9 大语言模型

徐辉，xuh@fudan.edu.cn

本章学习目标：

⋆⋆⋆ 理解词嵌入（Word Embedding）的基本原理与作用

⋆⋆⋆ 理解注意力机制（Attention）的核心思想

⋆⋆⋆ 理解基于 Decoder-only Transformer 的语言建模方法、自回归生成过程及其实现

大语言模型是一类以语言建模任务为核心、在大规模语料上训练的神经网络模型，其目标是通过自回归方式学习自然语言中词元（token）之间的统计与语义结构，从而实现文本的理解与生成。在实践中，现代大语言模型通常采用 Decoder-only Transformer 架构，并通过预测下一个词元来进行训练。

9.1 词元

9.1.1 词嵌入

对语言进行建模的第一步是将语言转化为可计算的数学表示。由于神经网络只能处理数值输入，自然语言中的词或符号需要被映射到向量空间中。这种将离散符号映射为连续向量表示的方法称为词嵌入（word embedding）。在词嵌入中，每个词元被表示为一个 d 维实向量，从而嵌入在同一个向量空间 $\mathbb{R}^d$ 中。

在最原始的表示方式中，一个词元可以用 one-hot 向量表示。设词表大小为 V，则每个词元被表示为一个 V 维向量，其中仅有一个位置为 1，其余为 0。然而，这种表示方式存在明显缺陷：维度高且稀疏，且不同词之间没有任何语义关联。为了解决这一问题，引入了词嵌入。词嵌入本质上可以看作是对 one-hot 向量的一次线性变换。设 embedding 矩阵为 $W \in \mathbb{R}^{V\times d}$，则一个 token i 的向量表示可以写为：

$$\mathbf{e}_i = \mathbf{o}_i W$$

其中 $\mathbf{o}_i$ 是对应的 one-hot 向量。由于 $\mathbf{o}_i$ 只有第 i 个位置为 1，上式等价于直接取矩阵 W 的第 i 行。因此，在实际实现中并不会显式构造 one-hot 向量，而是通过查表操作获得对应的 embedding。词嵌入的关键在于其“稠密性”和“可学习性”。与 one-hot 表示不同，embedding 向量是低维且稠密的，并且其数值是通过训练自动学习得到的，从而能够在向量空间中编码词语之间的语义与语法关系。

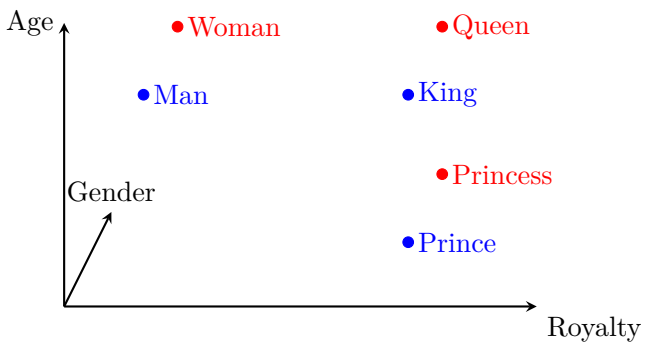

图 9.1: 词嵌入示意图

如果词嵌入的效果足够好，不同词之间的语义与语法关联关系可以在向量空间中得到体现。例如，在英语中，“King - Man + Woman ≈ Queen”，表示“King”和“Queen”之间的性别关系可以通过向量差来表达。同样，King – Prince ≈ Queen – Princess，表示成年与年轻之间的年龄关系在王室词汇中也能被捕捉。

那么，如何构建这样的向量空间，并将离散的词或 token 映射到其中呢？核心思想是：不人为规定词向量的具体含义，而是通过学习任务让模型自动发现这些表示。在现代大语言模型中，词嵌入一般不再是独立训练的模块，而是作为整个模型的一部分进行端到端训练。具体来说，embedding 矩阵 W 被视为模型的第一层参数，与后续的神经网络共同优化。在训练过程中，模型通常基于语言建模目标（如预测下一个 token 的概率），通过最小化损失函数来更新所有参数，包括 embedding。

