华东师范大学数据科学与工程学院实验报告

课程名称：当代人工智能	**年级：**2021 级	**上机实践成绩：**
指导教师：李翔	**姓名：**李睿恩	**学号：**10215501434
上机实践名称：文本摘要		**上机实践日期：**2023.12.22
上机实践编号：4	**组号：**无	**上机实践时间：**0:33

一、实验目的

- 实现文本摘要在医学领域的应用。

二、实验任务

- 构建 Seq2Seq 模型完成此次文本摘要任务。
- 选用一种 Encoder-Decoder 结构，选择包括但不限于 RNN / LSTM / GRU / Transformer / BERT / BART / T5 / OPT 等。本次实验实现了 BART 与 T5。
- 使用一种或多种评估指标对文本摘要结果进行分析。

三、实验环境

本次实验基于 **AutoDL 算力云**平台。我们在 AutoDL 算力云平台中租用了 2 块 RTX 4090(24GB)的 GPU，并在其提供的 JupyterLab 平台中进行了本次实验。

我们所租用的实验环境基于 Ubuntu 20.04，采用 Python 3.8 作为编程语言，并使用了 PyTorch 2.0.0 框架进行深度学习任务，同时借助 CUDA 11.8 实现对 GPU 的加速。

四、实验过程

4.1 Seq2Seq 与 Encoder-Decoder 结构介绍

Seq2Seq（Sequence-to-Sequence）是一种神经网络架构，用于处理序列到序列的任务。它是自然语言处理（Natural Language Processing）领域目前主流的神经网络架构。

Seq2Seq 模型一般由两个主要部分组成：Encoder（编码器）与 Decoder（解码器）。

Encoder 一般负责将输入序列转换为一个固定维度的表示，这个表示通常被称为隐藏状态向量，该向量包含了输入序列的信息。Encoder 的目标就是捕捉输入序列的语义信息，使得这些信息能够之后被 Decoder 有效地使用。

Decoder 会接受 Encoder 生成的向量，将其解码为目标序列。解码器逐步生成目标序列的元素，每一步都依赖于前一步生成的元素。Decoder 的目标就是根据隐藏状态向量生成与输入序列对应的输出序列。

Encoder-Decoder 架构如今被广泛使用，他们被使用在机器翻译、语音识别、文本摘要等任务重，有着非常不俗的表现。

4.2 Tranformer 架构简介

传统的循环神经网络（RNN）、长短期记忆神经网络（LSTM）与门控循环单元（GRU）是早期用于进行文本摘要的工具。但是，他们的效果并不让人满意。一方面，他们会存在较长的链条，这使得他们存在长期依赖问题，进而导致难以解决的梯度消失或梯度爆炸。另外，他们是逐步处理序列的，这使得他们难以掌握全局的信息。

Transformer 是一种更现代、基于自注意力机制（Self-attention）的神经网络架构。自注意力机制允许 Transformer 直接对整个序列进行并行处理，提高了计算效率和模型的学习能力。Transformer 还引入了位置编码，通过在输入嵌入中添加位置信息，模型可以区分不同位置的元素。

最新的 Transformer 还会使用多头注意力机制（Multi-head Attention），使得 Transformer 能够学习多个不同的注意力权重，从而更好地捕捉输入序列中的不同关系，使得模型更具表达能力。

基于上述原因，Transformer 成为了目前最流行的架构。

目前最流行的自然语言处理中会使用的架构有 BERT, GPT, T5 等，他们也是基于 Transformer 架构的 Encoder-Decoder 架构。区别在于，BERT 是只使用了 Encoder 的架构，GPT 是只使用了 Decoder 的架构，而 T5 是完整使用了 Encoder 与 Decoder 的架构。图 1 展示了 Transformer 架构的一个示意图。

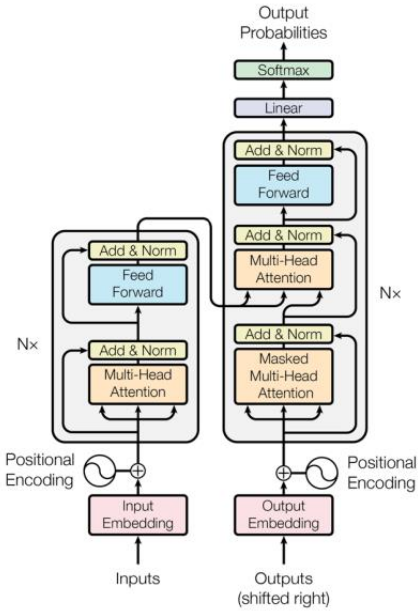

图 1 Transformer 架构示意图

4.3 T5 架构简介

T5 框架将所有自然语言处理任务重构为统一的文本到文本格式。其输入和输出始终是文本字符串，这与 BERT 模型形成了鲜明对比，因为 BERT 类的模型的输入有一定的局限性。T5 框架允许我们在任何自然语言处理任务上使用相同的模型、损失函数与超参数，这包括机器翻译、文本摘要、机器问答和分类任务。图 2 展示了 T5 架构能实现的功能。

