ChatGPT

Transformer 模型手写实现学习计划

本计划将帮助你在每日 **1–2 小时**的学习时间里，循序渐进地从零手写实现一个 Transformer 模型，并涵盖完整的**模型构建—训练—推理**流程。我们将采用“搭积木”式的方法：先实现若干基础模块（例如多头注意力、前馈网络、层归一化、残差连接、Embedding、位置编码），再组合成 Encoder、Decoder 等中间结构，最后组装成完整的 Transformer 模型 [1]。整个过程中，还将融入良好的工程实践，包括 Git 版本管理、代码结构规划、注释撰写以及实验日志记录等建议。

第1周：Transformer 原理学习与环境准备

- **理论学习**：首先花时间理解 Transformer 的整体架构和工作机制。阅读并总结 Transformer 论文结构（Encoder-Decoder框架，每侧堆叠多层 Self-Attention 和前馈网络等） [2]。推荐参考中文教程或博客（如动手学深度学习相关章节、知乎专栏文章等）来了解 Encoder 和 Decoder 各模块的作用和数据流。
- **环境搭建**：在你的 MacBook Air 上安装配置好深度学习开发环境。建议使用 PyTorch 框架（确保版本兼容 M1/M2 芯片或使用 CPU 后端）。创建项目的 Git 仓库，在根目录下添加 `README.md` 描述项目目标和计划，以及 `.gitignore` 文件以忽略虚拟环境、数据集等不需要纳入版本控制的文件 [3]。
- **Git 使用**：从第一周开始养成使用 Git 的习惯。每完成一项设置或代码草稿，就进行一次独立 commit，填写清晰的提交说明（commit message），描述本次变更的目的 [4]。遵循常见规范（如使用动词开头简述修改内容），确保日后查看日志能快速理解各次改动。
- **开发工具**：选择顺手的编辑器或IDE（如 VSCode/PyCharm）进行编码，并配置 Python 虚拟环境或 Conda 环境来管理依赖。安装必要的库如 `torch`、`numpy` 等。根据需要安装 Markdown 编辑器或日记工具，用于记录学习笔记和实验日志。

本周目标：清晰理解 Transformer 架构原理和各组成模块功能，搭建好开发环境和Git仓库，撰写项目说明文档，做好后续实现工作的准备。

第2周：实现 Embedding 层与位置编码

- **Embedding 实现**：编写词嵌入层，将输入的 token 序列转换为向量表示。可直接使用 PyTorch 提供的 `nn.Embedding` 实现 [5]。例如，定义 `Embedding(vocab_size, d_model)` 将每个词ID映射到长度为 d_model 的向量。同样准备一个输出端的 Embedding 层用于 Decoder 输入。注意在定义 Embedding 时，可以利用 `padding_idx` 参数指定 <PAD> 的索引，以便模型在计算时自动忽略填充部分 [6]。
- **位置编码 (Positional Encoding)**：实现位置编码模块，为模型提供序列位置信息 [5]。阅读公式理解位置编码的计算：通常使用固定的正弦余弦函数形式，将位置和维度映射到一个向量 [7]。编写一个类 `PositionalEncoding(d_model, max_len)` 预先计算一个尺寸为 `(max_len, d_model)` 的位置编码矩阵（可参考公式直接生成或逐元素计算）。在前向过程中，将该位置编码加到输入的 embedding 张量上。中文教程中通常将 Positional Encoding 看作对 Embedding 结果添加一个固定偏置 [5]。你可以验证位置编码实现的正确性，例如对于特定位置和维度，手工计算与函数输出是否一致。
- **模块测试**：为本周编写的 `Embedding` 和 `PositionalEncoding` 模块各自撰写简单的单元测试或脚本：构造一个小批次的索引序列，通过 Embedding 看输出形状是否为 `[batch, seq_len, d_model]`，再将输出加上位置编码并检查不同位置是否有所区别。考虑将这些测试代码保存在项目的 `tests/` 目录下，方便以后回归测试。
- **代码风格**：从这周开始正式编写代码，务必保持良好的编码风格。函数和类增加清晰的**注释和文档字符串**，解释参数含义和实现逻辑。注释应该补充说明代码的意图，避免简单翻译代码行为 [8]。例如，在

