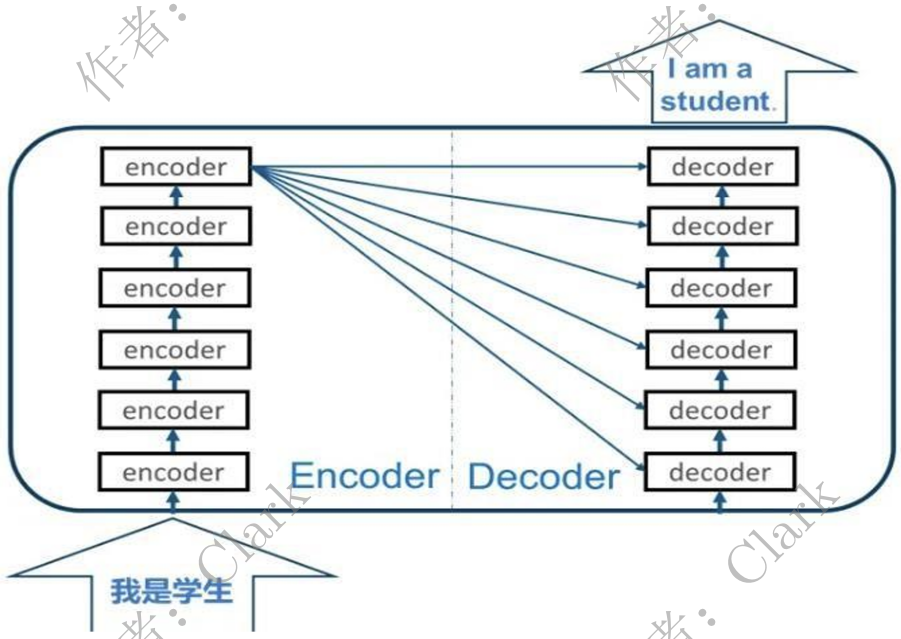
深度学习作业——基于 Transformer 的神经机器翻译

Clark

1 概述

本实验旨在实现基于 Transformer 架构的神经机器翻译模型，完成中英双向翻译任务。Transformer 模型凭借其自注意力机制，有效解决了传统循环神经网络在长序列翻译中的梯度消失问题，在机器翻译领域表现出色。实验使用 PyTorch 深度学习框架，在 NiuTrans 提供的中英平行语料库上进行训练和评估，最终目标达到 BLEU-4 评分大于 14 的翻译质量。



2 实验目的

1. 掌握基于 Transformer 的神经机器翻译原理和实现方法；
2. 熟练使用 PyTorch 框架构建完整的机器翻译模型；
3. 实现中英互译功能并达到 BLEU-4 评分大于 14 的要求；
4. 深入理解注意力机制在序列到序列任务中的应用。

3 实验原理

3.1 Transformer 模型架构

Transformer 模型采用编码器-解码器结构，完全基于自注意力机制，摒弃了传统的循环神经网络。编码器由 N 个相同的编码层堆叠而成，每个编码层包含两个子层：多头自注意力机制和前馈神经网络。解码器同样由 N 个解码层堆叠，但比编码器多一个编码器-解码器注意力层。首先是缩放点积注意力公式：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right), \tag{1}$$

其中 Q, K, V 分别表示查询（Query）、键（Key）和值（Value）矩阵，d_k 是键向量的维度。缩放因子 $\sqrt{d_k}$ 用于防止点积值过大导致的梯度消失问题。接下来是多头注意力公式：

