **统计Token频率**：构建一个 `token_freqs` 字典，统计在阶段2的预分词结果中，除了 special_tokens 以外，其余每个 token 的出现总次数。
2. **初始化词表 (Vocab)**：构建初始词表 `vocab` 。首先将 `special_tokens` 逐个添加进词表，然后将0到255的全部单个字节（ `bytes([i])` ）添加进词表。
3. **初始化Token拆分**：构建 `split_freqs` 字典，它的键是预分词后的 token，值是该 token 按字节拆分后的列表。例如 `b'hello'` -> `[b'h', b'e', b'l', b'l', b'o']` 。
4. **构建初始“对”频率和倒排索引**：
    - 创建一个 `pair_freqs` 字典，用于存储所有相邻字节对（pair）的全局频率。
    - 创建一个倒排索引 `pair_to_words` 字典。它的键是字节对，值是一个集合（Set），包含所有出现了这个字节对的 token。
    - 遍历 `split_freqs` 中的每一个 token 及其拆分列表。根据该 token 在 `token_freqs` 中的频率，累加计算其所有内部相邻字节对的频率，并存入 `pair_freqs` 。同时，将该 token 添加到其包含的每一个字节对在 `pair_to_words` 索引中的集合里。

阶段4：BPE Merge 迭代

循环共 `vocab_size - 初始词表大小` 次，在每次循环里：

1. **寻找最佳合并对**：
   - 从 `pair_freqs` 字典中找出频率最高的字节对 `best_pair`。
   - **关键**：如果存在多个频率相同的字节对，则选择字节序最大的那一个作为 `best_pair`。
   - 将 `best_pair` 存入 `merges` 列表。
2. **更新词表**：
   - 将 `best_pair` 的两个字节合并，形成一个新的 `new_token`。
   - 将这个 `new_token` 添加到 `vocab` 词表的末尾。
3. **高效更新频率与拆分**：
   - **定位受影响的Token**：利用阶段3建立的倒排索引 `pair_to_words`，直接获取所有包含 `best_pair` 的 token 列表（`words_to_update = pair_to_words[best_pair]`）
   - **遍历并更新受影响的Token**：遍历上面获取的 `words_to_update` 列表中的每一个 `token`：
     - **A. 移除旧统计**：获取该 `token` 的旧拆分（`old_split`）。遍历 `old_split`，将其中的所有字节对的频率从 `pair_freqs` 中减去（减去的值为该 `token` 的频率 `token_freqs[token]`），并从 `pair_to_words` 索引中移除该 `token`。
     - **B. 生成新拆分**：在 `old_split` 中，将所有出现的 `best_pair` 替换为 `new_token`，从而生成新拆分（`new_split`）。
     - **C. 更新Token拆分**：在 `split_freqs` 中，将该 `token` 对应的拆分列表更新为 `new_split`。
     - **D. 添加新统计**：遍历 `new_split`，将其中的所有新形成的字节对的频率加回到 `pair_freqs` 中，并更新 `pair_to_words` 索引。

Problem1：在 TinyStories dataset 以 10000 为 vocabulary size 训练一个字节级的 BPE。要求不使用 GPUs，RAM 小于等于 30GB，（1）运行时间小于等于 30 分钟；（2）pretokenization 时间小于 2 分钟。

我在 M4 芯片，RAM 为 32GB 的 Macbook Pro 上首次运行时间为28分钟，符合（1），但 pretokenization 时间为 6.7 分钟，远大于规定时间。优化过程如下：

1. 将 multiprocessing.Pool.map() 替换成 joblib，避免复制每个 chunk 到子进程，减少串行数据传输。修改后运行 pretokenization 的时间减少至 6.3 分钟，提升不大；
2. 调节 chunk 粒度，即增加分块数，提高缓存命中，避免 CPU 空转。初始的 chunk 数量等于进程的数量，现在将 chunk 数量改为 4 倍进程数量。修改后 pretokenization 的时间减少至 1.38 分钟，小于规定的 2 分钟。

总用时缩短到了 19 分钟。

Problem2：在 TinyStories tokenizer 训练过程中，哪部份占最长的时间？

答：构建 token 频率表和初始化词表这部份占了最长的时间，用时大概为 12.619 分钟。

Problem3：使用 TinyStories 构建的 vocab 中，最长的 token 是什么？它有意义吗？

答：最长的 token 是 b' accomplishment'，accomplishment 是一个完整的常见词，在儿童故事数据集中可能用于描述角色成功完成某事的情节，所以它有意义。

