当代人工智能实验四-文本摘要

--10215501435 杨茜雅

1、摘要：

在近年来的人工智能领域，深度神经网络（DNN）已成为一种革命性的技术，特别是在卷积神经网络（CNN）、循环神经网络（RNN）和 Transformer 模型方面。这些高级网络结构相较于传统的机器学习方法，它们通过层级结构自动提取和学习数据特征，有效克服了传统依赖人工特征设计的限制，从而显著提高了模型的性能。然而，深度学习模型对大规模数据的依赖性，尤其是在参数学习方面，也带来了显著的挑战，尤其是在数据量有限的情况下，可能导致过拟合和泛化能力不足的问题。

随着大数据技术和高性能计算硬件的飞速发展，自回归语言模型（如 GPT）和自编码语言模型（如 BERT）在自然语言处理领域取得了显著的进展。这些模型通过预训练和微调的方法在各种 NLP 任务中展现出卓越的性能。特别地，Facebook 开发的 BART 模型，结合了自编码和自回归特性，为处理复杂的语言任务提供了新的视角。

本研究利用 Facebook 预训练的 BART 模型，探讨了该模型在特定数据集上的应用效果。对 train.csv 数据进行预训练可以使得模型学习到数据上一些重要的权重，这些权重可以用来预测 train.csv 以外的数据。本实验还评估了模型的性能，观察训练中每个轮次的 loss 值变化并且采用 bleu_score 作为主要的评价指标来衡量模型的准确性。

2、任务定义：

文本摘要是指使用算法从一段较长文本中提取关键信息，生成简短、凝练的摘要。

本次实验是文本摘要在医学领域的应用:从详细的病情描述中提取关键信息生成简短清晰的诊断报告。

3、实验内容：

●构建 seq2seq 模型完成本次文本摘要任务;

● Encoder/Decoder 的选择包括但不限于 RNN/ LSTM/ GRU/Transformer/ BERT/ BART/T5/OPT 等

●评估指标的选择: BLEU-4、ROUGE、CIDEr 等

4、实验要求：

●选择一种 Encoder-Decoder 结构，使用哪种模型作为 Encoder 或 Decoder 不受限制，大家可以按照自己的硬件设备自行选择

●在硬件条件允许的情况下，尽量使用多种不同的模型并进行详细的实验分析

●评估指标的使用不受限制，可以仅使用一种或使用多种更加全面的进行结果分析

5、数据预处理

数据集已脱敏，这意味着数据集中的词都被转化为了数字并且无法再重新转化回单词进行语义的学习。

未脱敏的任务实例（医疗领域数据集）：

列名	数据类型	示例
description（输入）病情描述	string	左侧顶骨局部骨质缺如；两侧侧脑室旁见点状密度减低。右侧额部颅板下见弧形脑脊液密度影。脑室系统扩大，脑沟、裂、池增宽。中线结构无移位。双侧乳突气化差，内见密度增高。
diagnosis（输出）诊断报告	string	左侧顶骨局部缺如，考虑术后改变；脑内散发缺血灶；右侧额部少量硬膜下积液；双侧乳突炎。

已脱敏的任务数据集：

列名	数据类型	示例
description（输入）病情描述	string	101 47 12 66 74 90 0 411 234 79 175
diagnosis（输出）诊断报告	string	122 83 65 74 2 232 18 44 95

数据的训练集 train.csv 有 18000 条数据，有两个字段，description 和 diagnosis 分别代表输入和输出，需要自行划分验证集（80% 的数据被用作训练集，剩下的 20% 用作验证集）。测试集有 2000 条数据，模型学习到的权重可以用来预测 description 输入字段对应的 diagnosis 输出。

我们先通过 process_data 将训练和测试数据读取到列表中，对文本字段进行清洗和转换，并将结果以列表的形式返回。同时收集一些额外信息，如数据长度和唯一词的数量。在训练模式下，还会处理额外的 diagnosis 字段。

```python
def process_data(root:str, file_path:str, mode='train'):
    file = os.path.join(root, file_path)
    df = pd.read_csv(file, header=None, encoding='utf-8')
    len_data = 0
    data = []
    my_dict = set()
    data_str = ''
    res = []
    for i in range(1, len(df)):
        lst = [df[0][i]]
        # 移除开头和结尾的空格或者换行，并且将字符串中的多个空格转换为一个
        description = re.sub(' +', ' ', df[1][i].strip())
        res.append(description)
        description = [int(x) for x in description.split()]
        for i in range(len(description)):
            my_dict.add(description[i])
        lst.append(description)
        len_data = max(len_data, len(description))
        if mode == 'train':
            # 如果模式定义是训练模式，则需要读取 diagnosis
            diagnosis = re.sub(' +', ' ', df[2][i].strip())
            res.append(diagnosis)
            diagnosis = [int(x) for x in diagnosis.split()]
            lst.append(diagnosis)
            for i in range(len(diagnosis)):
                my_dict.add(diagnosis[i])
        data.append(lst)
    print(mode + '最大的 description 数据长度为：', len_data)
    print(mode + '的字典集合长度为：', len(my_dict))
    return data
```

再将其以 80：20 的比例分割成训练集与验证集，对数据进行 padding，即将短于最大长度 max_length 的序列用特殊的标记（通常是[0]）填充到最大长度，确保所有的输入序列长度统一，以便可以批量处理。（主要在 my_dataset.py 的 BartDataset 和 PredDataset 类中实现）