T5 框架是一个有效的迁移学习技术。在开源的网站上获取的 T5 框架是受到过预训练的，我们只需要对框架进行微调，就可以将其作为有效的模型投入到使用中。

T5 的模型有许多规模，如 T5-Small，T5-Base，T5-Large。在这里，我们将尝试使用 T5-Base 框架，基于 AutoDL 云上服务器，使用训练集对模型进行微调，从而尝试实现文本摘要。

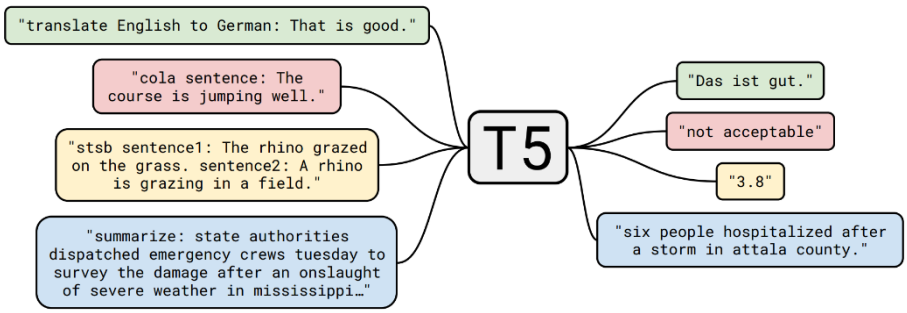

图 2 T5 架构功能介绍

4.3.1 引入必要模块

为了完成此次试验，一些模块的调用是必须的。调用的模块如图 3.

```
# 基础模块
import numpy as np
import pandas as pd
from rouge import Rouge
import torch
import torch.nn.functional as F
from sklearn.model_selection import KFold
from nltk.translate.bleu_score import sentence_bleu
from nltk.translate import meteor_score
from torch.utils.data import Dataset, DataLoader, RandomSampler, SequentialSampler

# 基于Transformers架构的T5框架
from transformers import BartTokenizer, BartForConditionalGeneration
```

图 3 调用模块

其中，numpy 与 pandas 是用于数据处理的模块；torch 模块用于创建张量、加载数据、创建优化器与构建神经网络；nltk.translate 模块可以协助我们使用 BLEU-4 和 METEOR 评估指标观测模型效果；transformers 架构可以允许我们调用 T5 的预训练模型；我们还调用了 Rouge 模块，使用 Rouge 评估指标观测模型效果。

4.3.2 自定义数据集

本次实验我们将使用给定的 train.csv 文件作为训练集，test.csv 文件作为测试集。为了能够更适配深度学习的模型，我们编写了 MedicalDataset 函数，从而使得数据集被处理成 PyTorch 更能适应的数据格式。

在该函数中，我们将数据统一了格式。首先，如果数据文本中有多个连续空格，我们会将这些空格合并为一个空格，避免数据格式的不统一。之后，我们会将文本转为模型可以处

理的编码（即 Encoder），但这些数据有一个最大长度限制，并且如果编码后的长度不满足最大长度限制，我们会要求编码对空缺位进行填充。我们会要求最后返回的编码是 PyTorch 可以处理的张量的形式，并且如果最后得到的编码超出了最大长度，我们会截断超出最大长度的部分。最后，我们会使用 PyTorch 中的 squeeze 函数，移除得到的编码中大小为 1 的维度，使得张量形状更加紧凑。其中，source_ids 与 target_ids 返回了编码后的 token IDs，source_mask 返回了二进制掩码，表明哪些位置是填充位，是注意力机制不应该关注的地方。图 4 展示了我们编写的 MedicalDataset 函数。

```
def __getitem__(self, index):
    # 如果文本中有多个连续空格
    # 则合并为一个空格
    diagnosis = str(self.diagnosis[index])
    diagnosis = ' '.join(diagnosis.split())
    description = str(self.description[index])
    description = ' '.join(description.split())
    # batch_encode_plus将文本转为模型可以处理的编码
    source = self.tokenizer.batch_encode_plus([description],
                                              max_length = self.source_len,     # 指定编码后文本最大长度
                                              padding = 'max_length',           # 填充使得每个文本长度相同
                                              return_tensors = 'pt',            # 返回Pytorch的张量
                                              truncation = True)                # 如果超过最大长度，需要截断
    target = self.tokenizer.batch_encode_plus([diagnosis],
                                              max_length = self.summ_len,       # 指定编码后文本最大长度
                                              padding = 'max_length',           # 填充使得每个文本长度相同
                                              return_tensors = 'pt',            # 返回Pytorch的张量
                                              truncation = True)                # 如果超过最大长度，需要截断

    # Pytorch中的squeeze函数移除大小为1的维度
    # 张量形状更紧凑
    # input_ids返回了编码后的文本的token IDs
    # attention_mask二进制掩码，表明哪些位置是填充，注意力机制不应放在填充位置
    source_ids = source['input_ids'].squeeze()
    source_mask = source['attention_mask'].squeeze()
    target_ids = target['input_ids'].squeeze()

    return {
        'source_ids': source_ids.to(dtype=torch.long),
        'source_mask': source_mask.to(dtype=torch.long),
        'target_ids': target_ids.to(dtype=torch.long)
    }
```

