深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇 二四 ～二〇 二五 学年度第 二 学期

课程编号 课序号 课程名称 主讲教师 评分

学 号 姓名 专业年级

教师评语:

题目: 基于 Transformer 的中译英机器翻译模型

摘要

本文聚焦基于 Transformer 的中译英机器翻译模型研究，旨在通过构建高效的神经机器翻译系统实现中文到英文的语义转换。

首先，模型采用 Transformer 架构，其核心基于自注意力机制，通过编码器与解码器的多层堆叠、多头注意力机制及位置编码技术，实现对序列信息的并行处理与语义建模。在数据集处理阶段，选用 2013 年 TED 演讲中英文双语语料（OPUS 语料库），通过 SentencePiece 工具采用 BPE 算法进行子词级编码，将中英文词表大小设为 16000，并通过填充操作统一序列长度为 60，有效处理未登录词问题并提升计算效率。

模型搭建基于 PyTorch 框架，词嵌入维度设为 512，结合正弦 - 余弦位置编码与双重掩码机制（padding mask 和 causal mask），主体包含 6 层编码器与解码器，每层 8 头注意力，前馈网络维度 2048。训练过程采用标签平滑损失函数（ε=0.1）与 Noam 学习率调度策略（warmup=4000），基于 Adam 优化器迭代 50 次，总参数约 68.7M。

解码阶段对比贪婪解码、束搜索（束宽=5）和 Top-k 采样（k=5），结果表明束搜索以平均 BLEU 分数 0.2050 显著优于其他策略；接着通过参数分析对比，最终确定束搜索（束宽=8）作为最优解码方式。翻译实验显示，模型对简单句（如日常对话）翻译准确率高，但在长句及复杂语义（如学术表达、上下文依赖场景）处理中存在语义漂移与语法错误，后续可通过参数调优与更大规模语料训练进一步优化。

模型简介

Transformer 是 Vaswani 等人于 2017 年提出的一种基于注意力机制的序列建模结构，在机器翻译、文本生成等自然语言处理任务中取得了突破性成果。与传统基于循环神经网络（RNN）或卷积神经网络（CNN）的模型不同，Transformer 完全基于自注意力机制（Self-Attention），能够并行处理整个输入序列，显著提高训练效率与建模能力。

Transformer 模型由编码器（Encoder）和解码器（Decoder）两部分构成，每部分均由多个结构相同的子层堆叠而成。编码器每一层包含一个多头自注意力机制（Multi-head Self-Attention）和一个前馈神经网络（Position-wise Feed-Forward Network）；解码器则在此基础上额外引入了用于与编码器交互的编码器-解码器注意力机制（Encoder-Decoder Attention），使解码器在生成目标序列时能够动态关注源语言信息。

为了保留序列中词语的位置信息，Transformer 引入了位置编码（Positional Encoding），将词向量与固定或可学习的位置信息相加，从而弥补模型缺乏序列顺序感知的缺陷。

多头注意力机制则通过在不同子空间中并行计算注意力表示，增强了模型的表达能力。此外，残差连接（Residual Connection）与层归一化（Layer Normalization）技术在每个子层中应用，提升了训练的稳定性和深层结构的可训练性。

总的来说，Transformer 模型以其结构简洁、并行高效、建模能力强的优势，已成为当前主流的神经机器翻译架构，也为后续如 BERT、GPT 等预训练模型提供了重要基础。

流程图如下（摘选自 *Attention Is All You Need*）：

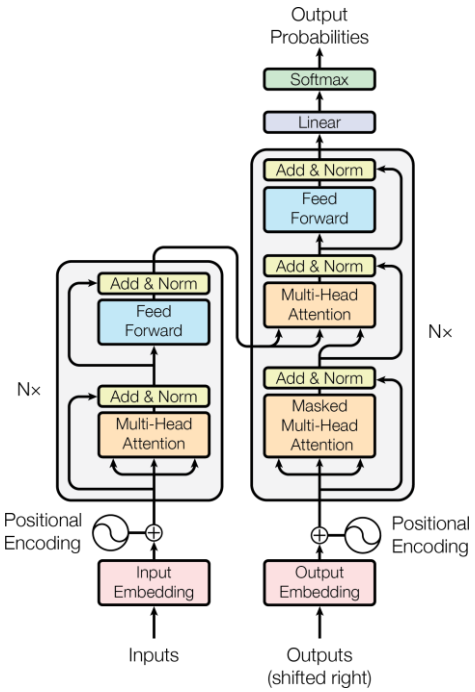


图 1

数据集处理

本文选取的数据集是 2013 年的 TED 演讲（中文是英文字幕的翻译），来源于 OPUS 语料库（网站：https://opus.nlpl.eu/）。数据集大小共 30MB，有 309156 个句子，中文英文各占一半。