在 BartDataset 类中：

●对于描述（self.des），序列在末尾添加了结束标记[1]和必要数量的填充[0]，直到达到 self.enc_max_length。

●对于诊断输入（self.diag_inputs），序列前面添加了开始标记[2]，末尾添加了结束标记[1]和必要数量的填充[0]。

●为了计算 attention mask（self.des_attention_masks 和 self.diag_attention_masks），有效的序列部分标记为[1]，而填充部分标记为[0]。

```
class BartDataset(Dataset):
    def __init__(self, meta_data: list, max_length=160, vocab_size=1400):
        self.meta_data = meta_data
        self.enc_max_length = max_length
        self.dec_max_length = max_length
        self.vocab_size = vocab_size
        self.des = []
        self.diag_inputs = []
        self.dec_labels = []
        self.dec_labels_short = []
        self.des_st_masks = []
        self.des_attention_masks = []
        self.valid_lens = []
        self.diag_attention_masks = []
        for i in range(len(meta_data)):
            description = meta_data[i][1]
            diagnosis = meta_data[i][2]
            len1 = len(description)
            len2 = len(diagnosis)
            if len(description) > max_length:
                description = description[: max_length]
            self.des.append(description + [1] + [0] * (self.enc_max_length - len(description) - 2) + [2])

            self.des_attention_masks.append([1] * len(description) + [0] *
                                            (self.enc_max_length - len(description)))
            self.diag_inputs.append([2] + diagnosis + [1] + [0] * (self.dec_max_length - len(diagnosis) - 2))
            self.dec_labels.append(diagnosis + [1] + [0] * (self.dec_max_length - len(diagnosis) - 2) + [2])
            self.dec_labels_short.append(diagnosis)
            self.diag_attention_masks.append([1] * (len(diagnosis) + 1) + [0] *
                                             (self.dec_max_length - len(diagnosis) - 1))
            self.valid_lens.append(len(description))
```

在 PredDataset 类中：

●对于输入 ID（self.input_ids），描述序列末尾同样添加了必要数量的填充[0]以达到 self.enc_max_length。

●Attention mask（self.attention_masks）也是根据序列有效部分和填充部分分别标记为[1]和[0]。

```
class PredDataset(Dataset):
    def __init__(self, meta_data:list, encode_max_length=160, decode_max_length=96, vocab_size=1400):
        self.meta_data = meta_data
        self.enc_max_length = encode_max_length
        self.ids = []
        self.input_ids = []
        self.attention_masks = []
        self.labels = []
        for i in range(len(meta_data)):
            id = [int(meta_data[i][0])]
            description = meta_data[i][1]
            self.ids.append(id)
            self.input_ids.append(description + [0] * (self.enc_max_length - len(description)))
            self.attention_masks.append([1] * len(description) + [0] * (self.enc_max_length - len(description)))
```

最后将数据通过 DataLoader 加载成可供预训练的数据，pred_dataloader 和 build_pre_bart_dataloader 用于创建和配置 PyTorch 的 DataLoader。这些 DataLoader 对象是用来迭代地加载数据集，以便可以在机器学习模型训练或预测时按批次提供数据。

```
def pred_dataloader(args, meta_data, shuffle=True):
    data_dataset = PredDataset(meta_data, encode_max_length=args.encode_max_length,
                               decode_max_length=args.decode_max_length, vocab_size=args.vocab_size)
    data_loader = torch.utils.data.DataLoader(data_dataset,
                                              batch_size=args.batch_size,
                                              shuffle=True,
                                              pin_memory=False,
                                              num_workers=args.NUM_WORKERS)
    return data_loader
```

pred_dataloader 函数

●这个函数使用 PredDataset 类（可能是用户自定义的）来创建一个数据集实例。

●它使用传入的 meta_data（元数据）和 args（参数）来配置数据集。

●args 对象提供了如下配置信息：encode_max_length：编码器的最大序列长度。

decode_max_length：解码器的最大序列长度。

vocab_size：词汇表的大小。

batch_size：批次大小。

NUM_WORKERS：加载数据时使用的工作进程数。

●DataLoader 被配置为是否根据 shuffle 参数打乱数据，以及其他参数如 pin_memory 和 num_workers。

●函数返回一个 DataLoader 实例，该实例可以在训练或预测时使用。

```
def build_pre_bart_dataloader(args, meta_data, shuffle=True):
    data_dataset = BartDataset(meta_data, max_length=args.encode_max_length, vocab_size=args.vocab_size)
    data_loader = torch.utils.data.DataLoader(data_dataset,
                                              batch_size=args.batch_size,
                                              shuffle=shuffle,
                                              pin_memory=False,
                                              num_workers=args.NUM_WORKERS)
    return data_loader
```

build_pre_bart_dataloader 函数

●类似，这个函数使用 BartDataset 来创建一个数据集实例，这通常用于序列到序列的任务。

●它同样利用 meta_data 和 args 来配置数据集。

●args 提供配置信息，包括 encode_max_length 和 vocab_size。

●创建了一个 DataLoader 实例，配置了批次大小、是否打乱数据等参数。

●与 pred_dataloader 不同的是，这里 shuffle 参数是直接作为函数参数传入，而不是从 args 中获取。

●函数返回配置好的 DataLoader 实例。

```
# Data processing
data = process_data('./', 'data/train.csv', mode='train')
test_data = process_data('./', 'data/test.csv', mode='test')

# split train.csv to train_data and valid_data
n_split = int(len(data) * 0.8)
train_data = data[:n_split]
valid_data = data[n_split:]
print('Train data length: ', len(train_data))
print('Valid data length: ', len(valid_data))
print('Test data length: ', len(test_data))
# Load data
train_loader = build_pre_bart_dataloader(args, train_data, shuffle=True)
valid_loader = build_pre_bart_dataloader(args, valid_data, shuffle=True)
```