值得注意的是，embedding 只是信息表示的起点。经过多层非线性变换（例如自注意力机制），每个词元会被映射为上下文相关的表示。因此，同一个词在不同语境中的最终向量表示是不同的。这种上下文相关的稠密向量才是大模型真正用于理解和生成语言的核心表示。

9.1.2 分词

在大语言模型中，词嵌入的输入单位通常并不是完整单词。词表（vocabulary）中既包含单个字符，也包含完整单词或更细粒度的子词（subword）。在实际应用中，现代大模型普遍采用基于子词的分词方法，其核心思想之一是“最长匹配”：在词表中优先寻找能够匹配当前文本前缀的最长词元。

例如，假设词表中同时存在：

`p`, `l`, `a`, `y`, `i`, `n`, `g`, `play`, `ing`

当输入字符串为：

`playing`

时，分词器会优先匹配最长的词元 `play` 和 `ing`，而不是先匹配较短的 `p`。

这种分词方式的好处是在有限词表规模下，同时兼顾以下几个方面：

- 高频词能够整体保留；
- 低频词能够拆分为更小的子词；
- 未登录词（OOV）仍然可以被表示；
- 对拼写错误或错词具有一定鲁棒性。

9.2 注意力机制

Transformer 的核心是注意力（Attention）机制，其思想是通过计算序列中各词元之间的相关性，实现信息的加权聚合。在该机制中，每个输入向量（词嵌入）会被映射为 Query（Q）、Key（K）和 Value（V），并基于 Query 与 Key 的相似度计算注意力权重，从而对 Value 进行加权求和。标准的缩放点积注意力定义为：

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

其中，

$$Q \in \mathbb{R}^{n\times d_k}, \quad K \in \mathbb{R}^{n\times d_k}, \quad V \in \mathbb{R}^{n\times d_v},$$

n 表示序列长度，d_k 表示 Query/Key 的特征维度，d_v 表示 Value 的特征维度。缩放因子 $\sqrt{d_k}$ 用于避免点积结果随维度增大而过大，从而稳定 softmax 的梯度分布。

设输入序列为若干词元表示 $x_1, x_2, ..., x_n \in \mathbb{R}^{d_{\text{model}}}$，其中 d_{model} 是词嵌入向量的维度。在进入注意力计算之前，每个 token 通过三组线性映射分别生成 Query、Key 和 Value：

$$\begin{aligned} q_i &= x_i W_Q, \\ k_i &= x_i W_K, \\ v_i &= x_i W_V, \end{aligned}$$

其中投影矩阵为：

$$W_Q \in \mathbb{R}^{d_{\text{model}}\times d_k}, \quad W_K \in \mathbb{R}^{d_{\text{model}}\times d_k}, \quad W_V \in \mathbb{R}^{d_{\text{model}}\times d_v}.$$

因此：

$$q_i \in \mathbb{R}^{d_k}, \quad k_i \in \mathbb{R}^{d_k}, \quad v_i \in \mathbb{R}^{d_v},$$

即在单个 attention head 中，Query/Key 被映射到用于相似度计算的子空间 $\mathbb{R}^{d_k}$，而 Value 位于 $\mathbb{R}^{d_v}$ 中用于信息聚合。