Problem4：在 OpenWebText dataset 以 32000 为 vocabulary size 训练一个字节级的 BPE。要求不使用 GPUs，RAM 小于等于 100GB，运行时间小于等于 12 小时。在构建的 vocab 中，最长的 token 是什么？它有意义吗？

答：最长的 token 是
b'\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82\xc3\x83\xc3\x82'。
b'\xc3\x83\xc3\x82' == “ÃÂ”，这个 token 是 44 字节连续重复 “ÃÂÃÂÃÂ...”，它没有意义。

二、Implementing the tokenizer

要求：实现一个 Tokenizer 类，实现根据词表和 merges 列表进行字符串的 encode 和 decode，同时支持用户提供 special_tokens，如果不在词表的话，将它们加入词表。这个类要包含 __init__，from_files，encode，encode_iterable，decode 方法。

这些方法里面，encode 是难点。实现思路如下：

- 为了使字符串按 merges 列表的顺序合并，需要额外定义一个 merge 到 merges 列表位置映射的字典 merge_rank。
- 构建一个词表位置到该位置对应 token 映射的字典 vocab_to_id。
- 按 special_tokens 对字符串进行分块，同时保留 special_tokens。
- 遍历所有分块，如果是 special_token，则直接编码后加入 encode_list，否则先对分块进行预分词处理。然后遍历所有预分词后的 tokens，对它们进行合并。合并的过程如下：
  - 构建包含各个 token 里所含有的单个字节列表 split；
  - 执行循环，遍历 split，构建两两相邻的字节对 pair_rank 字典，并根据字节对从 merge_rank 取出这个字节对在 merges 列表的位置，如果找不到，那么就终止循环，否则从 pair_rank 里选出在 merge_rank 排序最小并且字节值组合整数越大的对 best_pair。如果 best_pair 不在 merge_rank，则终止循环，否则创建一个新的 new_split 列表，然后遍历 split 列表，如果存在相邻字节对与 best_pair 相等的话，那就将它们合并后放入 new_split 列表，否则放入单个字节到 new_split 列表，遍历结束后，用 new_split 替换 split。
  - 遍历上一步得到的列表，使用 vocab_to_id 将它们转换成 tokenid 后放入最后的 encode 列表。

Problem1：使用 10K 大小的词表编码 TinyStories 数据集，tokenizer 的压缩比是多少？

压缩比 (bytes/token)：4.1194。

Problem2：根据我实现的 Tokenizer，预估 tokenize Pile dataset (825GB of text) 需要多长时间?

我的 Tokenizer Encoded 2.07 GB 数据花了 2498.18s，速度 0.85 MB/s。所以 tokenize Pile dataset 需要花 825 * 1024 / 0.85 = 993882.35 秒，约等于 11.5 天。

Problem3：使用 TinyStories 的 tokenizer，编码它的训练集数据，为什么使用 uint16 作为 tokenID 的数据类型是一个合适的选择?

- uint16 每个 token 占 2 个字节，范围是 0 ～ 65535，而 TinyStories 的词表大小是 10K，uint16 能完全覆盖。

三、Implementing the linear module

要求：实现一个继承自 torch.nn.Module 的 Linear 类，并执行线性变换。推荐使用下面的接口：

代码块

```
def __init__(self, in_features, out_features, device=None, dtype=None)
def forward(self, x: torch.Tensor) -> torch.Tensor
```

确保：

- 是 nn.Module 的子类；
- 调用超类构造函数；
- 由于内存顺序原因，使用 W 而不是 W^T 进行构造和存储，并将其放入 nn.Parameter 中；
- 权重矩阵 W 使用 $N(\mu = 0, \sigma^2 = \frac{2}{d_{in} + d_{out}})$，截断于 $[-3\sigma, 3\sigma]$ 进行初始化。

Linear 类的代码如下，注意 nn.Linear 的 weight 是按 (out_features, in_features) 的顺序存储，而这里的 weight 要按 (in_features, out_features) 的顺序存储。nn.Linear 的 forward 函数是 x @ W^T，而这里是 x @ W。

代码块

```
class Linear(nn.Module):
    def __init__(self, in_features, out_features, device=None, dtype=None):
        super().__init__()
        factory_kwargs = {'device': device, 'dtype': dtype}
        std = math.sqrt(2.0 / (in_features + out_features))
        weight = torch.empty(in_features, out_features, **factory_kwargs)
        torch.nn.init.trunc_normal_(weight, mean=0.0, std=std, a=-3*std,
b=3*std)
        self.weight = nn.Parameter(weight)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
```

```
        out = x @ self.weight # [*, in_features] @ [in_features, out_features]
        return out
```