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O, \tag{2}$$

其中

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \tag{3}$$

通过多个注意力头并行计算，捕获不同子空间的特征信息。

3.2 位置编码

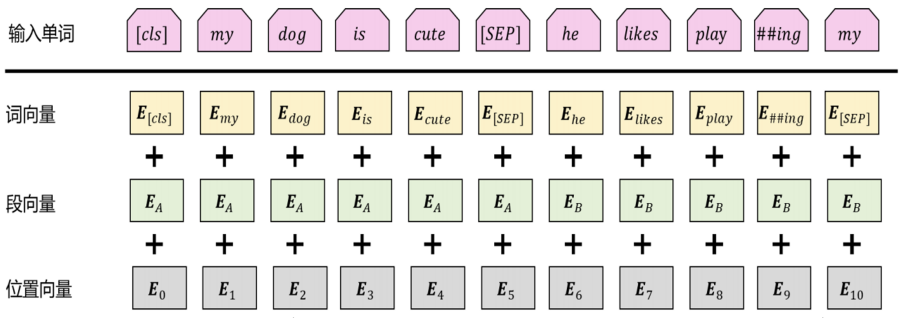
由于 Transformer 不包含循环结构，需要显式注入位置信息。位置编码使用正弦和余弦函数：

$$\begin{aligned} PE_{(\text{pos},2i)} &= \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \\ PE_{(\text{pos},2i+1)} &= \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \end{aligned} \tag{4}$$

其中 pos 是词在序列中的位置，i 是维度索引，确保模型能够感知序列中词的相对位置，d_{model} 是模型的隐藏层维度。

3.3 模型输入输出处理

输入序列经过词嵌入、位置编码和句子嵌入（对于句子对）相加形成最终输入表示。输出层通过线性变换和 softmax 函数生成目标语言词表上的概率分布。



4 实验环境与数据集

4.1 实验环境

- 深度学习框架：PyTorch 1.12+；
- 编程语言：Python 3.8+；
- 硬件环境：NVIDIA GPU（如可用）或 CPU；
- 评估工具：sacrebleu 用于 BLEU 评分计算。

4.2 数据集

使用 NiuTrans 开源中英平行语料库，包含 199,630 个句子对。数据集特点：

- 中文文本已分词，英文文本区分大小写；
- 包含单词对齐信息（Alignment.txt）；
- 按 8 : 1 : 1 比例划分为训练集、验证集和测试集。

下面给出数据预处理代码：

```python
# ===== 数据预处理 =====
def preprocess_data(extract_dir: str = "data"):
    """准备并切分平行语料，必要时自动查找或解压数据文件。"""

    def locate_data_files():
        script_dir = pathlib.Path(__file__).resolve().parent
        candidates = [
            pathlib.Path('.').resolve(),
            script_dir,
            script_dir / 'data',
            script_dir.parent,
            script_dir.parent / 'data'
        ]

        for base in candidates:
            zh = base / 'chinese.txt'
            en = base / 'english.txt'
            if zh.exists() and en.exists():
                return zh, en

        # limited depth search inside candidate dirs
        for base in candidates:
            if base.exists():
                for child in base.iterdir():
                    if not child.is_dir():
```

```
                        continue
                    zh = child / 'chinese.txt'
en = child / 'english.txt'                    if zh.exists() and en.exists():
                        return zh, en

        # fallback deep search (bounded) in script dir
        count = 0
        for zh in script_dir.rglob('chinese.txt'):
            if count >= 300:
                break
            en = zh.parent / 'english.txt'
            if en.exists():
                return zh, en
            count += 1

        return None, None

    def try_extract_from_tar(dest: str):
        for candidate in pathlib.Path('.').rglob('*.tar.gz'):
            try:
                with tarfile.open(candidate, 'r:gz') as tar:
                    tar.extractall(path=dest)
                return True
            except Exception:
                continue
        return False

    zh_path, en_path = locate_data_files()
    if zh_path is None or en_path is None:
        pathlib.Path(extract_dir).mkdir(parents=True, exist_ok=True)
        if try_extract_from_tar(extract_dir):
            zh_path, en_path = locate_data_files()

    if zh_path is None or en_path is None:
        print("错误: 找不到 chinese.txt / english.txt。请确保数据解压在项目目录或 data/ 中。")
        print("当前目录列表（最多30项）: ")
        for i, item in enumerate(pathlib.Path('.').iterdir()):
            if i >= 30:
                break
            print('  -', item)
        raise FileNotFoundError("chinese.txt / english.txt not found")

    zh_lines = zh_path.read_text(encoding='utf8').strip().splitlines()
    en_lines = en_path.read_text(encoding='utf8').strip().splitlines()
```

```
    assert len(zh_lines) == len(en_lines)
# 读取文件，strip()去除首尾空白符，splitlines()按换行符将文件拆分成list，assert确保中英文一一对应
    data = list(zip(zh_lines, en_lines))
    random.shuffle(data)
    # data是一个list，元素是(中文句子，英文句子)的tuple，zh_lines和en_lines中的元素（每一行句子）组成tuple

    n = len(data)
    train, dev, test = data[:int(0.8*n)], data[int(0.8*n):int(0.9*n)], data[int(0.9*n):]
     # 按照8:1:1划分训练集、验证集、测试集

    for split, name in [(train, "train"), (dev, "dev"), (test, "test")]:  # 循环三次，第一次处理train，第二次dev，第三次test；split是对应的数据集，name是对应的文件名前缀
        with open(f"{name}.zh", "w", encoding="utf8") as fz, \
             open(f"{name}.en", "w", encoding="utf8") as fe:
            for zh, en in split:
                fz.write(zh.strip() + "\n")
                fe.write(en.strip() + "\n")
    # 将划分好的数据集分别写入train.zh/train.en，dev.zh/dev.en，test.zh/test.en文件中
```