6、构建 seq2seq 模型：

Seq2Seq 模型是一种深度学习模型，主要用于将一个序列转换为另一个序列，在自然语言处理领域特别流行，用于如机器翻译、文本摘要、问答系统等任务，其核心特点是**能够处理变长的输入和输出序列**。

基本结构

Seq2Seq 模型通常包括两部分：一个编码器（Encoder）和一个解码器（Decoder）。

编码器：编码器的任务是读取并理解输入序列。它将输入的词、短语或句子转换成固定大小的内部状态或上下文（通常是一组数值），这个内部状态试图捕获输入序列的含义。

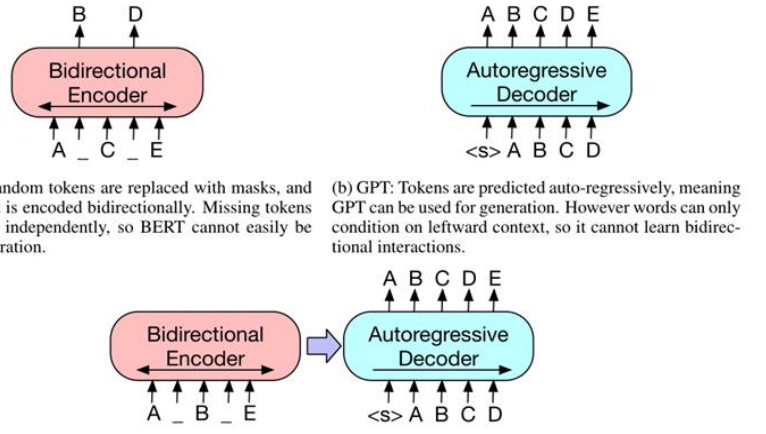
解码器：解码器则基于编码器的输出来生成目标序列。它逐步产生输出序列的每个元素，每一步的输出可能依赖于前一步的输出以及编码器的内部状态。

工作流程

在一个典型的 Seq2Seq 模型中，例如机器翻译，编码器读取源语言句子（比如英文），并生成一个或多个表示这个句子意义的内部状态。解码器使用这个状态来开始生成目标语言句子（比如中文），一步一步地输出单词，直到生成完整的翻译。

本次实验中选取的模型是由 Facebook AI 研究院开发的自然语言处理模型——BART，由一个 Transformer 编码器（类似于 BERT）和一个 Transformer 解码器（类似于 GPT）组成。编码器读取输入文本并编码信息，解码器则基于这些信息生成输出文本。

Bart 吸收了 Bert 的 bidirectional encoder 和 GPT 的 lefttoright decoder 各自的特点，建立在标准的 Seq2Seq Transformer model 的基础之上，这使得它比 Bert 更适合文本生成的场景；相比 GPT，也多了双向上下文语境信息。它有双向和自回归特性，即它的编码器是双向的，意味着在处理文本时考虑上下文中的所有词汇；而解码器是自回归的，即在生成文本时，每一步都依赖于前面的输出。在生成任务上获得进步的同时，它也可以在一些文本理解任务上取得很好的成绩。



(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.

(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.

(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

Figure 1: A schematic comparison of BART with BERT (Devlin et al., 2019) and GPT (Radford et al., 2018).

BART 虽然可以理解成是一个 BERT +GPT 的结构，但是相比于 BERT 单一的 noise 类型 (只有简单地用[MASK] token 进行替换这一种 noise)，BART 在 encoder 端尝试了多种 noise。因为 BERT 的这种简单替换导致的是 encoder 端的输入携带了有关序列结构的一些信息，比如序列长度，而这些信息在文本生成任务中一般是不会提供给模型的。BART 采用多种多样的 noise，意图是破坏掉这些有关序列结构的信息，防止模型去依赖这样的信息。

具体采用了以下几种 noise：

- **Token Masking:** 就是 BERT 的方法 ---- 随机将 token 替换成 [MASK]
- **Token Deletion:** 随机删去 token
- **Text Infilling:** 随机将一段连续的 token（称作 span）替换成一个 [MASK]，span 的长度服从$\lambda = 3$的泊松分布。注意 span 长度为 0 就相当于插入一个 [MASK]。
- **Sentence Permutation:** 将一个 document 的句子打乱
- **Document Rotation:** 从 document 序列中随机选择一个 token，然后使得该 token 作为 document 的开头