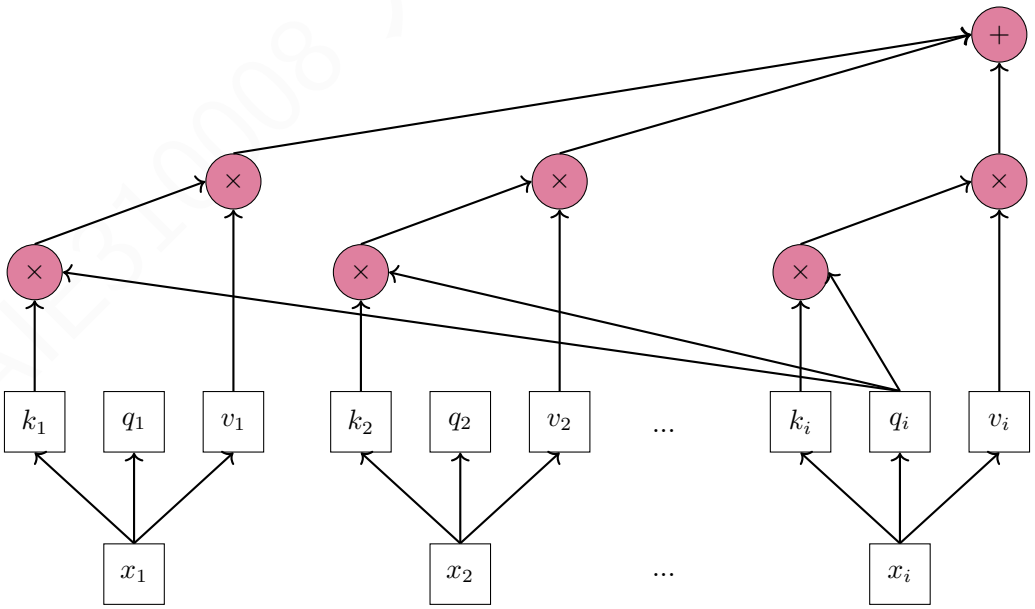

图 9.2: Token x_i 的注意力计算过程

图 9.2 展示了以第 i 个词元的嵌入向量为 Query 的注意力计算过程，具体步骤如下：

1) 设输入序列为 $x_1, x_2, \ldots, x_i \in \mathbb{R}^{d_{\text{model}}}$。通过线性映射得到：

$$q_i = x_i W_Q,$$

并对所有历史 token $j \le i$ 生成 Key 和 Value：

$$k_j = x_j W_K, \quad v_j = x_j W_V,$$

其中 $q_i, k_j \in \mathbb{R}^{d_k}$，$v_j \in \mathbb{R}^{d_v}$。

2) 计算 Query 与 Key 的点积相似度：

$$s_{ij} = q_i \cdot k_j,$$

图中紫色 $\times$ 节点表示该运算；

3) 通过 softmax 归一化得到注意力权重：

$$\alpha_{ij} = \text{softmax}\left(\frac{s_{ij}}{\sqrt{d_k}}\right), \quad \sum_{j=1}^{i} \alpha_{ij} = 1;$$

4) 对 Value 进行加权求和，得到输出：

$$\text{out}_i = \sum_{j=1}^{i} \alpha_{ij} v_j \in \mathbb{R}^{d_v},$$

图中紫色 $+$ 节点表示该聚合过程。

代码 9.1使用 PyTorch 实现了注意力机制，完整模拟了缩放点积注意力的计算流程。该实现首先通过三个独立的线性映射将输入特征投影为 Query、Key 和 Value 表示，然后通过矩阵乘法计算 Query 与 Key 之间的相似度得分，并进行缩放以提高数值稳定性。在引入因果掩码后，模型被限制为只能关注当前位置及其之前的历史信息，从而满足自回归建模的约束条件。随后，对相似度分数进行 softmax 归一化以获得注意力权重，并对 Value 向量进行加权求和得到最终输出表示。

```python
class SelfAttention(nn.Module):
    def __init__(self, d_model=256, d_k=256, d_v=256):
        super().__init__()

        # Query / Key / Value 的线性投影
        # W_Q: (d_model -> d_k)
        # W_K: (d_model -> d_k)
        # W_V: (d_model -> d_v)
        self.q_proj = nn.Linear(d_model, d_k)
        self.k_proj = nn.Linear(d_model, d_k)
        self.v_proj = nn.Linear(d_model, d_v)

        # 输出映射 (d_v -> d_model)
        self.out_proj = nn.Linear(d_v, d_model)

    def forward(self, x):
        # x: (B, T, d_model)
        B, T, _ = x.shape

        # 1. 线性映射得到 Q, K, V
        Q = self.q_proj(x)  # (B, T, d_k)
        K = self.k_proj(x)  # (B, T, d_k)
        V = self.v_proj(x)  # (B, T, d_v)

        # 2. 计算注意力分数 QK^T
        scores = Q @ K.transpose(-2, -1)  # (B, T, T)

        # scaled dot-product
        scores = scores / (Q.shape[-1] ** 0.5)  # sqrt(d_k)

        # 3. 因果掩码：对于第 i 行，只允许看到 j <= i 的位置
        mask = torch.tril(torch.ones(T, T, device=x.device)) # 生成 T×T 的下三角全 1 矩阵
        scores = scores.masked_fill(mask == 0, float("-inf"))

        # 4. softmax
        attn = F.softmax(scores, dim=-1)  # (B, T, T)

        # 5. 加权求和
        out = attn @ V  # (B, T, d_v)

        # 6. 输出投影
        return self.out_proj(out)  # (B, T, d_model)
```