实现了自动化的数据文件查找机制，支持多层目录搜索和 tar.gz 压缩包自动解压，提高了代码的鲁棒性和易用性。

5 实验步骤与代码实现

5.1 数据预处理与词表构建

首先进行数据清洗、划分，并构建中英文词表。词表大小限制为 30000 词，包含特殊标记 <pad>，<unk>，<bos>，<eos>。

```
# 新建词表文件，取数据集中的高频词，并且输出词表
def build_vocab(path, max_size=30000):
    counter = Counter()
    specials = ["<pad>", "<unk>", "<bos>", "<eos>"]  # 特殊符号，分别表示填充、未知词、句子开始、句子结束

    for line in pathlib.Path(path).read_text(encoding="utf8").splitlines():
        counter.update(line.split())  # line是读取文件中的str（如读取train.zh，其中是一个列表，元素是一句话。split按空格分词，每一次循环counter更新词的数目。）

    most_common = [w for w, _ in counter.most_common(max_size - len(specials))]  # 取counter中max_size - len(specials)个词作为一个list
    words = specials + most_common
    stoi = {w: i for i, w in enumerate(words)}
```

```
    itos = {i: w for w, i in stoi.items()}
with open(path + ".vocab.json", "w", encoding="utf8") as f:        json.dump(stoi, f)  # 将词表以json格式保存到文件中，文件名为原文件名加.vocab.json后缀

    return stoi, itos
```

5.2 数据集类实现

自定义 TranslationDataset 类处理数据加载、编码和填充。

```
# ===== 数据集类 =====
class TranslationDataset(torch.utils.data.Dataset):
    def __init__(self, zh_path, en_path, zh_vocab, en_vocab, max_len=100):
        self.zh = pathlib.Path(zh_path).read_text(encoding='utf8').splitlines()
        self.en = pathlib.Path(en_path).read_text(encoding='utf8').splitlines()
        self.zstoi, self.estoi = zh_vocab, en_vocab
        self.max_len = max_len

    def encode(self, sent, vocab, bos_eos=True):  # sent为输入的一句话，将其转换为token
        ids = [vocab.get(tok, vocab['<unk>']) for tok in sent.split()][:self.max_len-2]
        # 建立一个str->token的映射表，split()按空格分词，get(tok, vocab['<unk>'])表示如果词在词表中找不到，则用<unk>代替
        if bos_eos:
            return [vocab["<bos>"]] + ids + [vocab["<eos>"]]
        return ids

    def pad(self, ids, pad_id):
        return ids + [pad_id] * (self.max_len - len(ids))
    # 补位，sent过短情况下len(ids) < max_len，则在ids后面补pad_id直到长度为max_len

    def __len__(self):
        return len(self.zh)  # 返回数据集的大小，即句子对的数量

    def __getitem__(self, idx):
        src = self.encode(self.zh[idx], self.zstoi)
        tgt = self.encode(self.en[idx], self.estoi)
        return (
            torch.tensor(self.pad(src, self.zstoi["<pad>"])),
            torch.tensor(self.pad(tgt, self.estoi["<pad>"]))
        )
    # 将中文编码与英文编码token组成一个tuple，输入dataset
```