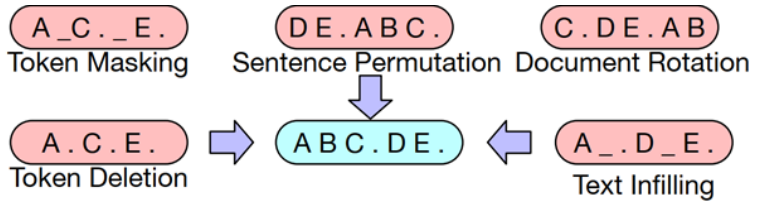
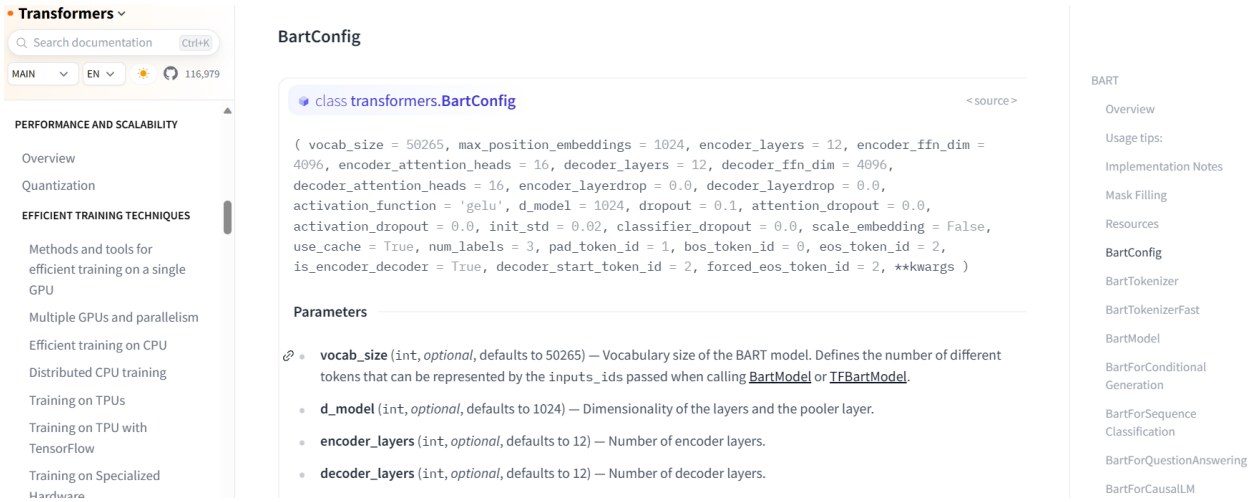


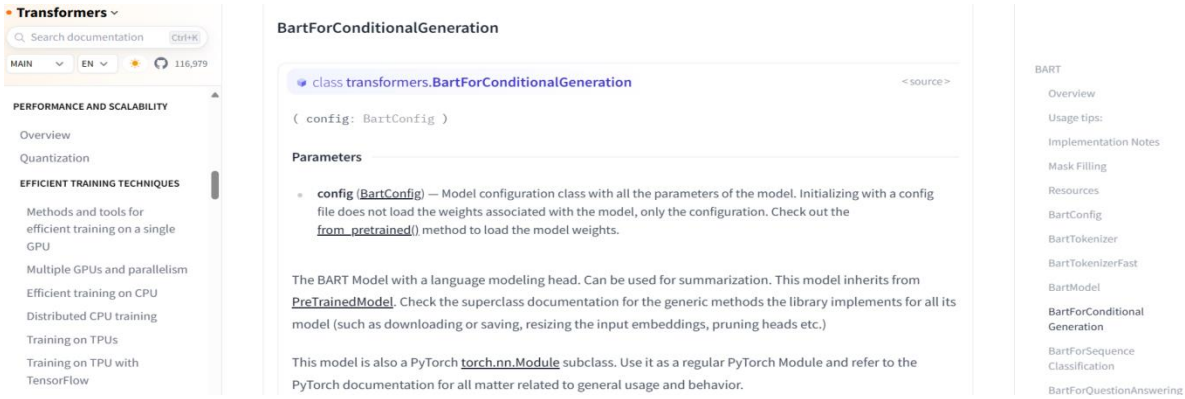
Figure 2: Transformations for noising the input that we experiment with. These transformations can be composed.

由于 BART 吸收了 BERT 双向自编码器和 GPT 自回归解码器的功能，所以它可以应用在多个下游任务上，如 Sequence Classification Task, Token Classification Task, Sequence Generation Task, Machine Translation 等等。本次实验的主题是文本摘要，所以相当于是一个 Sequence Generation Task.

在我们基于 BART 构建的 Seq2Seq 模型中，我们根据 Huggingface 提供的文档 BART [BART (huggingface.co)](#)使用 BartConfig 设定模型参数



并且使用 BartForConditionalGeneration 指定选取的预训练模型，再将训练数据放在预训练模型中训练参数。

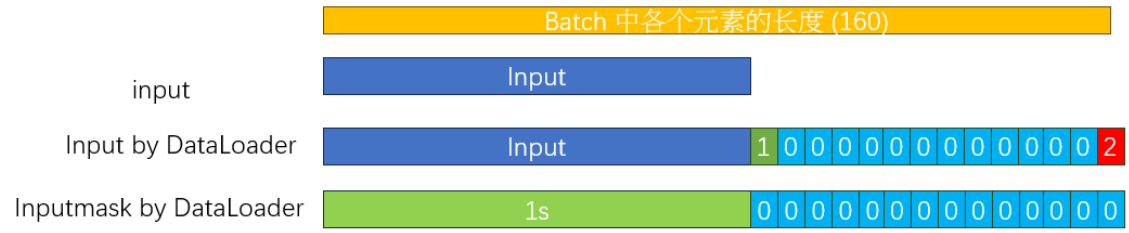


```python
class Seq2SeqModel(nn.Module):
    def __init__(self, model_name, vocab_size, label_smoothing=0.01):
        super(Seq2SeqModel, self).__init__()
        self.new_config = BartConfig(
            vocab_size=vocab_size,
            max_position_embeddings=1024,
            encoder_layers=6,
            encoder_ffn_dim=1024,
            encoder_attention_heads=16,
            decoder_layers=6,
            decoder_ffn_dim=1024,
            decoder_attention_heads=16,
            activation_function="gelu",
            d_model=256,
            dropout=0.5,
            attention_dropout=0.0,
            scale_embedding=False,
            use_cache=True,
            pad_token_id=0,
            bos_token_id=1,
            eos_token_id=2,
            mask_token_id=4,
            decoder_start_token_id=2,
            forced_eos_token_id=2,
        )
        # 从配置项中拉取预训练的 模型
        self.bart_model = BartForConditionalGeneration.from_pretrained(model_name, config=self.new_config,
                                                                       ignore_mismatched_sizes=True)
```