复杂公式实现处注明资料来源，在重要步骤处解释原因。通过良好注释，使代码即使隔一段时间后阅读也能快速理解。

本周目标：完成输入/输出 Embedding 层和 Positional Encoding 模块的实现与自测 [5] 。项目结构上，创建好相应的 Python 文件（如 `modules/embedding.py`，`modules/position.py`），并确保这些模块独立运行和通过简单测试。

第3周：实现 Scaled Dot-Product Attention 和多头注意力机制

- **自注意力机制**：实现 Transformer 的核心——**缩放点积注意力 (Scaled Dot-Product Attention)**。按照公式，对输入的查询矩阵 Q、键矩阵 K、值矩阵 V 计算注意力输出。其中 $Attention(Q,K,V) = \mathrm{softmax}(QK^T/\sqrt{d_k})V$。编写函数完成上述计算，注意在实现中保持张量维度正确，例如 Q 为 `[batch, seq_len_q, d_k]`，K为 `[batch, seq_len_k, d_k]` 等。可以先用单头的注意力机制函数进行测试：人为设置简单的 Q,K,V 检查输出是否符合预期。
- **多头注意力 (Multi-Head Attention)**：在单头注意力基础上，实现多头并行计算 [9] 。定义类 `MultiHeadAttention(n_heads, d_model)`，内部会初始化 W^Q, W^K, W^V 等全连接层，将输入投影为不同子空间，然后调用前述注意力函数。将 n个头的结果拼接并通过输出权重矩阵 W^O 投影回 d_{model}维空间 [10] [11] 。实现关键是正确 reshape 和 transpose 张量：例如，可以将输入张量形状从 `[batch, seq_len, d_model]` 变为 `[batch, n_heads, seq_len, d_k]` 以便并行计算每个头的注意力。
- **Mask 掩码机制**：为注意力机制添加 Mask 功能。在 Decoder 中的自注意力需要 Mask 来防止模型关注后续的词（future tokens） [12] 。实现方法是在计算 QK^T 得到的 score 张量上，根据 mask 将非法位置（True表示需要mask）填充为 $-\infty$ [13] ；Softmax 后这些位置权重即为0，从而达成遮蔽效果。编写 `generate_mask(seq_len)` 函数返回 Decoder 自注意力需要的下三角矩阵 mask，以及用于 Padding 部分的mask。测试时，可构造一个简单序列并手动设定某些 mask，检查经过 masked 注意力后这些位置输出是否确实为0。
- **模块调试**：仔细调试 MultiHeadAttention 的维度变化和mask应用。打印中间张量形状确认与设计一致（例如 Q 投影后应为 `[batch, n_heads, seq_len, d_k]`，score应为 `[batch, n_heads, seq_q, seq_k]`）。如果遇到维度对不齐或形状错误，利用断言和日志及时修正。在保证单个多头注意力模块正确后，再继续下一步。将这个模块加入项目，如 `modules/attention.py`，方便在 Encoder/Decoder 中调用。
- **阅读参考**：本周涉及的注意力机制是 Transformer精髓，建议阅读李沐《动手学深度学习》10.5节和相关博客来深入理解原理。例如，多头注意力通过多个注意力头捕获输入序列不同方面的关系，每个头处理信息的子空间可能不同 [9] 。理解这些概念将有助于编码和调试。

本周目标：实现 **MultiHeadAttention** 模块，包括 scaled dot-product attention 的函数和多头并行机制，并支持可选的 mask 掩码 [11] [13] 。通过简单用例验证其输出合理，为后续集成进 Encoder/Decoder 做好准备。