代码 9.1: 基于 PyTorch 的自注意力实现

9.3 Decoder-only Transformer

Decoder-only Transformer 是当前大语言模型（Large Language Models, LLMs）的主流架构，其核心特点是仅由 Transformer 的解码器（Decoder）堆叠而成，并通过自回归方式进行序列建模。在该结构中，模型以自左向右的方式逐步预测下一个词元，即在给定历史上下文 $x_{1:i-1}$ 的条件下建模 $P(x_i \mid x_{<i})$。

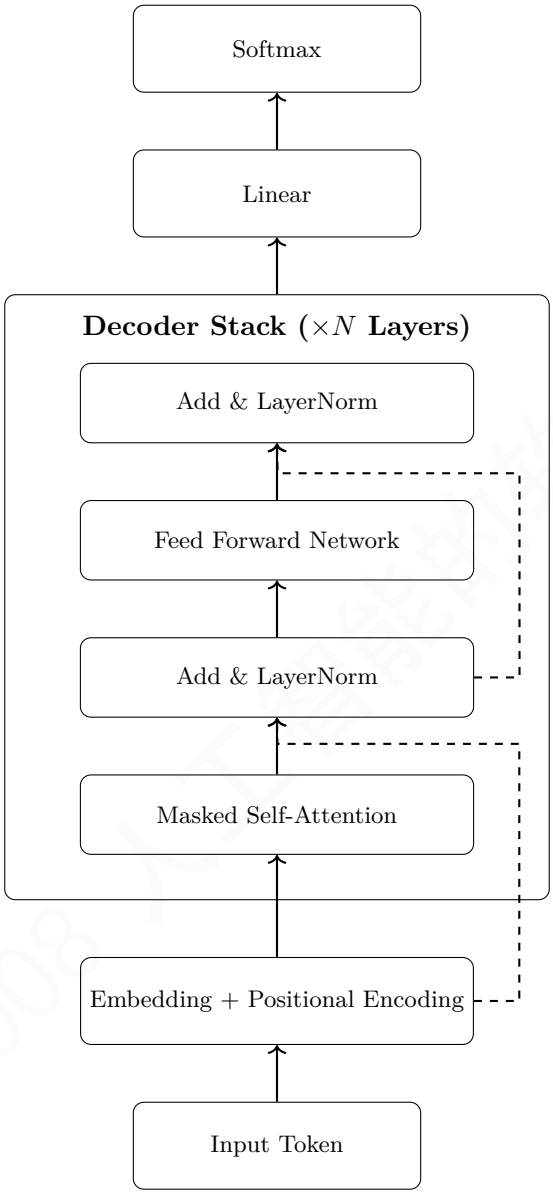

图 9.3: Decoder-only Transformer 结构示意图

如图 9.3所示，Decoder-only 结构由多层相同的 Decoder Block 堆叠而成。每个 Block 包含 Masked Self-Attention 与 Feed-Forward Network 两个子层，并结合残差连接与 LayerNorm 以稳定深层网络的训练过程。其中，Masked Self-Attention 通过因果掩码限制信息流，使得每个 token 仅能访问自身及其之前的历史 token，从而满足自回归建模的条件。

代码 9.2给出了单个 Decoder Block 对应的 PyTorch 实现，其结构与图 9.3严格一致。

```python
class DecoderBlock(nn.Module):
    def __init__(self, d_model=256, d_k=256, d_v=256, d_ff=1024):
        super().__init__()

        # Masked Self-Attention
        self.attn = SelfAttention(d_model, d_k, d_v)

        # Feed-Forward Network
        self.ffn = FeedForward(d_model, d_ff)

        # LayerNorm (Post-Norm)
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x):
        # x: (B, T, d_model)

        # 1. Attention → Residual → LayerNorm
        x = self.ln1(x + self.attn(x))

        # 2. FFN → Residual → LayerNorm
        x = self.ln2(x + self.ffn(x))

        return x


class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        # Position-wise FFN: (d_model → d_ff → d_model)
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

    def forward(self, x):
        # x: (B, T, d_model)
        return self.net(x)
```