数据集中有包含网站和一些乱码与怪异符号，这些均视为无效信息，均被剔除。

词元化（Tokenize）

为了提升模型对未登录词（OOV）的处理能力并减少词表规模，本文采用 Google 提出的开源工具 SentencePiece 对中英文语料进行子词级编码（Subword Tokenization）。

SentencePiece 是一种无需预分词（whitespace-free）的训练式分词器，能够直接从原始文本中学习高频子词单元，从而构建语言无关的、紧凑而高效的词汇系统。

本文采用 BPE（Byte Pair Encoding，算法原理见附录） 作为分词算法，在中英文语料上分别训练了两个独立的分词模型。模型训练过程设定词表大小为 16000，其中中文语料设置字符覆盖率为 0.9995（减少生僻字对模型训练的干扰），英文设置为 1.0，以充分保留常用字符信息。训练完成后，通过 SentencePieceProcessor 加载模型，实现源句与目标句的分词（编码）和去分词（解码）操作。该方法有效缓解了传统词级建模面临的 OOV 问题，同时提升了模型在低频词处理和泛化方面的表现。

经初步处理，token 数据如表 1 所示（UNK 表示未知 token）：无论中英文，未知 token 数量占比均非常少，近乎可以忽略不计，这也说明处理后的数据集比较干净。此外，中英文 token 序列长度分布如图 2 所示：无论中英文，绝大部分的 token 序列长度都是低于 60 的；因此出于对 GPU 内存与计算效率的考虑，将数据集中的每个 token 序列长度固定为 60（包括句子起始符和句子结束符），同时这也几乎不会对语义完整性造成影响。

语言类别	总 token 数	UNK 数	UNK 比例
中文	2,651,882	2,282	0.09%
英文	6,020,150	2,282	0.04%

表 1

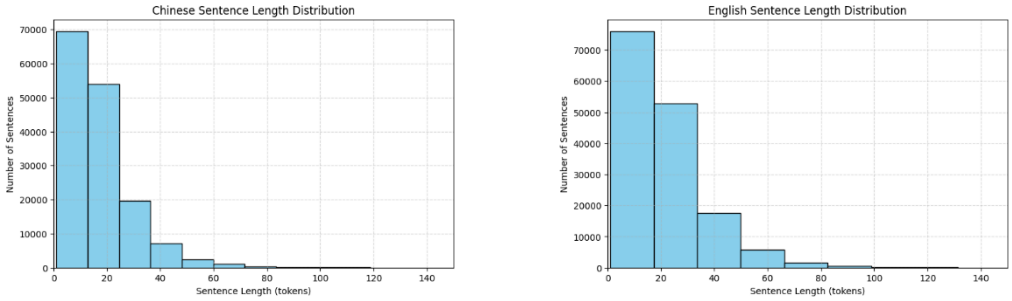


图 2

值得注意的是，为了统一输入序列长度，本文采用特殊的填充符 <PAD> 将所有源句与目标句补齐至最大长度 60。这种填充 token 被命名为 padding token。该 token 的索引设为 0，并在嵌入层设置 padding_idx=0，以避免在训练中对其进行梯度更新。同时，在注意力计算中使用 padding mask 屏蔽填充位置，防止模型关注无效信息。

此外，为了尽可能的有效训练模型和避免数据顺序偏见，数据集被随机划分为 90%的训练集和 10%测试集；并且，为了充分发挥 GPU 的计算性能和减少训练时间，本文将单次输入样本量（batch_size）设置为 256。