图 4 MedicalDataset 的实现

4.3.3 编写训练函数

我们需要基于确定格式的数据，设计训练函数。

在训练的初期，我们会要求获取输入数据的每一行，并摒弃最后一个标记位。随后，我们提取摘要文本除最后一个编码以外的所有编码作为 Decoder 的输入，记为 y_ids，并且在这里我们还会使用 contiguous 函数使得张量连续。我们也提取了摘要文本除第一个文本以外的所有编码作为 Decoder 正确的输出，记为 labels，并且在这里我们还使用了 detach 函数，要求这部分数据不参与梯度的计算。接下来，考虑到 labels 中存在一些因为填充而存在的没有意义的编码位，我们要求这些位置的数值被计为-100，这个数字在 PyTorch 中是一个特殊值，能使得我们在计算时可以忽略这些位置。

基于此，我们可以将这些数据投入到模型的训练中。我们在训练的过程中还要求，每遍历 10 个数据，就必须记录当时的损失函数，每遍历 500 个数据，就必须输出目前的损失函数值。这样的方式有助于我们可视化训练的过程，确保我们的模型的确学习到了文本的特征。

最后，我们使用反向传播，真正完成深度学习模型的训练过程。图 5 展示了 train 函数的

编写。

```
loss_array = []

def train(epoch, tokenizer, model, device, loader, optimizer):
    model.train()
    for _, data in enumerate(loader):
        y = data['target_ids'].to(device, dtype = torch.long)
        # 获取 y 的每一行，且不要最后一个标记
        # 用contiguous使张量连续
        y_ids = y[:, :-1].contiguous()
        # detach让labels不用参与梯度计算
        labels = y[:, 1:].clone().detach()
        # 填充位置对应的损失值设为-100
        # -100是Pytorch中的一个特殊值，计算损失时会忽略这些位置
        labels[y[:, 1:] == tokenizer.pad_token_id] = -100
        ids = data['source_ids'].to(device, dtype = torch.long)
        mask = data['source_mask'].to(device, dtype = torch.long)

        outputs = model(input_ids = ids,
                        attention_mask = mask,
                        decoder_input_ids = y_ids,
                        labels = labels)
        loss = outputs[0]

        if _%10 == 0:
            loss_array.append(loss.item())
        if _%500 == 0:
            print(f'Epoch: {epoch + 1}, Loss: {loss.item()}')

        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

图 5 train 的实现

4.3.4 编写测试函数

测试函数与训练函数在逻辑上有不小的区别。

我们开启模型的验证模式，并且设置一定的参数。我们对该模型输入原始文本，设置注意力机制中的掩码（attention_mask），并要求其最终生成的摘要的长度（max_length）不可以超过 150。在这里，我们还结合了束搜索（Beam Search）的思路，在每个时间步，选择当前最有可能的 2 个假设序列（num_beams），从而保留文本的多样性。我们还为重复标记设置惩罚（repetition_penalty），这可以鼓励模型生成不包含重复标记的序列。我们设置了长度惩罚（length_penalty），防止模型输出过短或过长的序列。另外，我们还设置了早停（early_stopping），要求模型在遇到序列结束标记的时候停止生成，如果不设置该值为 True，则模型会生成直到达到最大长度。

最后使用 Decoder，我们可以获得生成的文本与原始的文本。图 6 展示了验证过程的代码。

4.3.5 设置训练背景

在真正投入预测使用前，我们需要设置一些常数与训练背景。

我们设置了 tokenizer 是基于 Hugging Face Transformers 库中的 T5Tokenizer 类中的预训练过的 T5-Base 模型的分词器。这个分词器专门用于训练 T5-Base 的模型。