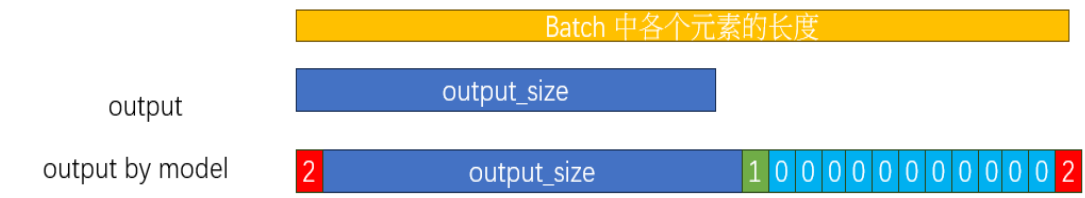
其中，对于几个关键的参数进行解释：

encoder_layers 以及 decoder_layers 定义为 6 是为了符合 bart-base 的 encoder 以及 decoder 层数。pad_token_id 是给每个句子加上 padding 让输入进的 batch 内的每一个句子形状都相同，但是对于 padding 部分不予以 attention 机制。bos_token_id 代表句子有效长度的结束，标记为 1。

eos_token_id 代表整个句子，包括 padding 部分结束，标记为 2.。这些在文件 my_dataset 中都已经实现，把这些参数让 bart 预训练模型知道，让模型“读懂”数据并且给出更好的 attention。



decoder_start_token_id 代表生成的每一个句子开头都是 2，forced_eos_token_id 代表生成的每一个句子结尾都是 2，这些是对于模型生成的语句，加上的一些字符。对于生成的句子，有效部分末尾都会有一个 bos_token_id, 即是代表句子内容的结束。



我们定义的 model_name 和 vocab_size 分别为 facebook/bart-base 以及 1300

```python
parser.add_argument('--model_name', default='facebook/bart-base', type=str)
parser.add_argument('--vocab_size', default=1300, help='')
```

选择的依据：

为何选择 bart-base?

因为选择更大的模型会导致过拟合，让预测准确率下降。同时加载预训练之后的模型权重会变得更慢，在后续训练与预测的时候也会更花时间与计算资源。对于一个训练+验证数据总共 18000 条的数据集，并且其中只包含 1300 个词，不需要使用太大的预训练模型。

为何选择 vocab_size 值为 1300?

因为通过将所有的训练数据放到一个 set 中，发现其取值是 9-1300 之间的整数，那么就干脆让模型生成 1300 以内的整数好了。

模型优化器设置

build_optimizer 的函数：根据给定参数构建和返回一个 PyTorch 优化器

●函数接受以下参数：

●optimizer_name: 一个字符串，指定要创建的优化器类型（如"AdamW"或其他）。

●pg: 参数组（parameter groups），通常是模型的参数。

●lr: 学习率。

●weight_decay: 权重衰减参数，用于正则化。

●构建 AdamW 优化器：

●如果 optimizer_name 是"AdamW"，函数将构建并返回一个 torch.optim.AdamW 优化器实例。

●AdamW 是 Adam 优化器的一种变体，它对权重衰减的处理方式与传统的 L2 正则化更接近。

●在这个分支中，优化器被配置为使用参数组 pg、betas 值为(0.9, 0.999)、权重衰减 weight_decay 和 eps 值为 1e-8（epsilon，防止除以零）。

●构建 SGD 优化器：

●如果 optimizer_name 不是"AdamW"，则默认构建一个随机梯度下降（SGD）优化器。

●SGD 优化器使用参数组 pg、学习率 lr、动量 momentum 值为 0.9 和权重衰减 weight_decay。

●返回优化器：最后，函数返回根据指定参数构建的优化器实例。

```python
import torch

def build_optimizer(optimizer_name, pg, lr, weight_decay):
    if optimizer_name == 'AdamW':
        optimizer = torch.optim.AdamW(pg,
                                      # lr=lr,
                                      betas=(0.9, 0.999),
                                      weight_decay=weight_decay,
                                      eps=1e-8)
    else:
        optimizer = torch.optim.SGD(pg,
                                    lr=lr,
                                    momentum=0.9,
                                    weight_decay=weight_decay)
    return optimizer
```

在主函数 main 中，我们会调用 build_optimizer 函数来创建模型的优化器。

```python
pg = [p for p in model.parameters() if p.requires_grad]
print('Model loaded successfully.')
# Define optimizer and scheduler
optimizer = build_optimizer(args.optimizer_name, pg, args.lr, args.weight_decay)
print('Optimizer configured successfully.')
```

模型学习率调度器设置

get_cos_schedule_with_warmup 和 build_scheduler 函数，用于创建和配置学习率调度器

函数 get_cos_schedule_with_warmup

这个函数创建一个自定义的学习率调度器，它结合了 warmup（预热）阶段和余弦衰减阶段。

参数:●

- ●optimizer: 优化器对象。
- ●num_warmup_steps: 预热步数，即在开始训练时逐渐增加学习率的步数。
- ●epochs: 总的训练轮数。
- ●lrf: 最终学习率因子，是最小学习率与初始学习率的比率。
- ●last_epoch: 最后一个 epoch 的索引。