Transformer 模型搭建

本文基于 PyTorch 框架自定义构建了一个标准的 Transformer 编码器-解码器结构，整体架构由词嵌入层、位置编码模块、Transformer 主体（由多个注意力层堆叠组成）、输出投影层等模块组成。Transformer 主体代码见附录，以下简单阐述该模型的搭建：

词嵌入层（Embedding）

输入的源语言与目标语言序列首先通过词嵌入层映射为稠密向量表示，嵌入维度为d_model=512。为了保证模型训练的数值稳定性，嵌入向量在输入 Transformer 之前乘以一个缩放系数$\sqrt{d_{model}}$。

位置编码（Positional Encoding）

前文中我们已经做好了 tokenize 和编码处理，接下就要创建位置编码矩阵。位置编码矩阵是由 max_len(允许的位置编码的最大长度)个 token 向量组成的，行数等于 token 的维度，列数等于 max_len。由于前文限制的 token 序列长度是 60，而且 d_model 这里设置为 512，那么位置编码矩阵维度等于 512×60。（更严谨的说，维度是 1×512×60，其中 1 表示 batch 维度，用于 broadcast，即所有样本共享相同的位置编码）

编码方式采用正弦-余弦位置编码，公式如下：

$$PE_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

其中，PE 表示位置编码，pos 表示单词在句子中的绝对位置（即索引）。

掩码机制（Masking）

在 Transformer 的计算过程中，为了确保模型的注意力不会关注到 padding token，本文引入了 padding mask。此外，为了实现目标端的自回归解码，使用了上三角 causal mask(因果掩码)，防止模型在生成第 t 个词时看到第 $t+1$ 个词。

Transformer 主体结构

模型主体由 PyTorch 提供的 nn.Transformer 模块构成，包含 6 层编码器（encoder layers）和 6 层解码器（decoder layers），每层使用 8 个多头注意力头（multi-head attention），前馈神经网络隐藏层维度为 2048，dropout 概率为 0.1。

输出层（Linear Projection）

解码器输出为每个位置上的上下文表示，接下来通过一个线性层投影为目标词表大小的 logits，作为后续计算交叉熵损失的输入。

模型训练与评估

损失函数定义

为了提升模型的泛化能力并缓解过拟合问题，本文在训练过程中引入了标签平滑（Label Smoothing）的损失函数。与传统的负对数似然损失（NLL Loss）不同，标签平滑在目标分布中为非目标类别分配少量概率质量，从而避免模型对训练集中的目标 token 产生过强的置信度。具体而言，目标分布不再是 one-hot 向量，而是将$(1-\varepsilon)$分配给正确类别， ε 平均分配给其余类别，从而形成更平滑的目标分布。

损失函数形式如下：

$$\mathcal{L} = (1-\varepsilon)\cdot \text{NLL} + \varepsilon \cdot \text{UniformLoss}$$

其中ε为平滑系数（本文设置$\varepsilon = 0.1$），NLL 是基于目标 token 的负对数概率损失，UniformLoss 表示模型对词表中所有 token 的预测分布与均匀分布之间的差异。

学习率调度器与优化器设置

为了提升训练的稳定性与模型的收敛效果，本文采用了 Noam 学习率调度策略（Noam Scheduler），其核心思想是在训练初期采用线性增加的学习率（warm-up），以避免模型在刚开始训练时参数更新过快导致不稳定；随后，学习率按训练步数的平方根反比递减，从而保证训练后期更加平稳并利于收敛。

本文$warmup$参数设置为 4000，因此初始学习率从 0.0001 开始线性增加至峰值 0.00035，然后再按训练步数的平方根反比递减。

此外，本文采用 Adam 优化器对模型参数进行优化。Adam 是结合了动量法和自适应学习率调整的优化算法，能够加速深度神经网络的收敛过程。

具体来说，Adam 通过一阶矩估计（均值）和二阶矩估计（未中心化的方差）自适应调整每个参数的学习率。本文设置超参数为 $\beta_1 = 0.9$ ， $\beta_2 = 0.98$ 。其中，β_1 控制一阶矩（梯度均值）的衰减率，β_2 控制二阶矩（梯度平方均值）的衰减率，$\varepsilon = 10^{-9}$ 用于防止除零错误，确保数值稳定性。再结合 Noam 学习率调度策略，优化器能够有效促进模型快速且稳定地收敛。