四、Implementing the embedding module

要求：实现一个继承自 torch.nn.Module 的 Embedding 类，并执行嵌入查找。推荐使用下面的接口：

代码块

```
def __init__(self, num_embeddings, embedding_dim, device=None, dtype=None)
def forward(self, token_ids: torch.Tensor) -> torch.Tensor
```

确保：

- 是 nn.Module 的子类；
- 调用超类构造函数；
- 使用 $N(\mu = 0, \sigma^2 = 1)$，截断于 $[-3, 3]$ 初始化 embedding 矩阵，并将其放入 nn.Parameter 中；
- 存储 embedding 矩阵时使用 d_model 作为最后一个维度。

Embedding 类的代码如下，只需将 token_ids 的 tensor 传给 Embedding 的权重矩阵即可返回这些 token_ids 的嵌入表。

代码块

```
class Embedding(nn.Module):
    def __init__(self, num_embedding, embedding_dim, device=None, dtype=None):
        super().__init__()
        factory_kwargs = {'device': device, 'dtype': dtype}
        weight = torch.empty(num_embedding, embedding_dim, **factory_kwargs)
        torch.nn.init.trunc_normal_(weight, mean=0.0, std=1.0, a=-3, b=3)
        self.weight = nn.Parameter(weight)

    def forward(self, token_ids: torch.Tensor) -> torch.Tensor:
        out = self.weight[token_ids]
        return out
```

五、Root Mean Square Layer Normalization

将 RMSNorm 实现为 torch.nn.Module，推荐使用下面的接口：

```
def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None)
def forward(self, x: torch.Tensor) -> torch.Tensor
```

注意：

- 在执行 normalization 之前先将输入向上转换为 torch.float32，然后再向下转换为原始的数据类型；
- 权重矩阵使用 1 初始化。

代码块

```
class RMSNorm(nn.Module):
    def __init__(self, d_model: int, eps: float = 1e-5, device=None, dtype=None):
        super().__init__()
        factory_kwargs = {'device': device, 'dtype': dtype}
        self.eps = eps
        self.d_model = d_model
        self.weight = nn.Parameter(torch.ones(d_model, **factory_kwargs))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        in_dtype = x.dtype
        x = x.to(torch.float32)
        RMS_x = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)
        result = x / RMS_x * self.weight
        return result.to(in_dtype)
```

六、Implement the position-wise feed-forward network

要求：实现 SwiGLU 前馈神经网络，由 SiLU 激活函数和一个 GLU 组成。

$$SiLU(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

$$GLU(x, W_1, W_2) = \sigma(W_1 x) \odot W_2 x$$

$$FFN(x) = SwiGLU(x, W_1, W_2, W_3) = W_2(SiLU(W_1 x) \odot W_3 x)$$

注意：

- 在这个特殊情况下，可以在你的实现中使用 torch.sigmoid 以确保数值稳定性；
- 在你的实现中，你应该设置 d_ff 约等于 8/3 * d_model，确保内部前馈层的维度是 64 的倍数以充分利用硬件。

我的思路是如果输入的 d_ff 不等于 8/3 * d_model 并且能整除 64，那么我就用 0 扩充 d_ff，代码如下：

```python
def adjust_weight(weight: torch.Tensor, target_shape: tuple) -> torch.Tensor:
    current_shape = weight.shape
    # Use the same device as weight, but default to CPU if .device is not accessible
    device = weight.device if weight.device.type != 'meta' else torch.device('cpu')
    new_weight = torch.zeros(target_shape, dtype=weight.dtype, device=device)
    min_dim0 = min(current_shape[0], target_shape[0])
    min_dim1 = min(current_shape[1], target_shape[1])
    new_weight[:min_dim0, :min_dim1] = weight[:min_dim0, :min_dim1]
    return new_weight

class SwiGLU(nn.Module):
    def __init__(self, d_model, d_ff, device=None, dtype=None):
        super().__init__()
        self.d_model = d_model
        self.d_ff = self.adjust_dff(d_model) if d_ff < self.adjust_dff(d_model) else d_ff
        self.w1 = Linear(d_model, self.d_ff)
        self.w2 = Linear(self.d_ff, d_model)
        self.w3 = Linear(d_model, self.d_ff)

    def adjust_dff(self, d_model: int) -> int:
        return (int((8/3) * d_model) + 63) // 64 * 64

    def load_weights(self, w1: torch.Tensor, w2: torch.Tensor, w3: torch.Tensor):
        w1_adj = adjust_weight(w1, (self.d_ff, self.d_model))
        w2_adj = adjust_weight(w2, (self.d_model, self.d_ff))
        w3_adj = adjust_weight(w3, (self.d_ff, self.d_model))
        with torch.no_grad():
            self.w1.weight.copy_(w1_adj.T)
            self.w2.weight.copy_(w2_adj.T)
            self.w3.weight.copy_(w3_adj.T)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        W1_x = self.w1(x)
        W3_x = self.w3(x)
        SiLU_x = run_silu(W1_x)
        out = self.w2((SiLU_x * W3_x))
        return out

def run_silu(in_features: Float[Tensor, " ..."]) -> Float[Tensor, " ..."]:
    return in_features * torch.sigmoid(in_features)
```