●学习率调整逻辑:

- ●在 warmup 阶段，学习率从 0 线性增加到初始学习率。
- ●在 warmup 结束后，学习率按照余弦函数减小，直到达到 lrf 指定的最小值。

```python
def get_cos_schedule_with_warmup(optimizer, num_warmup_steps, epochs, lrf, last_epoch=-1):
    def lr_lambda(current_step: int):
        if current_step < num_warmup_steps:
            return float(current_step + 1) / float(max(1, num_warmup_steps))
        epoch = (epochs - num_warmup_steps) // 1
        step = (current_step - num_warmup_steps) % epoch
        return max(
            1E-6, (1 + math.cos(step * math.pi / epoch)) / 2 * (1 - lrf) + lrf  # cosine
        )
    return LambdaLR(optimizer, lr_lambda, last_epoch)
```

函数 build_scheduler

这个函数根据给定的参数创建一个学习率调度器。

●参数:

- ●lr_decay_func: 学习率衰减函数类型，如'cosine'或'linear'。
- ●optimizer: 优化器对象。
- ●num_warmup_steps: 预热步数。
- ●epochs: 总的训练轮数。
- ●lrf: 最终学习率因子。
- ●gamma: 用于某些衰减函数的额外参数。

●创建调度器:

- ●如果是'cosine'，则使用 get_cos_schedule_with_warmup 创建余弦衰减调度器。
- ●如果是'linear'，则使用 transformers.get_linear_schedule_with_warmup 创建线性衰减调度器。

```python
def build_scheduler(lr_decay_func, optimizer, num_warmup_steps, epochs, lrf, gamma=None):
    if lr_decay_func == 'cosine':
        scheduler = get_cos_schedule_with_warmup(optimizer,
                                                 num_warmup_steps=num_warmup_steps,
                                                 epochs=epochs,
                                                 lrf=lrf)
    elif lr_decay_func == 'linear':
        scheduler = get_linear_schedule_with_warmup(optimizer,
                                                    num_warmup_steps,
                                                    epochs)
    return scheduler
```

在 main 函数中调用 build_scheduler 函数根据指定的学习率衰减函数（args.lr_decay_func）、优化器对象（optimizer）、预热步数（args.num_warmup_steps）、训练轮数（args.epochs）和最终学习率因子（args.lrf）来创建学习率调度器。

```python
scheduler = build_scheduler(args.lr_decay_func, optimizer, args.num_warmup_steps, args.epochs, args.lrf)
```

7、训练和验证-观察 loss 值和 bleu score

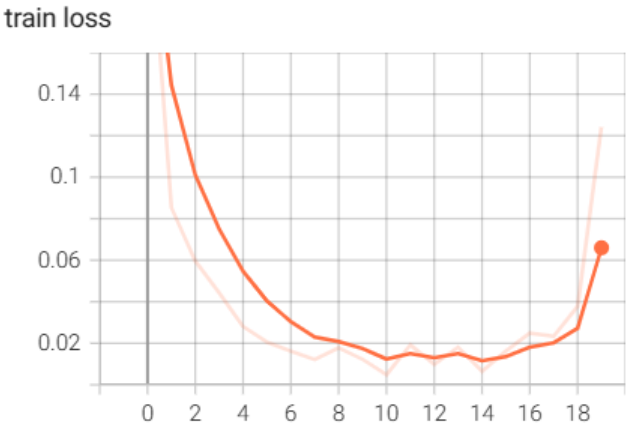
设定 num_epochs 为 20，定义 scheduler 以及 optimizer 进行混合精度训练，提升训练性能。每 5 个 epoch 保存一下模型参数，以便后面的验证和预测。通过 torch.utils.tensorboard 中的 SummaryWriter 工具将每次运行之后的结果保存在 runs/ 目录下，terminal run tensorboard --logdir='path/to/dir' 就可以查看对应训练中每个轮次的 loss 变化。

```
# Define optimizer and scheduler
optimizer = build_optimizer(args.optimizer_name, pg, args.lr, args.weight_decay)
print('Optimizer configured successfully.')
scheduler = build_scheduler(args.lr_decay_func, optimizer, args.num_warmup_steps, args.epochs, args.lrf)

scaler = torch.cuda.amp.GradScaler()
# 混合精度训练
for epoch in range(args.epochs):
    cur_lr = optimizer.state_dict()['param_groups'][0]['lr']
    print('lr:', cur_lr)
    print('weight decay:', optimizer.state_dict()['param_groups'][0]['weight_decay'])
    train_loss = train_one_epoch(train_loader, model, optimizer, epoch, device=device, is_adversial=False, scaler=scaler)
    valid_loss = valid(valid_loader, model, device, epoch, args.num_beams, args.file_valid)

    tb_writer.add_scalar('train loss', train_loss, epoch)
    tb_writer.add_scalar('valid loss', valid_loss, epoch)
    tb_writer.add_scalar('lr', cur_lr, epoch)
    if (epoch + 1) % 5 == 0:
        save_model(model, model_dir, args.model_name, epoch)
    scheduler.step()
```