代码 9.2: Decoder Block 的 PyTorch 实现（Post-Norm 结构）

代码 9.3给出了 Decoder-only Transformer 的整体实现。在输入端，模型首先将离散的词元序列映射为连续的向量表示，并叠加位置信息以引入序列顺序。其中，token embedding 的实现可视为对 one-hot 表示的一种高效替代：设词表大小为 V，embedding 矩阵为 $W \in \mathbb{R}^{V\times d}$，则一个词元的 one-hot 向量与 W 相乘即可得到其向量表示，而在实际实现中，`nn.Embedding` 通过查表操作直接取出矩阵对应行，从而避免显式构造 one-hot 向量并提高计算效率。随后，表示向量依次通过多个 Decoder Block 进行逐层特征变换。在输出端，模型通过 LayerNorm 对最终隐藏状态进行归一化，再映射到词表空间得到 logits 分布，用于预测下一个词元的概率分布。

```python
class DecoderOnlyTransformer(nn.Module):
    def __init__(
        self, vocab_size, d_model=256, d_k=256, d_v=256, d_ff=1024, num_layers=6, max_len=512
    ):
        super().__init__()

        # 词嵌入：将离散 token id 映射为连续向量表示；one-hot 的等价实现（查表代替矩阵乘法）
        self.token_emb = nn.Embedding(vocab_size, d_model)

        # 位置嵌入：为序列位置提供可学习的位置信息
        self.pos_emb = nn.Embedding(max_len, d_model)

        # Decoder Blocks 堆叠
        self.blocks = nn.ModuleList([
            DecoderBlock(d_model, d_k, d_v, d_ff) for _ in range(num_layers)
        ])

        # 输出层（映射到词表空间）
        self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, idx):
        # B 为 batch size，T 为序列长度
        # idx_{b,t} 表示第 b 个样本第 t 个位置的 token id，取值范围为 [0, vocab_size)
        B, T = idx.shape

        # 位置索引：(B, T)
        pos = torch.arange(T, device=idx.device).unsqueeze(0) # 生成 [[0, 1, 2, ..., T-1]]
        pos = pos.expand(B, T) # 扩展到 (B, T)

        x = self.token_emb(idx) + self.pos_emb(pos)

        # 通过多层 Decoder Block
        for block in self.blocks:
            x = block(x)

        # logits: (B, T, vocab_size)
        logits = self.lm_head(x)

        return logits
```

代码 9.3: Decoder-only Transformer 的 PyTorch 实现

9.4 模型训练与测试

9.4.1 模型训练

本节实现基于 Decoder-only Transformer 的语言模型训练流程。模型采用下一个词元预测作为训练目标，通过最大化序列的条件概率分布 $P(x_t \mid x_{<t})$ 来学习语言建模能力。训练过程主要分为两个阶段：