```python
def validate(tokenizer, model, device, loader):
    model.eval()
    predictions = []
    actual = []
    with torch.no_grad():
        for _, data in enumerate(loader):
            y = data['target_ids'].to(device, dtype = torch.long)
            ids = data['source_ids'].to(device, dtype = torch.long)
            mask = data['source_mask'].to(device, dtype = torch.long)

            generated_ids = model.generate(input_ids = ids,
                                           attention_mask = mask,
                                           max_length = 150,
                                           # Beam Search的参数数量
                                           num_beams = 2,
                                           # 重复惩罚
                                           repetition_penalty = 2.5,
                                           # 长度惩罚
                                           length_penalty = 1.0,
                                           # 要求在遇到End of Sequence时停止
                                           # 若设置为False，会生成直到最大长度
                                           early_stopping = True)
            preds = [tokenizer.decode(g,
                                      skip_special_tokens = True,
                                      clean_up_tokenization_spaces = True) for g in generated_ids]

            target = [tokenizer.decode(t,
                                       skip_special_tokens = True,
                                       clean_up_tokenization_spaces = True) for t in y]
```

图 6 validate 的实现

我们先观测数据集中文本的大小。通过数据处理与绘图，我们可以得到如图 7 的原始文本长度统计与如图 8 的摘要文本长度统计。这有助于我们确定编码的合适长度，以得到更好的模型表现。

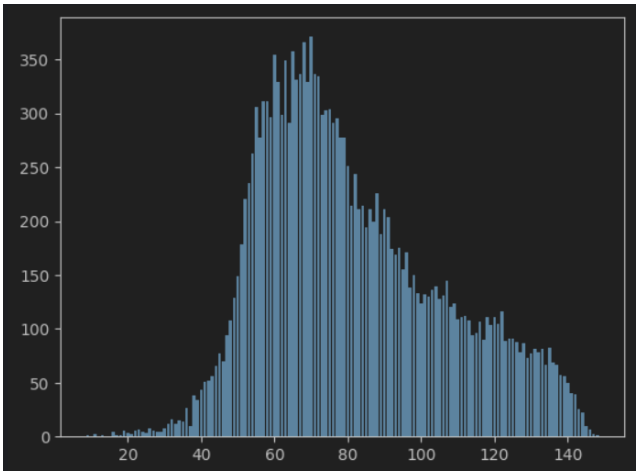

图 7 原始文本长度统计

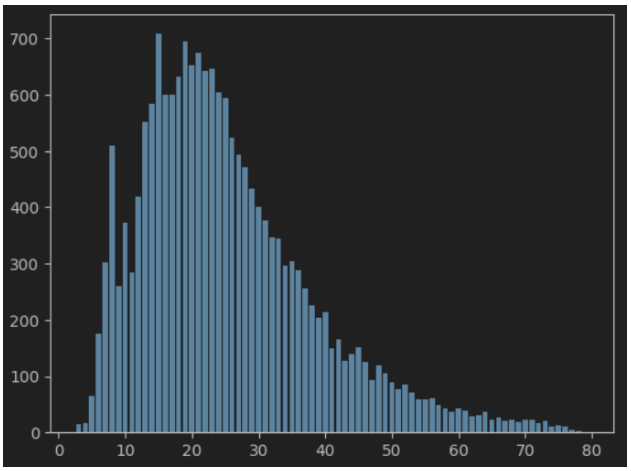

图 8 摘要文本长度统计

我们设定了训练的批次大小（TRAIN_BATCH_SIZE）为 2，测试的批次大小（VALID_BATCH_SIZE）为 2，从而让 GPU 可以发挥其效用。我们还设置了训练轮数（TRAIN_EPOCHS）为 3，这主要是因为在我们的多次实验中，我们发现过多的轮数不但耗时，最后得到的损失函数值也没有显著的下降，而且它还可能还导致了过拟合现象。我们另外设置了学习率为 10^{-4}，设置一个用于 numpy 与 torch 随机划分的随机数种子（SEED）为 42，规定上文提到的原始文本编码最大长度为 512，摘要编码最大长度为 150。最后，我们设置 torch.backends.cudnn.deterministic = True，这确保了每次训练的可复现性。

训练背景的设置整体如图 9 所示。

```
# 设定参数
TRAIN_BATCH_SIZE = 2 # 训练的批次大小
VALID_BATCH_SIZE = 2 # 测试的批次大小
TRAIN_EPOCHS = 3     # 训练的轮次数量
LEARNING_RATE = 1e-4 # 学习率
SEED = 42            # 随机种子
MAX_LEN = 512        # source_len
SUMMARY_LEN = 150    # summ_len