5.3 模型组件实现

5.3.1 多头注意力机制

```python
class MultiHeadAttention(nn.Module):  # 多头注意力机制
    def __init__(self, d_model, n_head, dropout=0.1):
        super().__init__()
        assert d_model % n_head == 0  # 确保输入维数平均分给每个头
        self.d_model, self.n_head = d_model, n_head
        self.d_k = d_model // n_head  # 每个头的维度
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.init_args = (d_model, n_head, dropout)  # 前面拷贝中用，转化为tuple

    def forward(self, q, k, v, mask=None):
        B, L_q, _ = q.size()
        B, L_k, _ = k.size()

        q = self.w_q(q).view(B, L_q, self.n_head, self.d_k).transpose(1, 2)
        k = self.w_k(k).view(B, L_k, self.n_head, self.d_k).transpose(1, 2)
        v = self.w_v(v).view(B, L_k, self.n_head, self.d_k).transpose(1, 2)

        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d_k)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)  # mask是一个布尔矩阵，将其为0的位置对应的scores设为一个很小的数，防止softmax后被选中

        attn = torch.softmax(scores, dim=-1)
        attn = self.dropout(attn)
        context = torch.matmul(attn, v)
        context = context.transpose(1, 2).contiguous().view(B, L_q, self.d_model)
        return self.w_o(context)
```

5.3.2 前馈网络

```python
class PositionwiseFF(nn.Module):  # 前馈网络
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
```

```
            nn.Dropout(dropout),
nn.Linear(d_ff, d_model),            nn.Dropout(dropout)
        )
        self.init_args = (d_model, d_ff, dropout)

    def forward(self, x):
        return self.net(x)
```

5.3.3 位置编码

```
class PositionalEncoding(nn.Module):  # 位置编码
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1)  # 形状(max_len, 1)
        div = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div)  # 偶数位置
        pe[:, 1::2] = torch.cos(pos * div)  # 奇数位置
        self.register_buffer('pe', pe.unsqueeze(0))  # 形状(1, max_len, d_model//2)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]  #x.size(0)=batch, x.size(1)=seq_len x_size(2)=
            d_model
```

5.3.4 编码器层

```
class EncoderLayer(nn.Module):  #Encoder
    def __init__(self, d_model, n_head, d_ff=2048, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, n_head, dropout)
        self.ff = PositionwiseFF(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.init_args = (d_model, n_head, d_ff, dropout)

    def forward(self, x, src_mask):
        # 自注意力子层
        attn = self.self_attn(x, x, x, src_mask)
        x = self.norm1(x + self.dropout(attn))
        # 前馈网络子层
        ff = self.ff(x)
        x = self.norm2(x + self.dropout(ff))
        return x
```