第4周：实现前馈网络和 Add&Norm，并组装 Encoder 层

- **前馈网络 (Feed-Forward Network)**：实现 Transformer 每层中的逐位置前馈全连接网络。通常包括两层线性变换和中间的非线性激活（如 ReLU）。例如隐藏层规模可设为 $d_\text{ff}=4d_\text{model}$（参照论文）。编写类 `PositionwiseFFN(d_model, d_ff)`，在 forward 中对输入张量形状 `[batch, seq_len, d_model]` 的每个位置独立地通过两层全连接得到相同形状输出 [14] 。可用简单输入测试网络的维度变化和数值合理性。
- **残差连接 + 层归一化 (Add & LayerNorm)**：实现 Add & Norm 模块，将子层输出与原输入相加，然后进行层归一化 [15] 。编写类 `AddNorm(dropout_rate)`，内部先对子层输出应用 `nn.Dropout`（dropout 率可参考配置如0.1），再与原输入相加后通过 `nn.LayerNorm` 正则化 [16] 。需要注意 LayerNorm 在 PyTorch 中默认对最后一维归一化，可直接用 `normalized_shape=d_model` 初始化。阅读论文可知，适当的 Dropout 对 Transformer 模型性能影响显著 [16] ，请确保在训练时启用 Dropout，在评估/推理时切换到 eval 模式关闭 Dropout。

- **组装 Encoder 层**：将上述模块组装成 **EncoderBlock**。Encoder 的每一层包括：多头自注意力 + Add&Norm，然后前馈网络 + Add&Norm [2]。编写 `EncoderBlock(d_model, n_heads, d_ff, dropout)`，在初始化中实例化一个 MultiHeadAttention 和一个 PositionwiseFFN，以及两个 AddNorm模块分别用于注意力子层和前馈子层。Forward 函数中，传入输入张量先经过注意力子层和第一个 AddNorm（注意传入必要的 **mask** 参数用于 padding Mask），然后输出再经过前馈网络和第二个 AddNorm。这样就完成一个 Encoder 层的计算。
- **Encoder 结构**：定义 `Encoder(num_layers)` 类来堆叠多个 EncoderBlock [17] [18]。可以将若干 EncoderBlock 放入 `nn.ModuleList` 方便迭代调用。Forward 中对输入依次应用每个 EncoderBlock，并将最终结果返回。此时 Encoder 可以接收形状 `[batch, seq_len, d_model]` 的张量以及相应的 padding mask，输出同形状的编码表示。
- **代码整合**：整理本周完成的模块代码，将 MultiHeadAttention、PositionwiseFFN、AddNorm、EncoderBlock 等放入合适的模块文件。**撰写注释**解释各步操作，尤其是在 EncoderBlock 中标明每一步对应论文哪部分结构，残差连接和 LayerNorm 放在何处等。通过小规模数据测试 EncoderBlock：例如输入一个张量（随机或来自 Embedding 的输出）和一个简单 mask，看多层 Encoder 堆叠输出形状正确，且逐层输出有变化。

本周目标：完成 **Encoder** 部分的实现：包括单层 EncoderBlock 的逻辑以及多层堆叠的 Encoder 模块 [17] [18]。确保Encoder在给定输入和padding mask时，可以输出编码后的序列表示。代码具备良好注释，方便后续调试和与Decoder的接口对接。