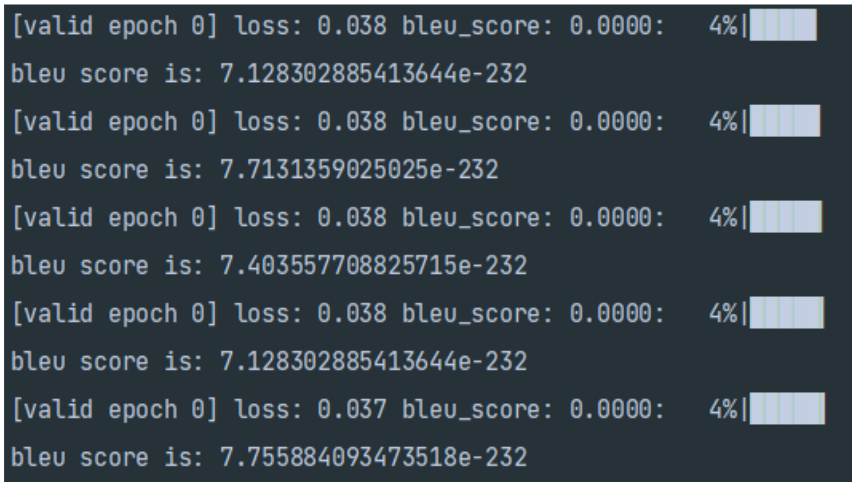
从终端查看训练过程中 loss 的变化，发现在第六个 epoch 以后 loss 值就基本不太变化，降到一个比较低的数值。

```
[train epoch 3] loss: 0.044 lm loss: 0.008 mlm loss: 0.362: 100%|
lr: 0.00025
weight decay: 0.0
[train epoch 4] loss: 0.028 lm loss: 0.008 mlm loss: 0.202: 100%|
lr: 0.0003
weight decay: 0.0
[train epoch 5] loss: 0.021 lm loss: 0.009 mlm loss: 0.114: 100%|
lr: 0.00035
weight decay: 0.0
[train epoch 6] loss: 0.016 lm loss: 0.009 mlm loss: 0.073: 100%|
lr: 0.0004
weight decay: 0.0
[train epoch 7] loss: 0.012 lm loss: 0.007 mlm loss: 0.049: 100%|
lr: 0.00045000000000000004
weight decay: 0.0
[train epoch 8] loss: 0.018 lm loss: 0.011 mlm loss: 0.067: 100%|
lr: 0.0005
weight decay: 0.0
[train epoch 9] loss: 0.012 lm loss: 0.007 mlm loss: 0.048: 100%|
lr: 0.00055
weight decay: 0.0
[train epoch 10] loss: 0.005 lm loss: 0.003 mlm loss: 0.021: 100%|
```

在本地查看 tensorboard 上面 summarywriter 给出的信息，在 localhost 的 6006 端口：



在这里我将 lr 每过一个 epoch 就加上 5e-5 是为了防止训练 loss 一直下降，因为如果 lr 一直保持在一个值就会让训练 loss 一直下降，我们不知道什么时候是过拟合。可以看到，9-14 个 epoch 附近训练的模型是比较合适的。
使用 load_state_dict 方法加载预训练后的模型权重以后，在验证集上进行 bleu 值的计算：



这里介绍一下我采用的评估指标的计算方法-BLEU：
BLEU 是一个广泛使用的评估自然语言处理系统输出质量的指标。它通过计算系统输出（候选译文）与一个或多个参考翻译之间的重叠度来工作。

$$BLEU = BP \times \exp\left(\sum_{n=1}^{N} W_n \times \log P_n\right)$$

它的计算步骤如下：
●N-gram 匹配：
●BLEU 考虑的是候选翻译与参考翻译间的 n-gram 重叠情况，n-gram 是指长度为 n 的连续词的序列。
●对于每个 n-gram 长度（通常从 1 到 4），计算候选翻译中的 n-gram 出现次数，并且检查这些 n-gram 在参考翻译中出现了多少次。
●修剪（Clipping）：
●为了避免对候选翻译中重复出现的 n-gram 给予过多的奖励，BLEU 使用了修剪技术。
●如果某个 n-gram 在候选翻译中出现次数超过了在任何一个参考翻译中的最大出现次数，则这个 n-gram 的计数会被修剪到该最大出现次数。
●计算精确度分数（Precision Scores）：
●对每个 n-gram 长度，计算修剪后的匹配数与候选翻译中该长度 n-gram 总数的比率，称为精确度分数。
●例如，对于 1-gram（单词）精确度，如果候选翻译中有 10 个词，而其中 5 个词在参考翻译中出现过，那么 1-gram 精确度就是 50%。
●计算几何平均分：
●BLEU 分数是所有 n-gram 精确度分数的几何平均值，这意味着它们相乘然后取 n 次根（n 是 n-gram 的长度）。
●短句惩罚（Brevity Penalty, BP）：
●为了惩罚过短的候选翻译，BLEU 引入了短句惩罚。

●如果候选翻译的长度小于参考翻译的长度，BLEU 会计算一个惩罚值，减少总分数。
●惩罚值是指数形式的，其中指数是参考翻译长度与候选翻译长度的比率。

$$BP = \begin{cases} 1 & lc > \mathrm{lr} \\ \exp(1 - \mathrm{lr}/lc) & lc \leq \mathrm{lr} \end{cases}$$

$$lc = \text{机器译文的长度}$$

$$lr = \text{最短的参考翻译句子的长度}$$

●最终 BLEU 得分：
将加权平均的精确度乘以短句惩罚（BP），得到最终的 BLEU 得分。当存在多个参考答案和多个候选答案时，通常采用的计算方式是：
●对每个候选答案，计算其与所有参考答案的 BLEU 得分。
●对每个参考答案和候选答案的配对计算 BLEU 分数。
●将每个候选答案的分数平均化，得到单个候选答案的平均 BLEU 分数。
●最后，将所有候选答案的平均 BLEU 分数再次平均，得到最终的平均 BLEU 分数。
这种方法可以确保评价指标在有多个参考翻译时更加公平和稳健。不过，BLEU 得分有其局限性，比如它不能很好地评估翻译的流畅性和语法正确性，只侧重于 n-gram 的字面匹配。