七、Implement RoPE

要求：实现一个 RotaryPositionalEmbedding 类，对输入的 tensor 进行 RoPE。推荐使用下面的接口：

代码块

```
def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None)
def forward(self, x: torch.Tensor, token_positions: torch.Tensor) ->
torch.Tensor
```

$$\theta_{i,k} = \frac{i}{theta^{2k/d}}, k \in [0, 1, ..., d/2 - 1]$$

$$R_k^i = \begin{bmatrix} cos(\theta_{i,k}) & -sin(\theta_{i,k}) \\ sin(\theta_{i,k}) & cos(\theta_{i,k}) \end{bmatrix}$$

代码块

```
class RotaryPositionalEmbedding(nn.Module):
    def __init__(self, theta: float, d_k: int, max_seq_len: int, device=None):
        super().__init__()
        theta_ik = theta ** (-torch.arange(0, d_k, 2, device=device) / d_k) #
[d_k / 2]
        pos = torch.arange(max_seq_len, device=device) # [max_seq_len]
        angles = torch.einsum("i,j->ij", pos, theta_ik) # [max_seq_len, d_k /
2]
        self.register_buffer("cos", torch.cos(angles), persistent=False)
        self.register_buffer("sin", torch.sin(angles), persistent=False)

    def forward(self, x: torch.Tensor, token_positions: torch.Tensor) ->
torch.Tensor:
        cos = self.cos[token_positions]
        sin = self.sin[token_positions]
        x1 = x[..., 0::2] # [..., d_k / 2]
        x2 = x[..., 1::2]
        rotated_x1 = x1 * cos - x2 * sin
        rotated_x2 = x1 * sin + x2 * cos
        # out = torch.stack([rotated_x1, rotated_x2], dim=-1).reshape(x.shape)
# [..., d_k / 2, 2] -> [..., d_k]
        out = torch.stack([rotated_x1, rotated_x2], dim=-1).flatten(-2)
        return out
```

八、Implement softmax

要求：写一个函数对一个 tensor 执行 softmax 运算。使用从第 i 维的所有元素中减去第 i 维中最大值的技巧来避免数值稳定性问题。

代码块

```
def run_softmax(in_features: Float[Tensor, " ..."], dim: int) -> Float[Tensor, " ..."]:
    dim_max = torch.amax(in_features, dim=dim, keepdim=True)
    dim_exp = torch.exp(in_features - dim_max)
    sum_dim_exp = torch.sum(dim_exp, dim=dim, keepdim=True)
    return dim_exp / sum_dim_exp
```

九、Implement scaled dot-product attention

要求：实现缩放点积注意力函数。你的实现应该处理形状为 (batch_size, ..., seq_len, d_k) 的键和查询，以及形状为 (batch_size, ..., seq_len, d_v) 的值，其中 ... 表示任意数量的其他类似批处理的维度（如果提供）。该实现应该返回形状为 (batch_size, ..., d_v) 的输出。

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

代码块

```
def run_scaled_dot_product_attention(
    Q: Float[Tensor, " ... queries d_k"],
    K: Float[Tensor, " ... keys d_k"],
    V: Float[Tensor, " ... values d_v"],
    mask: Float[Tensor, " ... queries keys"] | None = None,
) -> Float[Tensor, " ... queries d_v"]:
    score = torch.einsum("... q d, ... k d -> ... q k", Q, K) / math.sqrt(K.size(-1))
    # score = (Q @ K.transpose(-2, -1)) * (1.0 / math.sqrt(K.size(-1)))
    if mask is not None:
        score = score.masked_fill(mask == False, float('-inf'))
    score = run_softmax(score, dim=-1)
    att = score @ V
    return att
```