# 设置随机种子
torch.manual_seed(SEED)
np.random.seed(SEED)

# 可复现性
torch.backends.cudnn.deterministic = True

# Tokenizer
tokenizer = T5Tokenizer.from_pretrained("t5-base")
```

图 9 训练背景设置

4.3.6 五折交叉验证

为了验证我们的模型是否是合理的，我们在此进行五折交叉验证。我们调用 KFold 模块，从而确保我们每次都采取了完全没有重叠的验证集。

我们需要先进行模型的训练。我们使用了自定义数据集的函数 MedicalDataset，使用 DataLoader 进一步统一数据格式。随后，我们调用 T5ForConditionalGeneration 中的 T5-Base 预训练模型，我们将基于该预训练模型进行微调。最后，我们选择 Adam 作为优化器以进行对 T5-Base 模型的微调。五折交叉验证的训练代码执行流程如图 10.

```
for train_index, test_index in kfolds.split(df):
    des_trainset = df['description'][train_index]
    dia_trainset = df['diagnosis'][train_index]
    des_valset = df['description'][test_index]
    dia_valset = df['diagnosis'][test_index]
    train_dataset = pd.concat([des_trainset, dia_trainset], axis = 1).reset_index(drop = True)
    val_dataset = pd.concat([des_valset, dia_valset], axis = 1).reset_index(drop = True)

    training_set = MedicalDataset(train_dataset, tokenizer, MAX_LEN, SUMMARY_LEN)
    val_set = MedicalDataset(val_dataset, tokenizer, MAX_LEN, SUMMARY_LEN)

    training_loader = DataLoader(training_set, **train_params)
    val_loader = DataLoader(val_set, **val_params)

    # 使用预训练的T5-base模型
    model = T5ForConditionalGeneration.from_pretrained("t5-base")
    model = model.to(device)

    # 定义优化器，使用Adam
    optimizer = torch.optim.Adam(params = model.parameters(), lr = LEARNING_RATE)

    for epoch in range(TRAIN_EPOCHS):
        train(epoch, tokenizer, model, device, training_loader, optimizer)
```

图 10 五折交叉验证的训练代码实现

然后我们需要进行模型的验证。只需要直接调用 validate 函数即可实现模型的验证。最后，我们将每次交叉验证使用的原始摘要文本与生成的摘要文本存储在一个.csv 文件中。五折交叉验证的验证代码如图 11 所示。

```
predictions, actual = validate(tokenizer, model, device, val_loader)
final_df = pd.DataFrame({'Generated Text':predictions, 'Actual Text':actual})
final_df.to_csv('predictions' + str(num) + '.csv')
```

图 11 五折交叉验证的验证代码实现

最终我们可以得到如图 12 的输出。