参数更新具体公式如下：

$$\theta_{t+1} = \theta_t - \eta_t \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \varepsilon}$$

其中，

$$\eta_t = d_{\text{model}}^{-0.5} \cdot \min(t^{-0.5}, t \cdot \text{warmup}^{-1.5})$$

模型评估

本文模型是在 colab 平台上搭建训练的，GPU 型号 A100，迭代次数 50，训练时长约 3 个小时。模型总参数为 68.7 M，具体如表 2 和表 3 所示：

模块名称	参数量估计（百万）
嵌入层（src + tgt）	16.38 M
编码器（6 层）	18.90 M
解码器（6 层）	25.20 M
输出层	8.19 M
总计	**68.7 M**

表 2

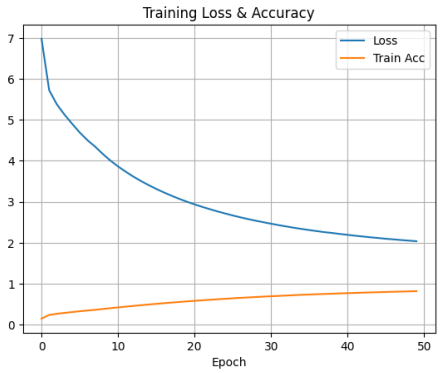
每层 Transformer 子层包含：

子层	参数结构	参数量
多头注意力 Q/K/V（8 头）	3 × (512 × 512)	786,432
输出投影（attention output）	512 × 512	262,144
LayerNorm ×2	2 × (512 × 2)	2,048
FFN：512 → 2048 → 512	512×2048 + 2048×512 = 2×1,048,576	2,097,152
FFN bias（两层）	2048 + 512	2,560

表 3

模型训练采用标签平滑的负对数似然损失函数，整体训练损失为所有 mini-batch 平均损失值之和。训练精度计算采用掩码准确率，即只统计目标序列中非 PAD token 的预测准确率，衡量模型在有效位置上的分类性能（测试损失和测试精度同理）。

训练损失，训练精度，测试损失，测试精度如图 3 和图 4 所示：



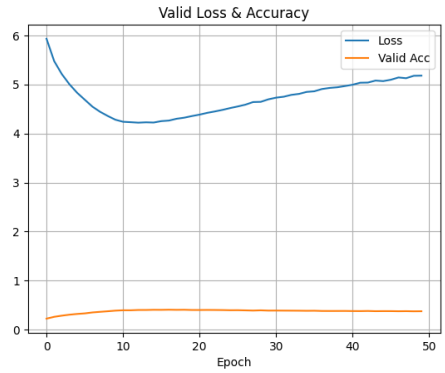


图 3

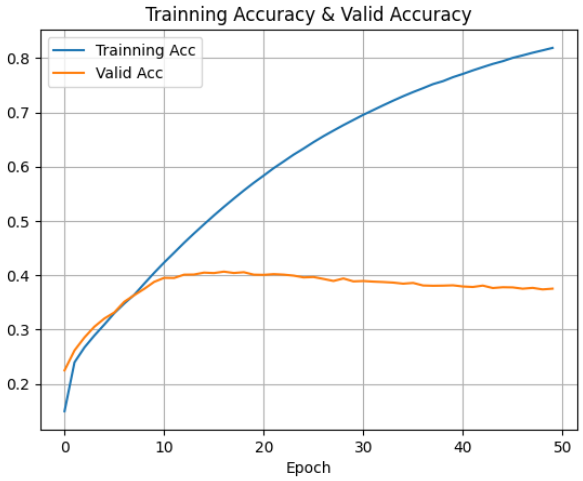


图 4

由上图可以发现，在这 50 次迭代中，训练损失逐渐下降，训练精度逐渐上升并最终达到 0.8 以上。但是测试损失和测试精度却呈现出“过拟合”状态（先增加后下降），测试精度在第 17 次迭代中达到最低值（约 0.4066），测试损失在第 13 次迭代中达到最低值（约 4.2202）。