十、Implement causal multi-head self-attention

要求：将因果多头自注意力机制实现为 torch.nn.Module。你的实现至少应接受以下参数：d_model，num_heads。

$$MultiHead(Q,K,V) = Concat(head_1,...,head_h)for head_i = Attention(Q_i,K_i,V_i)$$

$$MultiHeadSelfAttention(x) = W_O MultiHead(W_Q x, W_K x, W_V x)$$

注意一点，我在 Linear 类定义存储权重矩阵 W 是按输入维度的正常顺序存储的，而不是按 nn.Linear 那样存储 W 的转置。同时 forward 是 x @ W，即维度变化是 [*, in_features] @ [in_features, out_features]。

所以为了将一个输入特征向量分别映射到多个头，Multihead_self_attention 类的 Linear 层定义为：

代码块

```
self.q_proj = Linear(self.d_model, self.num_heads * self.d_k)
self.k_proj = Linear(self.d_model, self.num_heads * self.d_k)
self.v_proj = Linear(self.d_model, self.num_heads * self.d_v)
```

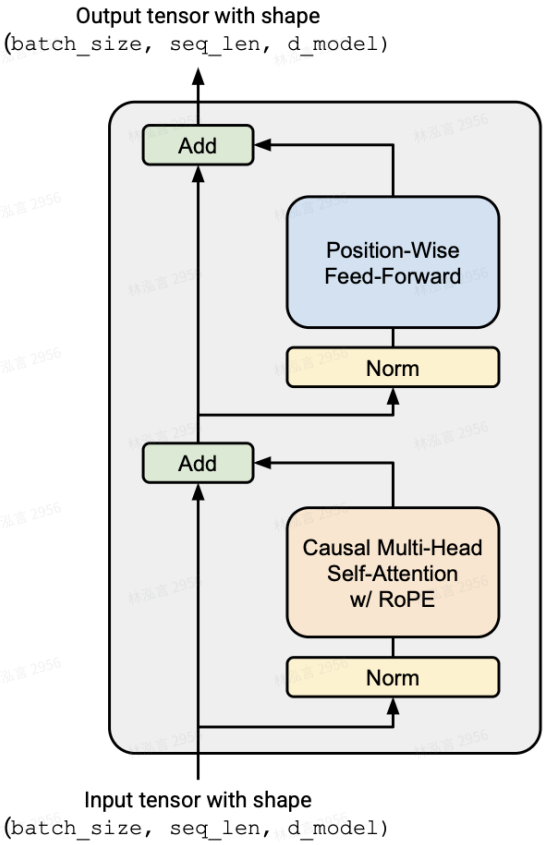
由于输入的维度是 Float[Tensor, " d_k d_in"]，所以要将它们转置后再赋给 Linear 的权重。

代码块

```
def run_multihead_self_attention(
    d_model: int,
    num_heads: int,
    q_proj_weight: Float[Tensor, " d_k d_in"],
    k_proj_weight: Float[Tensor, " d_k d_in"],
    v_proj_weight: Float[Tensor, " d_v d_in"],
    o_proj_weight: Float[Tensor, " d_model d_v"],
    in_features: Float[Tensor, " ... sequence_length d_in"],
) -> Float[Tensor, " ... sequence_length d_out"]:
with torch.no_grad():
    multihead_att.q_proj.weight.copy_(q_proj_weight.T)
    multihead_att.k_proj.weight.copy_(k_proj_weight.T)
    multihead_att.v_proj.weight.copy_(v_proj_weight.T)
    multihead_att.o_proj.weight.copy_(o_proj_weight.T)
```