```
Epoch: 3, Loss: 1.540496826171875
Epoch: 3, Loss: 0.765545666217804
Epoch: 3, Loss: 1.7242399454116821
Epoch: 3, Loss: 0.8849756717681885
在验证集上测试
Completed 0
Completed 100
Completed 200
Completed 300
```

图 12 五折交叉验证的训练与验证输出

4.3.7 生成摘要的评估

在这里，我们将使用 Rouge（Recall-Oriented Understudy for Gisting Evaluation）、BLEU（Bilingual Evaluation Understudy）和 METEOR（Metric for Evaluation of Translation with Explicit Ordering）来评估生成文本与原始文本的相似度。

Rouge-N 一般的计算方法为，分母是标准答案中 n-gram 的总数量，分子是模型生成的摘要和标准答案共有的 n-gram 的个数，即召回率。在 Python 中的 Rouge 模块中，Rouge 生成的还有准确率与 F1（即 β=1 的情况下的 F 值）分数。Rouge-L 则是参考了最长公共子序列的计算方式，分子总是最长公共子序列的长度。如果我们设 m 为原始文本的长度，n 是生成文本的长度，那么 Rouge-N 方法的公式如图 13，Rouge-L 方法的公式如图 14.

$$\text{ROUGE-N} = \frac{\sum_{S\in\{ReferemceSummaries\}}\sum_{gram_n\in S} Count_{match}(gram_n)}{\sum_{S\in\{ReferenceSummaries\}}\sum_{gram_n\in S} Count(gram_n)}$$

图 13 Rouge-N 公式

$$R_{lcs} = \frac{LCS(X,Y)}{m}$$
$$P_{lcs} = \frac{LCS(X,Y)}{n}$$
$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs}+\beta^2 P_{lcs}}$$

图 14 Rouge-L 公式

BLEU 通常被用于机器翻译。它通过比较输出和多个参考文本之间的 n-gram 重叠来工作。通常我们使用的为 BLEU-4，这个指标会考虑 1-gram，2-gram，3-gram 和 4-gram 的精度综合计算得到 BLEU-4 的数值。在计算 BLEU 时，我们会设置对短句的惩罚因子 BP，如果设 r 为参考语料库总长度，c 为翻译语料库总长度，那么 BP 的计算方法如图 15 所示。最后，我

们结合 BP 和先前计算的 n-gram 精度，可以得到 BLEU-4 的计算公式，如果我们设置 w_n 为 n-gram 的权重，p_n 为 n-gram 的取值，我们可以得到如图 16 的计算公式。

$$\mathrm{BP}=\begin{cases}1 & \text{if } c>r\\ e^{(1-r/c)} & \text{if } c\le r\end{cases}$$

图 15 惩罚因子计算公式

$$\mathrm{BLEU}=\mathrm{BP}\cdot\exp\left(\sum_{n=1}^{N}w_n\log p_n\right)$$

图 16 BLEU 计算公式

METEOR 也是通常被用于机器翻译的方法。它考虑了同义词匹配、词干匹配以及词序。我们需要先结合准确率 P 和召回率 R 计算 Fmean，计算公式如图 17。之后，我们计算碎片度惩罚。我们记在生成摘要与原始摘要中能够对齐的、且在各自句中连续排列的单词形成一个 chunk，#chunk 为 chunk 的数量。而#unigram_matched 则被我们记为生成摘要中能够被匹配的一元词组的数量。惩罚的计算公式如图 18。之后，使用如图 19 的公式即可计算出 METEOR 分数。

$$Fmean=\frac{10PR}{R+9P}$$

图 17 Fmean 计算公式

$$Penalty=0.5*\left(\frac{\#chunks}{\#unigrams\_matched}\right)^3$$

图 18 惩罚计算公式

$$Score=Fmean*(1-Penalty)$$

图 19 METEOR 计算公式

这些方法各自存在一定的缺点。Rouge 方法在评估生成文本时，对于不同的词汇表达可能不是很敏感。BLEU 只关注 n-gram 的匹配度，也因此忽略了语法结构和语义的一致性。METEOR 虽然考虑地比较全面，但设计最为复杂。

我们将使用这些指标评估我们的模型。

我们在五折交叉验证中，得到的每次验证的 Rouge、BLEU-4 与 METEOR 得分如表 1.

表 1 T5 五折交叉验证的评估

评估指标	验证 1	验证 2	验证 3	验证 4	验证 5