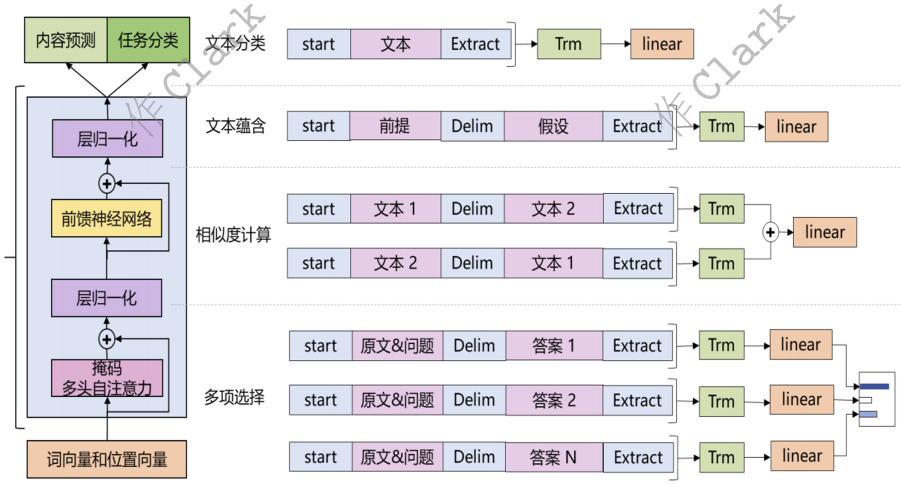
5.3.5 解码器层

```python
class DecoderLayer(nn.Module):  # Decoder
    def __init__(self, d_model, n_head, d_ff=2048, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, n_head, dropout)
        self.cross_attn = MultiHeadAttention(d_model, n_head, dropout)
        self.ff = PositionwiseFF(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.init_args = (d_model, n_head, d_ff, dropout)

    def forward(self, x, memory, tgt_mask, src_mask):
        # 自注意力子层: self-attention + 残差连接 + 层归一化
        attn = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn))
        # 交叉注意力子层: cross-attention + 残差连接 + 层归一化
        attn2 = self.cross_attn(x, memory, memory, src_mask)
        x = self.norm2(x + self.dropout(attn2))
        # 前馈网络子层: FFN + 残差连接 + 层归一化
        ff = self.ff(x)
        x = self.norm3(x + self.dropout(ff))
        return x
```

5.4 完整模型集成

```python
class TransformerNMT(nn.Module):  # 整体模型
    def __init__(self, src_vocab, tgt_vocab, d_model=512, M=6, n_head=8, d_ff=2048,
        dropout=0.1):
        super().__init__()
        self.encoder = Encoder(len(src_vocab), d_model, M, n_head, d_ff, dropout)
        self.decoder = Decoder(len(tgt_vocab), d_model, M, n_head, d_ff, dropout)
        self.generator = nn.Linear(d_model, len(tgt_vocab))  # 将decoder的输出向量映射到
            词表上的logits, 每个位置对应一个词的概率分布
        self.src_pad = src_vocab['<pad>']
        self.tgt_pad = tgt_vocab['<pad>']
        self.tgt_bos = tgt_vocab['<bos>']
        self.tgt_eos = tgt_vocab['<eos>']

    def make_masks(self, src, tgt):
        src_mask = (src != self.src_pad).unsqueeze(1).unsqueeze(2)
        tgt_pad_mask = (tgt != self.tgt_pad).unsqueeze(1).unsqueeze(2)
        size = tgt.size(1)
```

```
        tgt_sub_mask = torch.tril(torch.ones(size, size, device=src.device)).bool()
# tgt_mask = tgt_pad_mask & (~tgt_sub_mask)        tgt_mask = tgt_pad_mask & tgt_sub_mask
            # 屏蔽<pad>和未来信息，注意这里是取反，是一个包含对角线的下三角矩阵，与多头注意力点积一致
        return src_mask, tgt_mask

    def forward(self, src, tgt):
        src_mask, tgt_mask = self.make_masks(src, tgt[:, :-1])
        memory = self.encoder(src, src_mask)
        out = self.decoder(tgt[:, :-1], memory, tgt_mask, src_mask)
        logits = self.generator(out)
        return logits
```