第5周：实现 Decoder 层（含 Mask）并组装完整 Decoder

- **Decoder层结构**：Decoder 的结构与 Encoder 相似，但每层包含两个注意力子层：**Masked 自注意力**（对解码端输入，自身序列的未来位 mask 掉）和**编码器-解码器注意力**（利用Encoder输出的键、值，结合Decoder自身的查询） [19]。每个子层后也都有残差连接和LayerNorm。可以参考资料巩固理解：Decoder第一层自注意力通过Mask防止窥视未来信息，第二个注意力层引入Encoder输出供Decoder查询，从而融合源序列信息 [20]。
- **实现 DecoderBlock**：编写类 `DecoderBlock(d_model, n_heads, d_ff, dropout)`。初始化包括：一个 MultiHeadAttention（或可以重用Encoder的实现）用于 Masked 自注意力，第二个 MultiHeadAttention 用于 Encoder-Decoder 注意力，一个 PositionwiseFFN，以及三个 AddNorm模块（因为有三个子层需要Add&Norm）。Forward 函数参数需要传入 x（Decoder当前层输入）和 enc_output（Encoder输出），以及相应 mask：dec_mask（Decoder自注意力的mask矩阵）和 enc_dec_mask（Encoder-Decoder注意力的mask，一般用于padding遮挡encoder输出序列的无效部分）。实现顺序为：先对 x 应用第一个注意力（`self_attn`，传入 x 的 Query、Key、Value都为 x，以及 dec_mask）再 AddNorm；其输出作为 Query，再对 enc_output 进行第二个注意力（传入 Query=前一步输出, Key=Value=enc_output，以及 enc_dec_mask）再 AddNorm；最后通过前馈网络和第三个 AddNorm [21]。这样得到 DecoderBlock 的输出。
- **Decoder 结构**：和 Encoder 类似，实现 `Decoder(num_layers)` 来堆叠多层 DecoderBlock [22] [23]。保存多个 DecoderBlock 在 ModuleList，Forward 函数中循环每层，将上一层输出和 enc_output 依次传入下一层。在最后得到Decoder最终输出张量。Decoder Forward 需要的 mask参数包括 dec_mask（通常为序列的后向遮挡mask和padding mask的结合）以及 enc_dec_mask（对encoder输出的padding部分进行遮挡）。确保这两个 mask 的形状正确，例如 dec_mask 为 `[batch, 1, seq_len, seq_len]` 下三角矩阵 [24] [25]。
- **Mask 生成**：在 Decoder类中，或者单独编写函数，生成上述两类 mask。典型做法是：给定输入序列张量 `x`（形状 `[batch, seq_len]`），可以通过比较得到 padding mask矩阵 `pad_mask = (x == <PAD>)` [24]；再利用下三角矩阵与 pad_mask 合并得到 `dec_mask` [26]。同时 `enc_dec_mask` 可以通过编码端输入的 pad_mask 来构造，使得Decoder在第二注意力层不考虑Encoder输出中的 <PAD>位置。务必测试 mask 生成函数，针对不同长度输入确保 mask 形状和内容符合预期（例如 `dec_mask` 对角线以上部分应为 True表示遮挡）。
- **组装完整 Transformer**：现在将 Encoder 和 Decoder 连接起来。定义 `TransformerModel` 类封装整体模型，初始化包括：词嵌入层 `embed_src` 和 `embed_tgt`，PositionEncoding层 `pos_enc`（若设计为

单独类，也可在 Embed 时加），Encoder 和 Decoder 实例，以及最后的输出层（线性层映射到目标词表大小）。Forward过程：先对源序列索引通过 `embed_src` (加 pos_encoding) 得到编码输入，对目标序列通过 `embed_tgt` (加 pos_encoding) 得到解码器输入，然后将源编码输入喂入 Encoder 得到 `enc_output`，再将解码器输入和 `enc_output` 喂入 Decoder，最后通过线性层将 Decoder输出转为词表维度并取 softmax 概率 [27] [28]。在这个过程中需要生成并传递前述各种 mask：源序列的 pad_mask 给 Encoder，目标序列的 `dec_mask` 和 `enc_dec_mask` 给 Decoder [29] [30]。整合时注意维度匹配，例如确保 Embed 输出的维度 d_model 与 Encoder/Decoder一致。

- **模块调试**：组装好模型后，用小规模数据跑一次前向传播检查有没有维度或索引错误。可以设计一个极小的示例（比如词表大小仅几个词，序列长度很短）人工构造输入，打印 TransformerModel 输出是否合理（比如对于输入序列希望模型输出序列长度相同且每个位置得到一个 `vocab_size` 概率分布）。如果前向传播通过，则说明模型结构基本正确。
- **代码版本管理**：本周集成了 Encoder 和 Decoder，代码量大增。请确保在实现完每部分后及时 git commit。尤其在调通整个模型前向传播后，做一次里程碑式提交，并附带简要消息如“Complete Transformer forward pass (encoder-decoder integrated)”。这有助于将当前稳定版本记录下来，便于将来回溯。

本周目标：完成 **Decoder** 各模块和完整 Transformer 模型的实现，支持将源句子转换为目标句子的概率分布输出。经过本周，模型的结构和前向流程已全部打通 [31] [27]。为下周开始的模型训练奠定基础。