Rouge 1-P	0.633	0.623	0.623	0.593	0.670
Rouge 1-R	0.406	0.399	0.418	0.367	0.402
Rouge 1-F1	0.462	0.457	0.470	0.426	0.474
Rouge 2-P	0.426	0.413	0.419	0.396	0.459
Rouge 2-R	0.265	0.262	0.276	0.240	0.263
Rouge 2-F1	0.302	0.299	0.310	0.278	0.312
Rouge L-P	0.608	0.599	0.599	0.571	0.647
Rouge L-R	0.389	0.384	0.402	0.353	0.389
Rouge L-F1	0.442	0.439	0.453	0.410	0.458

BLEU-4	0.309	0.304	0.323	0.278	0.291
METEOR	0.368	0.361	0.380	0.331	0.365

经过比对，我们认为这个五折交叉验证的结果是比较好的。因此，我们将该模型正式投入训练与测试中。

4.3.8 T5 进行实际测试

我们通过与上述过程类似的方法，用完整的训练集对模型进行了微调。我们最终得到了如图 20 所示的损失函数曲线，这可以说明我们实际微调的过程是很有效的。

图 20 T5 架构微调的损失函数曲线

最终我们将实际测试得到的摘要文本进行评估，得到了如表 2 的评估结果。

表 2 T5 模型测试的评估

Rouge 1-P	Rouge 1-R	Rouge 1-F1	Rouge 2-P	Rouge 2-R	Rouge 2-F1
0.666	0.404	0.472	0.460	0.272	0.318

Rouge 3-P	Rouge 3-R	Rouge 3-F1	BLEU-4	METEOR
0.645	0.392	0.458	0.302	0.371

这个表现是比较好的，因此我们可以认为 T5 在文本摘要中可以有不错的表现。

4.4 BART 架构简介

BART（Bidirectional and Auto-Regressive Transformers）是由 Facebook AI Research 的研究人员发布的一种架构，在自然语言处理领域有较好的效果。它的特点在于，它的 Encoder 是双向的，可以同时考虑输入序列的上下文信息，有助于捕获更丰富的语义信息。它的 Decoder 是自回归的，这意味着在生成输出序列时，模型会逐步生成每个词，将前面的词作

为上下文信息，这使得 BART 在文本生成、文本摘要类任务中有不错的表现。而且，BART 也是经过预训练的，只需要我们进行微调就可以得到不错效果的架构。BART 的示意图如图 21 所示。

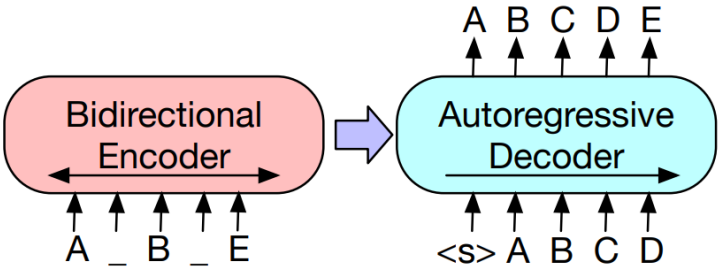

图 21 BART 示意图

4.4.1 BART 进行实际测试

对 BART 架构进行测试非常简单，我们只需要将之前为 T5 架构编写的代码中的 T5Tokenizer 改为 BartTokenizer，T5ForConditionalGeneration 改为 BartForConditionalGeneration，代码中的“t5-base”改为“facebook/bart-base”即可。

其余对数据集的预处理，训练背景的设置，大致均与先前的设置一样，不作过多修改。这里有一个地方的修改必须提及，如果我们完全按照 T5 模型所使用的参数来运行代码，我们意外发现最后无论我们输入的是怎么样的数字序列，最后得到的摘要都是一模一样的。由于最后生成的摘要长度都比较短，我们怀疑是生成摘要的编码长度限制太短考虑到该模型的学习出现了问题，因此我们增加了生成摘要的编码长度为 200。经过这样的修改后，我们得到了不错的结果。

BART 也是大模型之一，进行五折交叉验证的时间非常久，碍于实验的时间限制，我们没有对模型进行五折交叉验证，而是直接将其投入到实际测试中，观察实验结果。如果其最终实验结果较好，我们可以认为该模型的训练较为成功。

图 22 展示了 BART 架构的微调过程中的损失函数曲线。

图 22 BART 架构微调的损失函数曲线

最终我们将实际测试得到的摘要文本进行评估，得到了如表 3 的评估结果。

表 3 BART 模型测试的评估

Rouge 1-P	Rouge 1-R	Rouge 1-F1	Rouge 2-P	Rouge 2-R	Rouge 2-F1
0.640	0.543	0.564	0.448	0.380	0.393

Rouge 3-P	Rouge 3-R	Rouge 3-F1	BLEU-4	METEOR
0.609	0.517	0.537	0.465	0.497

4.5 模型对比

作为大模型，他们的训练都需要耗费极大的算力与时间成本。即使是租用了具有 2 块 RTX-4090，T5-Base 与 Bart-base 每一次训练 3 个 Epoch 都需要耗费 30~40 分钟的时间。在这一点上，他们并没有过多的区别。