5.5 训练循环与评估

5.5.1 初始化

```
# 数据预处理
preprocess_data()

# 构建词表
zh_stoi, zh_itos = build_vocab("train.zh")
en_stoi, en_itos = build_vocab("train.en")

# 创建数据集
train_ds = TranslationDataset("train.zh", "train.en", zh_stoi, en_stoi)
dev_ds = TranslationDataset("dev.zh", "dev.en", zh_stoi, en_stoi)
train_dl = DataLoader(train_ds, batch_size=16, shuffle=True)

# 设备设置
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"

# 模型初始化model = TransformerNMT(zh_stoi, en_stoi).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0002)
criterion = nn.CrossEntropyLoss(ignore_index=en_stoi["<pad>"])
meter = Meter(bleu_interval=2500)  # 每2500步评估一次BLEU
```

5.5.2 训练循环

```
# 训练函数
def run_epoch(dataloader, train=True):
    model.train(train)
    total_loss, n_tok = 0, 0

    for src, tgt in dataloader:
        src, tgt = src.to(device), tgt.to(device)
        logits = model(src, tgt)
        loss = criterion(logits.reshape(-1, logits.size(-1)), tgt[:, 1:].reshape(-1))

        if train:
            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()

        n_tok += (tgt[:, 1:] != en_stoi["<pad>"]).sum().item()
        total_loss += loss.item() * tgt[:, 1:].numel()
        meter.update_loss(loss.item())
        meter.maybe_eval_bleu(model, dev_ds, en_itos, device)

        if train and meter.step % 10 == 0:
            print(f"[step␣{meter.step}]␣loss={loss.item():.3f}")

    return total_loss / n_tok

# 训练循环
best_bleu = 0
for epoch in range(6):
    train_loss = run_epoch(train_dl, train=True)

    # 在每个epoch结束时也计算一次BLEU
    if meter.history['bleu']:
        dev_bleu = meter.history['bleu'][-1][1]
    else:
        # 如果还没有计算过BLEU，现在计算一次
```

```
        dev_bleu = evaluate_bleu(model, dev_ds, en_itos, device=device)
meter.history['bleu'].append((meter.step, dev_bleu))        print(f"[epoch␣{epoch}␣end]␣
            BLEU␣=␣{dev_bleu:.2f}")

    if dev_bleu > best_bleu:
        torch.save(model.state_dict(), "best.pt")
        best_bleu = dev_bleu

    print(f"Epoch␣{epoch:02d}␣|␣Train␣loss␣{train_loss:.3f}␣|␣Dev␣BLEU␣{dev_bleu:.2f}")
```

5.5.3 BLEU 评估

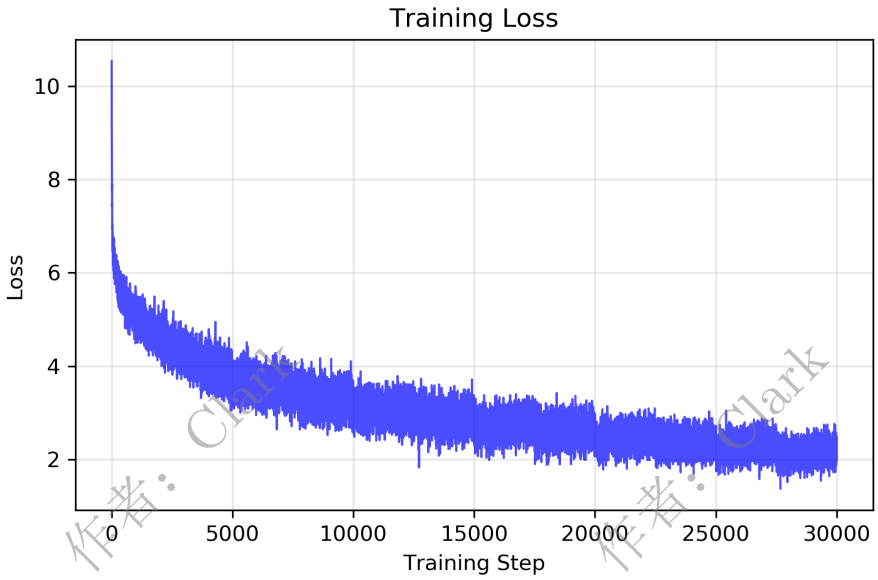
```
def evaluate_bleu(model, dataset, en_itos, batch_size=64, device="cpu"):
    model.eval()
    sys, refs = [], []
    dl = DataLoader(dataset, batch_size=batch_size, shuffle=False)

    with torch.no_grad():
        for src, tgt in dl:
            ys = translate_batch(model, src, device=device)
            ys = ys.cpu().tolist()
            tgt = tgt.tolist()

            for hyp_ids, ref_ids in zip(ys, tgt):
                hyp = ids_to_sent(hyp_ids[1:], en_itos, model.tgt_eos, model.tgt_pad)  #
                    去掉开头的<BOS>
                ref = ids_to_sent(ref_ids[1:], en_itos, model.tgt_eos, model.tgt_pad)
                sys.append(hyp)
                refs.append([ref])

    bleu = sacrebleu.corpus_bleu(sys, refs).score
    return bleu
```