由于算力资源有限，本文并没有进一步做模型参数调优。因此接下来，本文将着重探索模型在翻译上的应用。

翻译

解码简介

在神经机器翻译（Neural Machine Translation, NMT）任务中，目标序列的生成过程通常采用自回归方式：在给定源语编码表示及当前生成的目标前缀 $(y_1, y_2, \ldots, y_{t-1})$ 的条件下，模型预测第 t 个目标词 y_t 的概率分布

$$P(y_t \mid y_1, y_2, \ldots, y_{t-1}, x) = \text{Decoder}(y_{<t}, x)$$

然而，由于每步预测均涉及词表中的全部词项，穷尽搜索所有可能输出序列的组合是不现实的，因此必须设计高效的解码策略（Decoding Strategy），以在合理的计算成本下生成高质量译文。本文主要比较以下三种主流方法：

贪婪解码（Greedy Decoding）：

贪婪解码在每个时间步 t 都直接选择概率最大的词项：

$$y_t = \arg\max_{w \in \mathcal{V}} P(w \mid y_1, \ldots, y_{t-1}, x)$$

其整体生成序列为：

$$\hat{y} = \arg\max_{y_1} \ldots \arg\max_{y_T} P(y_t \mid y_{<t}, x)$$

尽管计算效率高，但贪婪解码属于局部最优策略，由于每步决策不可回溯，易导致生成序列偏离全局最优解。

束搜索（Beam Search）

为克服贪婪解码的局限，束搜索（Beam Search）在每一时间步保留概率最高的 k 个候选序列（称为束宽，beam width）。具体地，定义候选序列为 $y_{1:t}$ ，其得分为：

$$\text{Score}(y_{1:t}) = \log P(y_{1:t} \mid x) = \sum_{i=1}^{t} \log P(y_i \mid y_{<i}, x)$$

在每一步，beam search 按得分从高到低选取 k 个序列扩展，并在最终从所有完整序列中选出得分最高者作为最终输出：

$$\hat{y} = \arg\max_{y \in \text{Beam}_k} \log P(y \mid x)$$

束搜索在精度和速度之间提供良好平衡，是多数神经翻译系统的默认选择。

前-k采样（Top-k Sampling）

与上述确定性方法不同，Top-k 采样是一种随机化的生成策略，旨在提升翻译多样性。在每个解码步中，仅从概率最高的前 k 个词构成的集合 $\mathcal{V}_k \subset \mathcal{V}$ 中进行采样：

$$y_t \sim P(w \mid y_{<t}, x), \quad w \in \mathcal{V}_k$$

该方法通过限制采样空间来控制生成质量，同时保留一定的随机性，适合对话生成、摘要生成等任务。

解码分析

为了评估不同解码方式的效果，本文采用 BLEU 分数评估（最低分是 0，最高分是 1）。其公式如下：

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log P_n\right)$$

其中，BP是长度惩罚项，P_n 为第 n-gram 的精确率。

翻译测试样本为 translation_samples.txt，共 21 个句子，里面包含的句子类型有：基础简单句，一般陈述句，否定句，时态，疑问句，连词，复合句，长句（学术类），歧义（需要语境）。将束搜索的参数束宽（beam width）设置为 5，前 k 采样的参数 k 设置为 5；它们与贪婪解码的 BLEU 分析对比如下：

解码方式（Mode）	平均 BLEU 分数（Avg BLEU）
Greedy 解码	0.1630
Beam Search 束搜索	0.2050
Top-k 采样	0.0476