第6周：数据准备与简单训练实验

- **准备训练数据**：选择一个简单的序列到序列任务数据集进行实验。由于硬件限制，建议使用**小规模数据集**。可以采用公开的迷你中英翻译数据集（例如只有几千句的中英平行语料）或自定义的玩具数据。例如，你可以人工构造一个简单映射任务：输入为一串字符，输出为其有规则变换后的字符串（如将字符串倒序排列）来验证模型是否能学到规律 [32] [33]。若希望使用真实翻译数据，可参考一个仅几KB大小、中英词表约1万的简易数据集 [34]（如周弈帆博客提到的资料）。关键是数据量要小到可以在有限资源上训练，同时又足以体现模型功能。
- **数据预处理**：编写脚本加载平行语料，对句子进行分词和建立词表（如果使用自带分词的简易数据，可直接读取词表）。将文本转换为索引序列，过滤过长的句子，添加句首 `<BOS>` 和句尾 `<EOS>` 标记，处理 OOV 为 `<UNK>` 等。然后对序列进行**padding**到统一长度。可以利用 PyTorch 的 `Dataset` 和 `DataLoader` 来管理数据批次，例如编写 `TranslationDataset` 类读取处理后的 `(src, tgt)` 对，并用 `DataLoader` 实现批量迭代。注意在 `collate_fn` 中实现动态padding，使每个batch按该batch最大长度pad，而不要用全局最大长度浪费计算。
- **Git 分支管理**：在开始编写训练代码前，建议创建一个新的 Git 分支（例如 `training-dev`），用于开发训练相关功能。这样即使调参或修改过程中出现问题，主分支的模型实现部分仍保持完整。定期将稳定的修改合并回主分支。通过分支管理隔离实验代码，也是良好实践。
- **训练脚本**：实现模型训练循环（可放在 `train.py` 或笔记本中）。步骤包括：初始化模型参数，设置优化器（如 Adam）和损失函数（使用交叉熵损失函数 **CrossEntropyLoss**，并忽略 `<PAD>` 标签的损失贡献，可以设置 `ignore_index` 为 PAD索引）。在每个 epoch 内，遍历训练数据集：对每个 batch 调用模型前向计算得到预测分布，用 `loss_fn` 计算损失（注意将目标序列右移一位作为模型输出的比较对象），再反向传播梯度并调用优化器 `step()` 更新参数。每隔一定步数打印损失或计算在验证集上的指标。还可加入梯度裁剪以防止梯度爆炸。
- **短暂训练**：从小批量和少量 epoch 开始训练，观察模型是否收敛。例如，用一个很小的数据集（或者仅取训练集的一部分）进行**过拟合测试**——如果模型能在该数据上达到极低的损失，说明模型和训练流程基本正确。过程中注意 **<PAD> 字符的处理**：它不应对损失和梯度产生影响 [35]。验证方式是观察损失计算是否忽略了PAD填充的位置。
- **日志记录**：在训练脚本中引入**日志记录**机制。使用 Python 的 `logging` 模块或简单的打印，将每轮(epoch)的损失、准确率等保存到日志文件。 [36] 建议将日志同时输出到控制台和文件，文件保存在 `logs/` 目录并按日期或实验名称命名。这有助于你跟踪训练进度，事后分析不同参数配置的效果。如果愿意，可以使用 TensorBoard 等可视化工具跟踪损失曲线和模型参数分布变化。

- **实验日志**：开始建立“实验日志册”。每次尝试新的超参数配置或训练方案时，在日志中记录：所用数据集、模型超参数（层数、头数、d_model等）、优化器参数（学习率、batch大小等）、以及训练结果摘要（最终训练/验证损失，特殊现象）。可以使用 Markdown 文件或Excel表格记录。良好的实验记录有助于后续分析和避免重复踩坑 [37] [38] 。

本周目标：准备并预处理一个小型平行语料，搭建基本的训练循环，在该数据上成功跑通模型训练一小段时间，观察到损失下降。确保 **PAD** 的处理正确无误，模型能够在小数据上正常学习对齐关系或简单规律。完成训练代码后，将其和预处理脚本一同提交至Git仓库。