十一、Implement the Transformer block

要求：实现按下图所示结构的 pre-norm 的 Transformer 块，你的实现至少应接受以下参数：d_model，num_heads，d_ff。



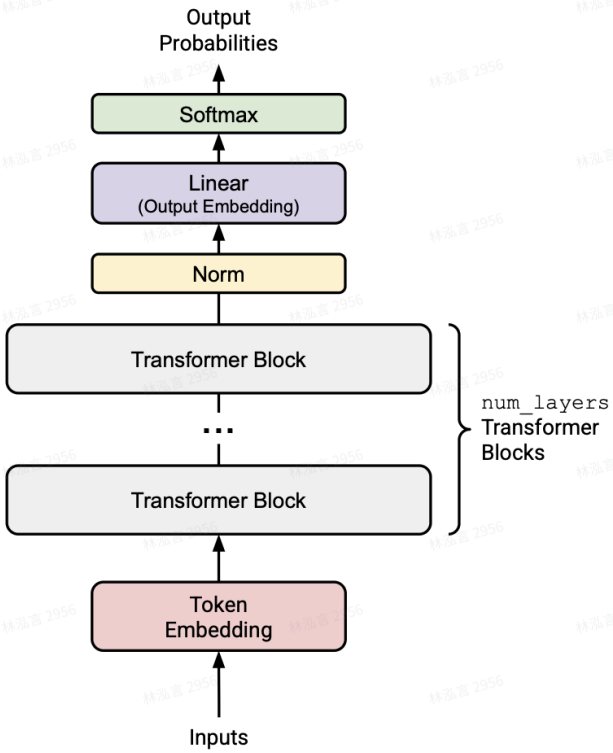
代码块

```
class Transformer_block(nn.Module):
    def __init__(self, d_model: int, num_heads: int, d_ff: int, max_seq_len: int, theta: float | None = None):
        super().__init__()
        if theta is not None:
            pos_encode = RotaryPositionalEmbedding(theta, d_model // num_heads, max_seq_len)
            self.attn = Multihead_self_attention(d_model=d_model, num_heads=num_heads, pos_encode=pos_encode, theta=theta)
        else:
            self.attn = Multihead_self_attention(d_model=d_model, num_heads=num_heads)
        self.rmsn_1 = RMSNorm(d_model=d_model, eps=1e-5)
        self.rmsn_2 = RMSNorm(d_model=d_model, eps=1e-5)
        self.pw_ffn = SwiGLU(d_model=d_model, d_ff=d_ff)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        attn = self.attn(self.rmsn_1(x))
        out1 = x + attn
        out2 = self.pw_ffn(self.rmsn_2(out1))
        out = out1 + out2
        return out
```

十二、Implement the Transformer LM

要求：实现按下图所示结构的Transformer 语言模型。你的实现至少应接受上述所有 Transformer 块的构造参数，以及下面的附加参数：vocab_size，context_length，num_layer。



代码块

```
class Transformer_lm(nn.Module):
    def __init__(self, vocab_size:int, context_length:int, num_layers: int, d_model: int, num_heads: int, d_ff: int, rope_theta: float | None = None):
        super().__init__()
        self.transformer = nn.ModuleDict(dict(
            token_emb = Embedding(num_embedding=vocab_size, embedding_dim=d_model),
            n_block = nn.ModuleList([Transformer_block(d_model=d_model, num_heads=num_heads, d_ff=d_ff, max_seq_len=context_length, theta=rope_theta) for _ in range(num_layers)]),
            rmsn_l = RMSNorm(d_model=d_model, eps=1e-5)
        ))
        self.linear_emb = Linear(d_model, vocab_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        tkemb = self.transformer.token_emb(x)
        for block in self.transformer.n_block:
            tkemb = block(tkemb)
        tkemb = self.transformer.rmsn_l(tkemb)
        out = self.linear_emb(tkemb)
        return out
```

十三、Implement Cross entropy

要求：写一个函数计算交叉熵损失。

$$l(\theta;D)=\frac{1}{|D|m}\sum_{x\in D}\sum_{i=1}^{m}-logp_\theta(x_{i+1}|x_{1:i})$$

$$p(x_{i+1}|x_{1:i})=softmax(o_i)[x_{i+1}]=\frac{exp(o_i[x_{i+1}])}{\sum_{a=1}^{vocab\_size}exp(o_i[a])}$$

代码块

```
def run_cross_entropy(inputs: Float[Tensor, " batch_size vocab_size"], targets: Int[Tensor, " batch_size"]) -> Float[Tensor, ""]:
    dim_max = torch.amax(inputs, dim=-1, keepdim=True)
    dim_submax = inputs - dim_max
    dim_logsumexp = dim_submax - torch.log(torch.sum(torch.exp(dim_submax), dim=-1, keepdim=True))
    return torch.mean(torch.gather(input=-dim_logsumexp, dim=-1, index=targets.unsqueeze(-1)))
```

十四、Implement AdamW

要求：将 AdamW 优化器作为 torch.optim.Optimizer 的子类实现。

Algorithm 1 AdamW Optimizer

init(θ) (Initialize learnable parameters)
$m \leftarrow 0$ (Initial value of the first moment vector; same shape as θ)
$v \leftarrow 0$ (Initial value of the second moment vector; same shape as θ)
for $t = 1, \ldots, T$ **do**
  Sample batch of data B_t
  $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (Compute the gradient of the loss at the current time step)
  $m \leftarrow \beta_1 m + (1-\beta_1) g$ (Update the first moment estimate)
  $v \leftarrow \beta_2 v + (1-\beta_2) g^2$ (Update the second moment estimate)
  $\alpha_t \leftarrow \alpha \frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ (Compute adjusted α for iteration t)
  $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$ (Update the parameters)
  $\theta \leftarrow \theta - \alpha\lambda\theta$ (Apply weight decay)
end for