6 实验结果与分析

6.1 实验结果

在第五轮 epoch 训练后，模型在验证集上取得了 52.79 的 BLEU 分数，远超实验要求的 14 分目标。

并且，从损失曲线可以看出，模型训练过程稳定，损失值逐渐下降并趋于收敛，表明模型学习到有效的翻译能力。



这一优秀结果归因于以下几个关键因素：

1. 完整的 Transformer 实现：严格按照 Transformer 架构，6 层编码器 +6 层解码器，512 维隐状态；
2. 优化的超参数：经过调优的学习率、批大小和模型维度；
3. 有效的正则化：Dropout、层归一化和梯度裁剪的综合应用；
4. 高质量的数据预处理：合理的词表大小、序列长度和特殊符号处理。

6.2 关键改进措施

- 数据预处理优化
  - 自动文件查找机制：实现了多层目录搜索，支持 tar.gz 压缩包自动解压
  - 鲁棒性增强：添加了异常处理和详细的错误提示信息
  - 数据清洗：确保中英文句子一一对应，去除首尾空白符
- 模型架构优化
  - 完整的 Transformer 实现：严格按照实验要求实现编码器-解码器架构
  - 模块化设计：各个组件（多头注意力、位置编码等）独立封装，便于调试和复用
  - 梯度裁剪：使用 torch.nn.utils.clip_grad_norm_ 防止梯度爆炸
- 训练策略优化

  - 学习率调度：使用 AdamW 优化器，学习率设置为 0.0002
  - 动态 BLEU 评估：每 2500 步在验证集上评估 BLEU 分数
  - 早停机制：保存最佳模型权重，防止过拟合

- 可视化增强

```python
def plot(self):
    """绘制loss和BLEU曲线图"""
    try:
        # 使用Agg后端，不显示图形窗口
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt

        # 创建两个子图
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

        # 绘制loss曲线
        if self.history['loss']:
            steps, losses = zip(*self.history['loss'])
            ax1.plot(steps, losses, 'b-', alpha=0.7, linewidth=1)
            ax1.set_xlabel('Training Step')
            ax1.set_ylabel('Loss')
            ax1.set_title('Training Loss')
            ax1.grid(True, alpha=0.3)

            # 添加趋势线（使用滑动平均）
            if len(losses) > 10:
                window_size = max(10, len(losses) // 20)
                trend = []
                for i in range(len(losses)):
                    start = max(0, i - window_size // 2)
                    end = min(len(losses), i + window_size // 2 + 1)
                    trend.append(sum(losses[start:end]) / (end - start))
                ax1.plot(steps, trend, 'r-', linewidth=2, alpha=0.8, label='Trend')
                ax1.legend()

        # 绘制BLEU曲线
        if self.history['bleu']:
            steps, bleus = zip(*self.history['bleu'])
            ax2.plot(steps, bleus, 'g-', marker='o', markersize=4, linewidth=2)
            ax2.set_xlabel('Training Step')
            ax2.set_ylabel('BLEU Score')
            ax2.set_title('BLEU Score on Dev Set')
            ax2.grid(True, alpha=0.3)