第7周：模型正式训练与性能调优

- **迁移到GPU环境**：在具备 RTX 3090 的 CentOS 服务器上配置运行环境（安装对应的CUDA版本、驱动和PyTorch GPU版本）。将上周实现的模型和训练脚本迁移过去（通过 Git 拉取最新代码）。用 `.to(device)` 方法将模型、张量转移到 GPU 上进行计算，以加速训练 [39] 。先在GPU上跑一个batch测试确保没有CUDA相关报错。
- **完整数据训练**：如果小数据实验效果良好，本周尝试在稍大一些的数据集上训练模型。例如使用前述的简易中英翻译数据全集（如果之前只取了子集）。设定适当的 epoch 数，让模型在GPU上充分训练。监控训练过程中的损失曲线是否收敛。如果发现模型在训练集上收敛但验证集表现变差，考虑采取**早停(early stopping)**策略或减少epoch防止过拟合。
- **优化技巧**：尝试 Transformer 论文中的优化技巧，例如**学习率预热和衰减**（warm-up）。论文采用了先warm-up然后下降的学习率调度策略。你可以使用 PyTorch 的 `LambdaLR` 实现类似调度器，在前 N 步内线性增加学习率，然后按 sqrt(d_model) 比例衰减。观察使用调度器后模型收敛是否更平稳。
- **梯度累积**：如果显存有限无法增大 batch，可以尝试 **Gradient Accumulation**（梯度累积）手段，即累积多个小批次的梯度再更新权重，相当于提高有效批次大小。实现时，可在代码中每隔几步调用一次 `optimizer.step()` 并zero grad。注意调整学习率步长相应缩放。
- **模型保存**：在训练过程中定期保存模型检查点（checkpoint）。使用 `torch.save(model.state_dict(), filepath)` 保存当前权重，以便中断后恢复或选择最佳模型。保存策略可以是每个epoch保存一次，或者根据验证集指标保存最佳模型。请将 checkpoint 文件存储在项目的 `checkpoints/` 文件夹，并将其路径加入 `.gitignore` （无需纳入版本库）。
- **观察与记录**：训练期间密切关注日志输出，尤其是**注意力权重**是否产生预期行为。可以选取几个时间步的注意力矩阵输出，打印或可视化，直观检查模型是否在对齐相关位置（例如翻译任务中，注意力是否对齐中英文对应词）。这虽然不是严格量化指标，但有助于深化对模型工作的直觉。如果发现训练有问题（如损失不下降或Nan），及时中止并利用日志和代码检查问题，比如梯度爆炸、数据处理错误等。
- **实验记录**：将本次大规模训练的结果详细记录到实验日志中，包括最终模型在训练集和验证集上的 Loss、准确率或 BLEU 分数等指标，以及训练用时、超参数设置。当模型达到满意性能时，记录下作为里程碑的一次实验配置。若效果不理想，也记录问题并思考可能的改进，下次实验如何调整。

本周目标：在完整数据集上成功训练 Transformer 模型并获得初步可用的模型参数。掌握在 GPU 上调优模型的流程，包括学习率调度、梯度累积等技巧。输出模型在验证集上的性能指标，保存训练好的模型权重备用。

第8周：模型评估与推理测试

- **模型评估**：使用保留的验证集或测试集评估模型效果。例如对翻译任务，计算 BLEU 分数或简单地人工检查翻译结果质量。编写评估脚本 `evaluate.py` ，读取测试集中若干句子，通过模型生成翻译结果。由于我们的模型目前是基础的 Teacher Forcing 训练，需要在推理时**自回归生成**译文。实现方法：给定源句子，先获取 Encoder 输出；然后循环地用 `<BOS>` 作为初始输入喂给 Decoder，每步取输出概率最高的词（或采样），作为下个时刻输入，直到输出 `<EOS>` 或达到最大长度 [40] [41] 。得到完整输出序列后，与参考翻译对比。你可以实现一个函数 `translate(sentence)` 来封装上述逻辑，方便地生成单句翻译。