表 4

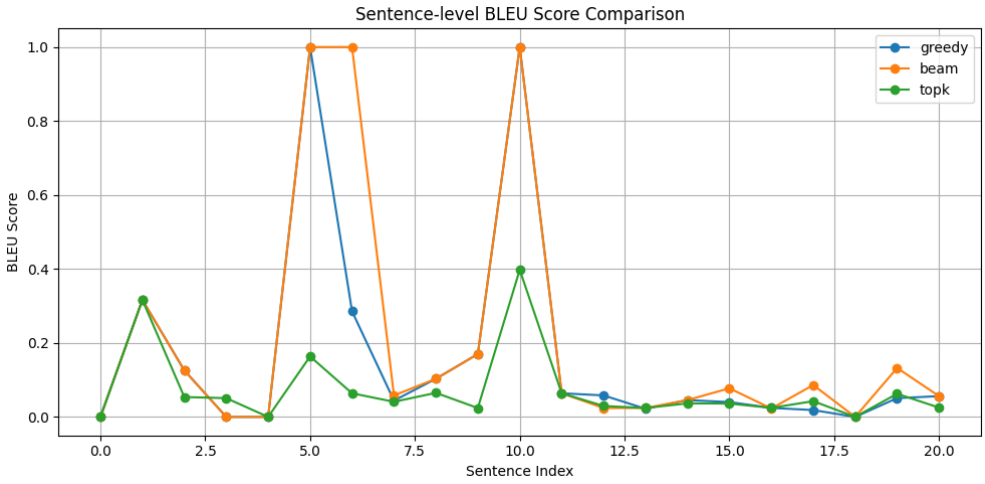


图 5

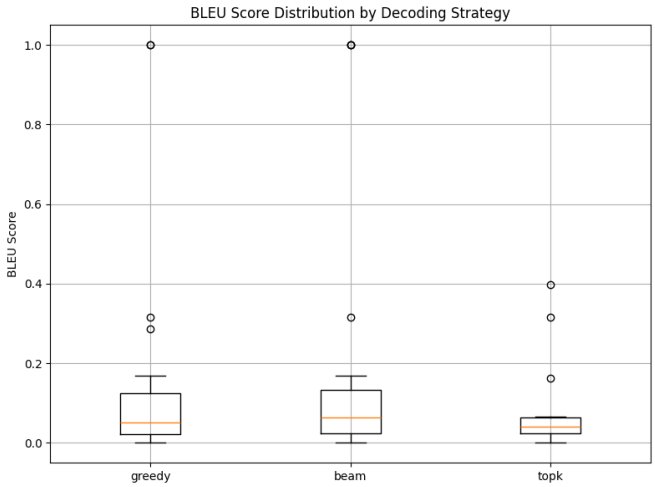


图 6

从表中可见，束搜索解码策略以 0.2050 的 BLEU 分数显著优于贪婪解码（0.1630）与 Top-k 采样（0.0476），说明其在生成译文时更能有效探索高概率路径，从而提升翻译质量。此外，折线图显示束搜索和贪婪解码在特定句子实现 BLEU 峰值（束搜索的峰值更多），显著优于 Top-k 采样，表明其对复杂语义更强的捕捉能力；箱线图进一步证实束搜索具有最高中位数，揭示其整体性能优势，贪婪解码性能与束搜索相近，前-k 采样因随机性导致分布离散度高。束搜索被验证为精准性需求场景的最优解。