在最终的模型测试中，我们可以发现他们在各个评估指数中有着不同的表现。图 23 展示了二者的显著对比。

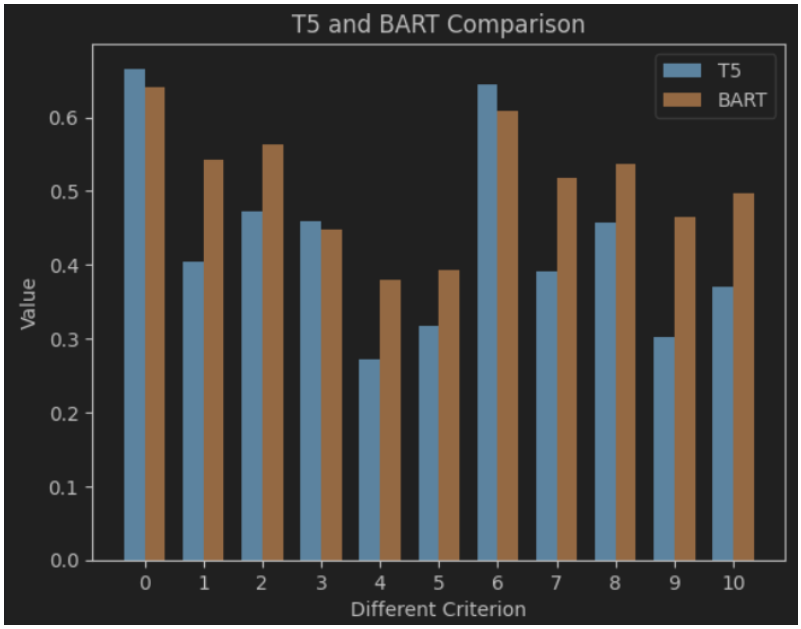

图 23 T5 与 BART 各指标对比

我们会发现，T5 模型在衡量 Rouge 中的准确度指标时，表现通常略好于 BART，但一旦涉及召回率、F1 分数、考虑 n-gram 的 BLEU 与考虑较为综合的 METEOR，BART 的表现就会明显优于 T5。

如果这不是由于我们对 T5 的参数设置过差，这可以说明我们所编写的 BART 在此次实验中，通常意义上的效果最好。

4.6 难点与解决

在本次实验中，我们的确遇到过多种问题。

由于我们知道 RNN、LSTM 等传统模型已经不再流行，因此我们大胆挑战了搭建大模型。但是，大模型的搭建是我们从未涉足过的领域，因此理解大模型的编写逻辑对我们来说是挑战的第一关。本人甚至在尚未完全理解大模型总体结构的基础上，连微调都尚未进行就去尝试对摘要进行生成。但最终，我们查阅大量技术文档与参考文献，找到了一定的思路。

我们此次的代码参考了一个在 github 中的仓库。

（URL: https://github.com/abhimishra91/transformers-tutorials）

尽管该仓库在本作业完成时（2023 年 12 月 26 日）已经获得了 778 个 Stars，但在我们试图复现其中的代码的过程中，我们发现代码会出现错误。经过研究报错，我们发现其中一个函数的参数名字是错误的，他们误将自己命名的参数 lm_labels 作为了函数中的本应名为 labels 的参数的名字，这导致代码的运行无法正常运作。目前，本人对该仓库提交了一个 Pull Request，并对仓库的拥有者发出了一封邮件，但目前对方尚未有任何回复，仓库仍然停留在十个月前的最后一次更新。

在我们运行 Bart-base 模型时，我们误以为只需要维持在 T5-base 中使用的参数就可以获得不错的结果，但仔细观察生成的预测文件，我们发现每一个文本对应的生成摘要居然都是一模一样的数字串，这让我们感到非常沮丧。但我们通过观察这些生成摘要的特征，怀疑我们是将生成摘要的最长编码设置地过短，进而修改了参数，最终得到了比 T5-base 还要优秀的结果。

五、 总结与分析

Seq2Seq（Sequence-to-Sequence）是一种神经网络架构，用于处理序列到序列的任务。它是自然语言处理（Natural Language Processing）领域目前主流的神经网络架构。不过，传统的结构如 RNN，LSTM，GRU 因为效果相对较差已经失势，目前的主流是使用了注意力机制（Attention）的 Transformer 架构。自注意力机制允许 Transformer 直接对整个序列进行并行处理，提高了计算效率和模型的学习能力。目前最流行的自然语言处理中会使用的架构有 BERT, GPT, T5 等，他们也是基于 Transformer 架构的 Encoder-Decoder 架构。

Rouge、BLEU、METEOR 是自然语言处理领域常用的指标。Rouge 考虑了生成文本的内容覆盖度，但在评估生成文本时对于不同的词汇表达形式不太敏感。BLEU 使用 n-gram 精确匹配度衡量两个文本的相似性，但忽略了语法结构和语义的一致性。METEOR 相对来说综合考虑的内容较多，考虑了近义词和词形变化，但也因此设计较为复杂。

从实验结果来看，BART 在通常意义上可能比 T5 在文本摘要上有着更好的表现，尤其是在召回率，F1 分数，BLEU 与 METEOR 上。这可能可以启发我们之后在选择文本摘要的模型时，更多关注 BART 模型的表现。

大模型的运算非常消耗算力，一台普通的计算机可能对此无能为力。这需要我们寻求更高的算力平台（如租用服务器，置备更多 GPU）。大模型的代码编写相较于常见的代码而言，内容更长，函数更复杂，理解与编写有不小的难度，但大模型也是目前人工智能发展的重要趋势，因此我们在大模型上的学习道阻且长。

六、参考文献与技术文档

[1] Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74-81).

[2] Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311-318).

[3] Banerjee, S., & Lavie, A. (2005, June). METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization* (pp. 65-72).

[4] NLP 机器翻译全景：从基本原理到技术实战全解析（知乎）
URL: https://zhuanlan.zhihu.com/p/665885910

[5] abhimishra91 / transformers-tutorials（github Repository）
URL: https://github.com/abhimishra91/transformers-tutorials