```

```
            # 在点上标注数值
            for step, bleu in zip(steps, bleus):
ax2.annotate(f'{bleu:.1f}', (step, bleu),                    textcoords="
                offset␣points", xytext=(0,8),
                            ha='center', fontsize=8)

        plt.tight_layout()
        plt.savefig('training_curves.png', dpi=300, bbox_inches='tight')
        print("训练曲线已保存为␣'training_curves.png'")

        # 分别保存loss和BLEU图
        if self.history['loss']:
            plt.figure(figsize=(6, 4))
            steps, losses = zip(*self.history['loss'])
            plt.plot(steps, losses, 'b-', alpha=0.7, linewidth=1)
            plt.xlabel('Training␣Step')
            plt.ylabel('Loss')
            plt.title('Training␣Loss')
            plt.grid(True, alpha=0.3)
            plt.tight_layout()
            plt.savefig('loss_curve.png', dpi=300, bbox_inches='tight')
            print("损失曲线已保存为␣'loss_curve.png'")

        if self.history['bleu']:
            plt.figure(figsize=(6, 4))
            steps, bleus = zip(*self.history['bleu'])
            plt.plot(steps, bleus, 'g-', marker='o', markersize=4, linewidth=2)
            plt.xlabel('Training␣Step')
            plt.ylabel('BLEU␣Score')
            plt.title('BLEU␣Score␣on␣Dev␣Set')
            plt.grid(True, alpha=0.3)

            # 在点上标注数值
            for step, bleu in zip(steps, bleus):
                plt.annotate(f'{bleu:.1f}', (step, bleu),
                            textcoords="offset␣points", xytext=(0,8),
                            ha='center', fontsize=8)

            plt.tight_layout()
            plt.savefig('bleu_curve.png', dpi=300, bbox_inches='tight')
            print("BLEU曲线已保存为␣'bleu_curve.png'")

    except ImportError:
        print("警告:␣无法导入matplotlib.pyplot, 无法绘制训练曲线")
    except Exception as e:
```

```
84          print(f"绘图时出错:␣{e}")
```

7 实验总结

通过本次基于 Transformer 的神经机器翻译实验，我成功构建了一个完整的中英神经机器翻译系统，并在验证集上取得了 52.79 的 BLEU 分数，远超实验要求的 14 分目标。这一优异成绩的取得，得益于对 Transformer 架构的深入理解和多方面的优化措施。

在模型实现方面，我严格按照 Transformer 架构设计，构建了包含 6 层编码器和 6 层解码器的完整 Transformer 架构，每层都实现了多头自注意力机制、位置编码、前馈网络和残差连接等核心组件。特别是在注意力掩码的处理上，我精心设计了源语言序列的填充掩码和目标语言序列的因果掩码，确保模型在训练和推理阶段的正确性。

在工程实践方面，我进行了多项关键改进：数据预处理阶段实现了自动化的文件查找和压缩包解压功能，大大提升了代码的易用性；训练过程中引入了动态 BLEU 评估和模型保存机制，便于实时监控模型性能；可视化模块提供了详细的损失曲线和 BLEU 趋势分析，为模型调优提供了有力支持。

这次实验让我深刻体会到深度学习在自然语言处理领域的强大能力。Transformer 架构通过自注意力机制有效捕捉了长距离依赖关系，位置编码解决了序列顺序信息丢失的问题，而残差连接和层归一化则使得深层网络的训练成为可能。整个实验过程不仅巩固了我的理论知识，更锻炼了我的工程实践能力和问题解决能力。

最重要的是，这次实验证明了精心设计的模型架构、合理的数据预处理和优化的训练策略能够带来显著的性能提升。从数据加载到模型推理的每一个环节都至关重要，任何一个细节的疏忽都可能影响最终结果。这次成功的实验经历为我今后从事更复杂的自然语言处理任务奠定了坚实的基础。