束搜索与 Top-k 采样在不同参数下（参数取值范围[3,12]）的性能如下：

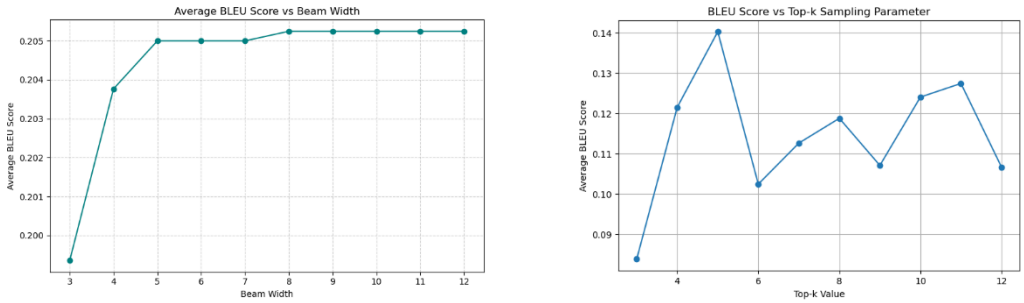


图 7

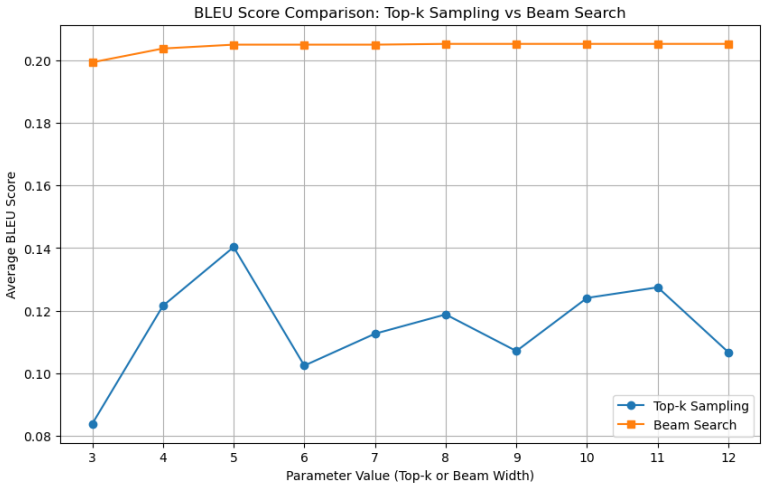


图 8

由上图可知，束搜索的平均 BLEU 分数随束宽的增加而递增，直到束宽为 8 时达到峰值且随后呈一条直线。而 Top-k 的发挥却非常的差，不仅波动大，而且其峰值 0.14 甚至不如束搜索的最低值 0.20。

综上所述，解码方式将采用束搜索，束宽设置为 8，以求最佳翻译效果。

翻译应用

完整翻译效果请见附录，这里展示部分翻译效果（原文的第 1,6,11,16,21 句）：

原文（中文）	参考译文（Reference）	模型翻译（Generated）
你好。	Hello.	Hello there.
她住在北京。	She lives in Beijing.	She lives in Beijing.
你会说英语吗？	Can you speak English?	Can you speak English?
近年来，人工智能的发展改变了我们的生活方式。	In recent years, the development of artificial intelligence has changed our way of life.	And over the years, behavior has changed our lifestyle.
我妈妈说我爸爸不在家。	My mom said my dad wasn't home.	My mom says, "No dad's not in my family.

从翻译样例可以看出，模型在翻译日常短句（如“她住在北京”、“你会说英语吗？”）时表现稳定，能够准确还原原文语义。然而，对于结构较复杂或信息密度较高的句子（如“近年来，人工智能的发展改变了我们的生活方式”），模型存在语义漂移和表达不清的问题，反映出其在处理长句和抽象概念时的能力仍有限。此外，个别句子（如“我妈妈说我爸爸不在家”）的翻译出现明显语法和语义错误，表明模型在上下文理解和句法结构生成方面仍有提升空间。整体来看，模型在简单句场景下翻译质量较高，但在复杂语境下仍需进一步优化。

附录

子词编码算法原理（BPE）

Byte Pair Encoding（BPE）是一种基于频率的子词切分算法，广泛用于神经机器翻译、语言建模等任务中。其基本思想是：

1. 初始将文本视为由单个字符构成的序列；
2. 在每轮迭代中，找出出现频率最高的相邻字符（或子词）对；
3. 将其合并为一个新的子词；
4. 重复执行，直到达到预定的词汇表大小。

例如，对输入序列 l o w, l o w e s t，BPE 会逐步合并 l+o→lo，再合并 lo+w→low，最终形成如 low, est, er 等子词。