代码块

```
class AdamW(torch.optim.Optimizer):
```

```
    def __init__(self, params: Iterable[torch.nn.Parameter], lr: float = 1e-3, betas: tuple[float, float] = [0.9, 0.999], eps: float = 1e-8, weight_decay: float = 0.01):
        if lr < 0:
            raise ValueError(f"Invalid learning rate: {lr}")
        if betas[0] < 0:
            raise ValueError(f"Invalid betas[0]: {betas[0]}")
        if betas[1] < 0:
            raise ValueError(f"Invalid betas[1]: {betas[1]}")
        if eps < 0:
            raise ValueError(f"Invalid eps: {eps}")
        if weight_decay < 0:
            raise ValueError(f"Invalid weight_decay: {weight_decay}")
        defaults = {"lr": lr, "betas": betas, "eps": eps, "weight_decay": weight_decay}
        super().__init__(params, defaults)

    def step(self, closure: Optional[Callable] | None = None):
        loss = None if closure is None else closure()
        for group in self.param_groups:
            alpha = group["lr"]
            beta1, beta2 = group["betas"]
            eps = group["eps"]
            lambde = group["weight_decay"]
            for p in group["params"]:
                if p.grad is None:
                    continue
                state = self.state[p]
                t = state.get("t", 1)
                grad = p.grad.data
                m = state.get("m", torch.zeros_like(grad))
                v = state.get("v", torch.zeros_like(grad))
                m_ = beta1 * m + (1 - beta1) * grad
                v_ = beta2 * v + (1 - beta2) * torch.square(grad)
                alpha_t = alpha * (math.sqrt(1 - beta2**t) / (1 - beta1**t))
                p.data -= alpha_t * (m_ / (torch.sqrt(v_) + eps))
                p.data -= alpha * lambde * p.data
                state["m"] = m_
                state["v"] = v_
                state["t"] = t + 1
        return loss
```

十五、Implement cosine learning rate schedule with warmup

代码块

```
def run_get_lr_cosine_schedule(
    it: int,
    max_learning_rate: float,
    min_learning_rate: float,
    warmup_iters: int,
    cosine_cycle_iters: int,
):
    alpha_t = 0.0
    if it < warmup_iters:
        alpha_t = it / warmup_iters * max_learning_rate
    elif it >= warmup_iters and it <= cosine_cycle_iters:
        alpha_t = min_learning_rate + 0.5 * (1 + math.cos(((it-warmup_iters)/(cosine_cycle_iters-warmup_iters))*math.pi)) * (max_learning_rate - min_learning_rate)
    elif it > cosine_cycle_iters:
        alpha_t = min_learning_rate
    return alpha_t
```

十六、Implement gradient clipping

代码块

```
def run_gradient_clipping(parameters: Iterable[torch.nn.Parameter], max_l2_norm: float) -> None:
    grads = []
    for pt in parameters:
        if pt.grad is not None:
            grads.append(pt.grad)
    grads_l2norm = 0.0
    for gd in grads:
        grads_l2norm += (gd ** 2).sum()
    grads_l2norm = torch.sqrt(grads_l2norm)
    if grads_l2norm >= max_l2_norm:
        ft = max_l2_norm / (grads_l2norm + 1e-6)
        for gd in grads:
            gd *= ft
```

十七、Implement data loading

要求：编写一个函数，该函数接受一个 numpy 数组 x（包含 token ID 的整数数组）、一个 batch_size、一个 context_length 和一个 PyTorch 设备字符串（例如，“cpu”或“cuda:0”），并返回一对张量：采样的输入序列和相应的下一个 token 目标。

```python
def run_get_batch(
    dataset: npt.NDArray, batch_size: int, context_length: int, device: str
) -> tuple[torch.Tensor, torch.Tensor]:
    st = torch.randint(len(dataset) - context_length, (batch_size,))
    input_seq = torch.stack([torch.from_numpy((dataset[i : i + context_length]).astype(np.int64)) for i in st])
    target_seq = torch.stack([torch.from_numpy((dataset[i + 1 : i + 1 + context_length]).astype(np.int64)) for i in st])
    if 'cuda' in device:
        input_seq = input_seq.pin_memory().to(device, non_blocking=True)
        target_seq = target_seq.pin_memory().to(device, non_blocking=True)
    else:
        input_seq = input_seq.to(device)
        target_seq = target_seq.to(device)
    return input_seq, target_seq
```