- 数据准备：对原始文本进行词元化，并构造固定长度的训练样本；
- 模型训练：基于定长数据样本进行自回归训练。

代码 9.4实现了数据读取与训练样本构造过程。首先从本地文件 `shakespeare.txt`[1] 中读取原始语料，并基于字符级 tokenizer 将文本映射为离散的 token id 序列，从而将自然语言转化为可计算的整数表示。在此基础上，采用滑动窗口方式从长序列中截取固定长度为 T（即 block_size）的连续片段，构造训练样本对 (x, y)。其中输入序列 x 表示当前位置的上下文 token 序列，而目标序列 y 为其在时间维度上右移一位后的对齐结果，即 $y_t = x_{t+1}$，用于监督模型学习 next-token prediction 任务。最后，DataLoader 将这些样本组织为 mini-batch 形式，以支持并行计算与稳定训练，从而提升模型在大规模语料上的优化效率。

```python
with open("shakespeare.txt", "r", encoding="utf-8") as f:
    text = f.read()

chars = sorted(list(set(text))) # 字符级 tokenizer
vocab_size = len(chars)

char2idx = {ch: i for i, ch in enumerate(chars)}
idx2char = {i: ch for i, ch in enumerate(chars)}

# 文本 → token id 序列
data = torch.tensor([char2idx[ch] for ch in text], dtype=torch.long)

# 构造训练样本 (x, y)
class NextTokenDataset(Dataset):
    def __init__(self, data, block_size):
        self.data = data
        self.block_size = block_size

    def __len__(self):
        return len(self.data) - self.block_size - 1

    def __getitem__(self, idx):
        x = self.data[idx : idx + self.block_size] # 从长序列中截取长度为 block_size 的输入序列 x
        y = self.data[idx + 1 : idx + self.block_size + 1] # 目标序列 y 是 x 的“右移一位版本”
        return x, y

block_size = 128
batch_size = 32
dataset = NextTokenDataset(data, block_size)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

代码 9.4: 文本数据读取与训练样本构造

[1] https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

代码 9.5实现了 Decoder-only Transformer 的训练过程。模型接收输入序列 x 并输出每个位置的词表 logits，通过交叉熵损失函数与目标序列 y 计算误差，并使用 AdamW 优化器进行参数更新。整个训练过程以 mini-batch 方式进行，从而提升训练效率与稳定性。

```
model = DecoderOnlyTransformer(vocab_size=vocab_size, d_embed=256, d_ff=1024,
        num_layers=6, max_len=block_size).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4, weight_decay=1e-2)
num_epochs = 10

# 训练循环
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0

    for x, y in dataloader:
        x = x.to(device)
        y = y.to(device)

        # 前向传播
        logits = model(x)  # (B, T, V)

        # Cross Entropy Loss
        loss = F.cross_entropy(
            logits.view(-1, logits.size(-1)),  # (B*T, V)
            y.view(-1)                         # (B*T,)
        )

        # 反向传播与参数更新
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch {epoch+1}, loss = {total_loss / len(dataloader):.4f}")
```

代码 9.5: Decoder-only Transformer 的训练过程

9.4.2 模型测试

在模型训练完成后，我们通过自回归方式进行文本生成测试。代码 9.6实现了基于贪心解码的生成过程。模型首先将输入提示文本编码为词元序列，然后在上下文窗口 T 的限制下逐步预测下一个词元，并将预测结果拼接回输入序列中，实现逐步生成。生成过程可在达到最大生成步数或满足终止条件（如生成 EOS token）时提前停止。

```python
def generate(model, prompt, char2idx, idx2char, max_new_tokens=200, block_size=128):
    model.eval()

    # 将输入文本编码为 token id 序列
    idx = torch.tensor([char2idx[c] for c in prompt], dtype=torch.long).unsqueeze(0)  # (1, T)

    with torch.no_grad():
        for _ in range(max_new_tokens):

            # 截取最近 block_size 个 token 作为上下文窗口
            idx_cond = idx[:, -block_size:]  # (1, T)

            # 前向传播得到 logits
            logits = model(idx_cond)  # (1, T, V)

            # 仅取最后一个时间步的预测分布
            logits = logits[:, -1, :]  # (1, V)

            # greedy decoding: 选择概率最大的 token
            probs = torch.softmax(logits, dim=-1)
            next_idx = torch.argmax(probs, dim=-1, keepdim=True)  # (1, 1)

            # 如果生成 EOS token, 则提前停止
            if next_idx.item() == char2idx[eos_token]:
                break

            # 拼接到输入序列
            idx = torch.cat([idx, next_idx], dim=1)

    # 解码为文本
    out = "".join([idx2char[i] for i in idx[0].tolist()])
    return out
```

代码 9.6: Decoder-only Transformer 的文本生成

练习

1) 分析 Decoder-only Transformer 的参数量构成，并推导其随模型维度（d_{model}）、层数（N）以及前馈隐藏维度（d_{ff}）变化的规模关系。

2) 在 Kaggle 或本地环境中实现一个简化版 Decoder-only Transformer（可基于字符级或子词级 tokenizer），完成训练与文本生成任务，并分析不同超参数（如层数、上下文长度、embedding 维度）对训练效率和生成效果的影响。