该方法的优势在于能够学习到语言中的常见词干、前缀、后缀等结构，同时避免构建过大的词表，并显著缓解了低频词和 OOV 的建模难题。

Transformer 主体代码

```
class TransformerModel(nn.Module):
    def __init__(self, src_vocab_size, tgt_vocab_size,
                 d_model=512, nhead=8,
                 num_encoder_layers=6, num_decoder_layers=6,
                 dim_feedforward=2048, dropout=0.1):
        super().__init__()

        # 词嵌入层：将 token id 映射为向量，padding_idx=0 表示忽略 PAD
        self.src_embedding = nn.Embedding(src_vocab_size, d_model, padding_idx=0)
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model, padding_idx=0)

        # 位置编码：添加序列位置信息（PositionalEncoding 请见代码附件）
        self.pos_encoder = PositionalEncoding(d_model)
        self.pos_decoder = PositionalEncoding(d_model)

        # Transformer 编码器+解码器主体
        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=dim_feedforward,
            dropout=dropout
        )

        # 输出层：将 transformer 输出映射为目标词表大小的 logits
        self.fc_out = nn.Linear(d_model, tgt_vocab_size)
```

```
        # 保存 d_model，用于缩放 embedding
        self.d_model = d_model

    def forward(self, src, tgt):
        """
        前向传播：
        src, tgt: [batch_size, seq_len]，输入和输出的 token ID 序列
        """
        # 构造 mask
        src_mask = self._generate_padding_mask(src)
        tgt_mask =
self._generate_square_subsequent_mask(tgt.size(1)).to(tgt.device)
        tgt_padding_mask = self._generate_padding_mask(tgt)

        # 词嵌入 + 位置编码，并乘以 √d_model
        src_emb = self.pos_encoder(self.src_embedding(src) *
math.sqrt(self.d_model))
        tgt_emb = self.pos_decoder(self.tgt_embedding(tgt) *
math.sqrt(self.d_model))

        # 调整为 [seq_len, batch, d_model] 以符合 PyTorch Transformer API
        src_emb = src_emb.transpose(0, 1)
        tgt_emb = tgt_emb.transpose(0, 1)

        # 输入 transformer 模型
        output = self.transformer(
            src_emb, tgt_emb,
            src_key_padding_mask=src_mask,          # 忽略 src 中的 PAD token
            tgt_mask=tgt_mask,                     # 强制自回归（防止看到后面的词）
            tgt_key_padding_mask=tgt_padding_mask   # 忽略 tgt 中的 PAD token
        )

        # 输出维度从 [seq_len, batch, d_model] → [seq_len, batch, vocab_size]
        output = self.fc_out(output)

        # 转回 [batch, seq_len, vocab_size] 以便后续 loss 或 softmax
        return output.transpose(0, 1)

    def _generate_square_subsequent_mask(self, sz):
        """
        生成解码器的上三角 mask，防止模型看到未来的信息（自回归）
        结果是 [sz, sz] 的矩阵，下三角为 0，上三角为 -inf
        """
```

```
        mask = torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)
        return mask

    def _generate_padding_mask(self, seq):
        """
        为源或目标序列生成 padding mask
        输入: [batch_size, seq_len]
        输出: [batch_size, seq_len]（True 表示是 PAD，需要 mask 掉）
        """
        return (seq == 0)
```

完整翻译效果

原文（中文）	参考译文（Reference）	模型翻译（Generated）
你好。	Hello.	Hello there.
谢谢你。	Thank you.	Thank you.
我喜欢这本书。	I like this book.	And I loved this book.
今天天气很好。	The weather is nice today.	It's very good weather.
他正在看电视。	He is watching TV.	So he's walking.
她住在北京。	She lives in Beijing.	She lives in Beijing.
我不喜欢咖啡。	I don't like coffee.	I don't like coffee.
我昨天没去上班。	I didn't go to work yesterday.	I wasn't there yesterday.
他已经完成了作业。	He has already finished his homework.	He has done it.
你叫什么名字？	What’s your name?	What is your name?
你会说英语吗？	Can you speak English?	Can you speak English?
明天你有空吗？	Are you free tomorrow?	Can you have another tomorrow?
虽然他很累，但他还是完成了工作。	Although he was tired, he still finished the work.	He got angry, and he worked pretty well.
如果天气好，我们就去爬山。	If the weather is good, we will go hiking.	If we go to the weather, we're going to get good weather.

原文（中文）	参考译文（Reference）	模型翻译（Generated）
我喜欢猫，因为它们很可爱。	I like cats because they are cute.	I love cats, because they're cute.
近年来人工智能改变了生活方式。	In recent years, the development of artificial intelligence has changed our way of life.	And over the years, behavior has changed our lifestyle.
项目目的是提高学生英语口语能力。	The purpose of this project is to improve students’ spoken English skills.	And the English medium focused on trying to express itself.
未来十年气候变化或成最大挑战。	In the next decade, climate change may become one of the greatest challenges the world faces.	Many of the biggest challenges in the next decade will be the decade of climate change.
他打了他一巴掌。	He slapped him.	And he's following me over.
我以为你不会来了。	I thought you wouldn’t come.	I thought you're okay.
妈妈说爸爸不在家。	My mom said my dad wasn't home.	My mom says, "No dad's not in my family.