- **测试推理**：挑选几例测试数据，通过 `translate()` 函数看模型翻译是否合理。例如简单句子的主谓宾是否正确翻译，有无明显遗漏。若有条件，可以计算 BLEU、ROUGE 等指标量化性能。由于我们的模型可能在小数据上训练，翻译质量有限，但至少应看到模型学到一定对应关系。
- **错误分析**：分析模型输出的常见错误。如果发现翻译结果有系统性偏差，如常把某个词翻译错、或句子过短/过长，思考可能原因（数据分布、模型容量等）。例如，如果模型输出普遍偏短，可能是长度惩罚未考虑；若专有名词翻译错，可能是未登陆词处理问题。这些发现可以帮助指导下一步改进，例如扩充数据、调整超参数或引入长度控制机制等。
- **代码优化与整理**：在确保模型运行正确后，本周花时间**重构和整理代码**。检查是否有冗余或不规范之处，例如魔术数字、重复代码，将其提取成函数或配置。在整个项目中贯彻**模块化设计**思想，确保各模块职责单一、命名清晰。完善代码注释和文档，对重要类和函数补充 docstring 描述其用途和参数。清理掉调试用的打印或多余的代码片段。
- **工程文档**：编写项目的 **README** 或用户手册，详细说明项目的用途、实现细节、运行方法。README 可以包括：项目背景介绍、模型架构简述、依赖安装方法、训练和推理的使用示例、结果展示等。这样不仅巩固自己的理解，也方便日后参考或他人使用。
- **版本管理**：将本周的重要改动提交到 Git。你可以考虑创建 pre-release 标签，如 `v1.0`，标记当前实现已完成主要功能。养成撰写**清晰的Release说明**的习惯，总结此版本的功能和性能。Git 提交信息依然要规范书写，清楚传达每次修改内容。通过良好的版本管理，你的项目将具有可追溯性和发布记录。

本周目标：完成模型在测试集上的评估，能够进行基本的**文本推理**得到翻译结果。分析模型表现并整理改进思路。同时对项目代码和文档进行扫尾完善，使仓库具备良好的可读性和可用性。

第9周：拓展练习与深入学习

- **超参数调优**：尝试调整一些超参数观察对模型的影响。例如改变 Transformer 层数、注意力头数或前馈层维度，查看模型性能变化。由于时间有限，可有针对性地尝试一两项（比如层数从6减少到4看看性能变化）。将不同设置的结果记录在实验日志中，巩固对模型容量与效果关系的认识。
- **不同任务迁移**：若对其他任务感兴趣，可用现有代码**迁移**到类似的序列任务上。例如把模型应用到简单的**摘要生成**任务（数据为文章-摘要对），或者**序列标注**任务上（将 Transformer 做编码器，连接分类层输出标签）。尝试少量修改验证模型的通用性。但注意在新任务上可能需要调整数据处理和输出层等部分。
- **阅读前沿**：利用这周时间拓展理论知识，阅读 Transformer 相关的进阶研究。如**BERT**、**GPT** 等预训练模型架构，了解它们在标准Transformer基础上的改动（如去掉Decoder、双向训练等）。也可关注Transformer在CV领域的应用（Vision Transformer）等扩展。通过阅读博客或论文综述，更新自己对大模型最新进展的认知 [42] 。这些阅读有助于从更高角度理解你实现的模型在AI发展中的位置。
- **项目优化**：考虑为项目引入更完善的工程工具。例如，添加 **单元测试** 来覆盖关键模块（可以使用 `pytest` 框架编写测试用例）；配置 **CI流水线** 实现自动跑测试；或者整合 **参数配置文件**（如用 `yaml` 保存超参数，使用 `argparse` 自动读取）来方便地切换不同实验配置 [38] [36] 。这些工程实践能进一步提高项目的质量和开发效率。
- **分享与总结**：将你的学习成果整理成博客或报告。在知乎、CSDN 或个人博客撰文介绍手写Transformer的经历，包括遇到的挑战、踩过的坑（比如 `<pad>` mask 的处理非常关键 [43] ）、以及收获的经验。这不仅帮助梳理知识，也能让他人受益并给你反馈。通过讲解给别人，你对Transformer的理解会更加深刻。

本周目标：在巩固已有成果的基础上探索进阶方向。通过调参实验和不同任务的简单迁移，验证模型与代码的泛化能力；通过阅读和分享，总结自己的**Transformer实践心得**，为今后更复杂的深度学习项目打下基础。