十八、Implement model checkpointing

代码块

```python
def run_load_checkpoint(
    src: str | os.PathLike | BinaryIO | IO[bytes],
    model: torch.nn.Module,
    optimizer: torch.optim.Optimizer,
) -> int:
    checkpoints = torch.load(src)
    model.load_state_dict(checkpoints["model_state"])
    optimizer.load_state_dict(checkpoints["optimizer_state"])
    return checkpoints["iteration"]

def run_save_checkpoint(
    model: torch.nn.Module,
    optimizer: torch.optim.Optimizer,
    iteration: int,
    out: str | os.PathLike | BinaryIO | IO[bytes],
):
    checkpoints = {
        "model_state": model.state_dict(),
        "optimizer_state": optimizer.state_dict(),
        "iteration": iteration
    }
    torch.save(checkpoints, out)
```

十九、部分 Problem 的回答

1. Unicode 字符 chr(0) 会返回什么?

答：'\x00'。

2. 这个字符用 repr() 和 print 打印是有什么不同?

答：用 repr() 打印的是 ''\\x00''，用 print 打印的是空值，什么也没有。

3. 这个字符在 text 里面会发生什么?

>>> chr(0)

>>> print(chr(0))

>>> "this is a test" + chr(0) + "string"

>>> print("this is a test" + chr(0) + "string")

答：直接输出字符都会以 '\x00' 的形式返回，但是使用 print 打印出来的是空值，什么也没有，在 text 里不占位置。

4. 为什么在训练 tokenizer 时更倾向使用 UTF-8 编码的字节，而不是 UTF-16 或 UTF-32?

答：因为 UTF-8 编码是三者中最紧凑的，对英文、数字、常用标点字符使用 1 个字节编码。例如，对字符串 Hello 进行三种编码方式编码，UTF-8 的总字节数最小。

代码块

```
s = "Hello"
s.encode("utf-8")    # b'Hello'      -> 5 bytes
s.encode("utf-16")   # b'\xff\xfeH\x00e\x00l\x00l\x00o\x00' -> 12 bytes
s.encode("utf-32")   # b'\xff\xfe\x00\x00H\x00\x00\x00...' -> 24 bytes
```

5. 思考下面的函数，它将一个 UTF-8 字节串解码成一个 Unicode 字符串，为什么这个函数不正确?提供一个字节串例子让它产生不正确的结果。

代码块

```
def decode_utf8_bytes_to_str_wrong(bytestring: bytes):
    return "".join([bytes([b]).decode("utf-8") for b in bytestring])
>>> decode_utf8_bytes_to_str_wrong("hello".encode("utf-8"))
'hello'
```

答：这个函数错误的原因在于它将 UTF-8 字节串逐个字节地解码，而不是将它作为一个整体去解码，在遇到一个多字节字符时就会解码出错失败。

代码块

```
s = "你好"
b = s.encode("utf-8")  # b'\xe4\xbd\xa0\xe5\xa5\xbd'
```

```
3  decode_utf8_bytes_to_str_wrong(b)
4  UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe4 in position 0:
   unexpected end of data
```

6. 给出一个两字节长的序列，它无法被解码成任何 Unicode 字符。

答：UTF-8 编码规范要求除了第一个字节以外，后续的字节必须以 10 开头。因此只要提供一个第二字节不以 10 开头的序列就可以，例如 b'\xc3\x29'。

7. 运行下面的随机梯度下降 (SGD) 示例，并使用其他三个学习率：1e1，1e2 和 1e3，仅进行 10 次训练迭代。这些学习率对应的损失会发生什么变化？它会衰减得更快、更慢，还是会发散？

代码块

```
1  from collections.abc import Callable, Iterable
2  from typing import Optional
3  import torch
4  import math
5  class SGD(torch.optim.Optimizer):
6    def __init__(self, params, lr=1e-3):
7      if lr < 0:
8        raise ValueError(f"Invalid learning rate: {lr}")
9      defaults = {"lr": lr}
10     super().__init__(params, defaults)
11   def step(self, closure: Optional[Callable] = None):
12     loss = None if closure is None else closure()
13     for group in self.param_groups:
14       lr = group["lr"]
15       for p in group["params"]:
16         if p.grad is None:
17           continue
18         state = self.state[p] # Get state associated with p.
19         t = state.get("t", 0) # Get iteration number from the state, or
   initial value.
20         grad = p.grad.data # Get the gradient of loss with respect to p.
21         p.data -= lr / math.sqrt(t + 1) * grad # Update weight tensor in-place.
22         state["t"] = t + 1 # Increment iteration number.
23     return loss
```

答：使用 1e1 的学习率损失会衰减更慢，使用 1e2 的学习率损失衰减会更快，使用 1e3 的学习率损失会发散。