第10周：项目收尾与良好开发习惯巩固

- **代码回顾 (Code Review)**：最后一周，对整个项目代码进行一次系统的走查和重构。检查每个模块是否遵循了**单一职责原则**，函数是否过长需要拆分，变量命名是否清晰规范。确保不存在明显的性能隐患和bug。尝试从零运行一遍项目（从数据处理到训练到推理）来验证代码的健壮性。

- **注释和文档**：完善所有代码文件的头部说明和关键段落注释。遵循最佳实践给代码添加注释：注释应解释代码意图而非冗余重复代码含义，不能用注释掩盖糟糕的代码设计 [8]。对于复杂逻辑，在注释中引用参考资料链接或论文公式编号 [44]，方便读者深入了解背景。编写开发文档，说明项目的模块组织、主流程和如何继续扩展。
- **Git 历史整理**：查看 Git 提交历史，根据需要**整理分支**。将完成的功能合并回主分支，删除临时分支。给最终的主分支打上版本标签（如 `v1.0.0`）。如果某些杂乱的调试提交对他人阅读历史造成困扰，学习使用 `git rebase -i` 交互式压缩整理提交（注意谨慎操作避免破坏历史）。撰写简洁的Release说明，概括项目内容和使用方法。
- **良好习惯反思**：对照本计划开头的目标，总结自己在编程能力、深度学习模型细节、Git使用和项目管理等方面的提升。从中提炼出几条今后工作的准则，如“保持小步提交并写明变化动机”、“定期记录实验过程”、“编写自解释的代码避免过度依赖注释”等。这些习惯的养成将大大有益于你未来的开发工作。
- **下一步计划**：思考项目的下一步可能方向。比如尝试引入 **多头注意力可视化工具** 来观察注意力分布、或将模型部署到简单的服务接口上供演示。如果对性能不满意，可以计划使用更大规模数据再训练，或者尝试结合现有框架（如 HuggingFace Transformers）验证手写模型的正确性。制定简单的后续学习计划，保持进步。

本周目标：圆满收尾 Transformer 手写实现项目。代码质量和工程结构达到较高水平，项目经过清理和文档完善后具备良好的可读性和可维护性。你不仅成功实现了从零搭建并训练 Transformer 模型的全过程，还在实践中养成了良好的编程和项目管理习惯，为独立完成更大型的深度学习项目打下坚实基础。

---

[1] [2] [19] 使用Pytorch从零实现Transformer模型_transformer pytorch-CSDN博客
https://blog.csdn.net/xu1129005165/article/details/137586535

[3] 各类Python项目的项目结构及代码组织最佳实践_python项目结构-CSDN博客
https://blog.csdn.net/captain5339/article/details/128017400

[4] Git Commit Message 最佳实践-腾讯云开发者社区-腾讯云
https://cloud.tencent.com/developer/article/2339119

[5] [9] [15] 大模型基础——从零实现一个Transformer(1)_从零实现transformer-CSDN博客
https://blog.csdn.net/fan_fan_feng/article/details/139484057

[6] [7] [10] [11] [12] [14] [16] [17] [18] [21] [22] [23] [27] [28] [31] 熬了一晚上，我从零实现了Transformer模型，把代码讲给你听-腾讯云开发者社区-腾讯云
https://cloud.tencent.com/developer/article/1890943

[8] [44] 建议收藏：编写代码注释的最佳实践 - 环信
https://www.easemob.com/news/6973

[13] [20] [24] [25] [26] [29] [30] [32] [33] [40] [41] Transformer学习-最简DEMO实现字符串转置_transformer demo-CSDN博客
https://blog.csdn.net/m0_61864577/article/details/137291065

[34] [35] [43] PyTorch Transformer 英中翻译超详细教程 | 周弈帆的博客
https://zhouyifan.net/2023/06/11/20221106-transformer-pytorch/

[36] [37] [38] [39] 高效管理深度学习实验 - pprp - 博客园
https://www.cnblogs.com/pprp/p/14869872.html

[42] 三万字最全解析！从零实现Transformer（小白必会版） - 知乎专栏
https://zhuanlan.zhihu.com/p/648127076