8、遇到的问题与自己的思考

通过观察，我们可以发现 bleu score 非常低。一开始我还以为是在 batch 内部使用 corpus bleu score 的问题，不过后来查阅资料发现，在存在多个参考答案和多个候选答案的时候，计算机制使用双重循环对每个候选答案和参考答案进行配对，并计算每个配对的 BLEU 分数。然后，将每个候选答案的 BLEU 分数取平均，得到最终的平均 BLEU 分数。

后来想到，预训练模型的选取不是瞎选的，主要还是需要 tokenizer 匹配。因为中文的数据集不能用英文的预训练模型，反之亦然。因为它们的 tokenizer 要么是无法对应到相应的 token，要么是有对应的 token ，但是没有语义可以学习，学习效果很差。而输入数字和输出数字也是这样，首先我不能按照官方文档所给的那样，使用 Bart 的 tokenizer，因为根本没办法进行 tokenizer，1300 个数字能够都好好地进行 tokenize 就基本不可能。只能通过数字来进行 seq2seq，但是这样就完全学不到语义，甚至是乱学。

那么为什么 loss 值不高，0.03 的水平，bleu score 还那么低呢？
因为 loss 值是关于 ground truth 的，输出的数据可能是一条一条像模像样的，在那 1300 个词中产生，长度也差不多对应，总体符合分布的情况。但是对于 bleu score 这个计算指标，reference 和 generate 的句子可能有几个 1-gram 的对应，但是更高 gram 的对应基本没有的情况下，bleu score 的值就会是很低的。就相当于在一个 1300 个数里面盲猜。看到几乎所有的句子 bleu score 都是 7.几 e-232 ，可以知道这个数就是 0。所以盲猜得到的结果 bleu score 就是 0 也不奇怪了。

所以得出结论：
在没有任何上下文语义的数据，或者用错了 tokenizer 的数据，在预训练模型上面训练，即使优秀如 BART，也依然很难学习到什么。我认为我的结论还是比较合理的。既然如此，也没有什么 fine-tuning 的必要了，因为我将 dropout 值分别设置为 0.1, 0.2, 0.5, 训了几个 epoch 之后，bleu score 的表现依旧相当地稳定。通过简单概率论的论证，可以知道，本次

数据不适合在预训练语言模型上进行学习和后续的微调。

9、遇到的问题及解决方法：

1、torchtext 的安装

直接输入 pip install torchtext 会使自己的 pytorch 自动更换版本，成为 cpu 版本，因为 torchtext 的版本严格对应 torch 的版本，只能卸载重装

```
In[3]: print(torchtext.__version__)
0.15.2
In[4]: import torch
In[5]: print(torch.__version__)
2.0.1+cpu
```

解决方法：对于 torch 版本，输入 pip install torchtext==0.xx.y 安装，在我的 torch 版本 1.13.0 中，xx = 13 + 1, y = 0. 所以输入命令 pip install torchtext==0.14.0 安装成功，没有报错。

输入 print(torch.__version) 与 print(torchtext.__version) 之后，都显示正常了。

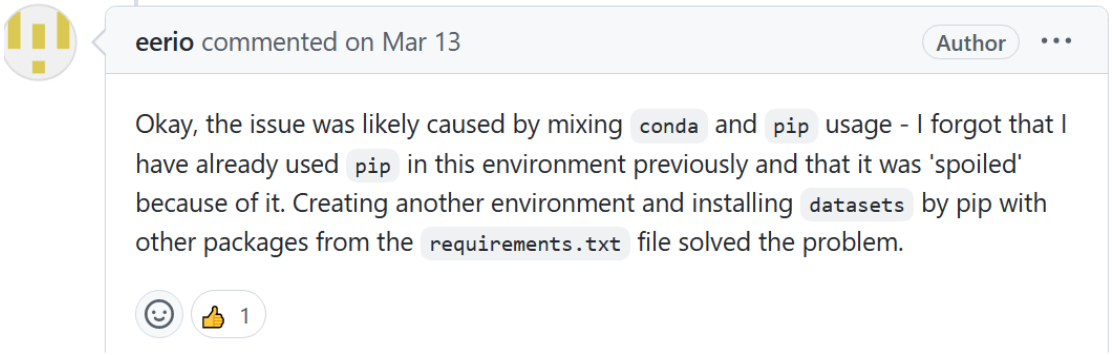
```
In[3]: print(torch.__version__)
1.13.0+cu116
In[4]: import torchtext
In[5]: print(torchtext.__version__)
0.14.0
```

2、使用 huggingface 模块，安装遇到的问题

我在 pycharm 的 UI 上面直接点了 install package datasets, 随后在 console 中输入 from datasets import load_dataset 之后，出现报错 ImportError & ModuleNotFoundError. 由于自己的报错截图已经无法找到，所以从 huggingface 官方项目的 github 仓库中找到相关 issue 的报错信息，大致如下：

```
$ python3
Python 3.8.15 (default, Nov 24 2022, 15:19:38)
[GCC 11.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import datasets
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    from .arrow_dataset import Dataset, concatenate_datasets
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    from .arrow_reader import ArrowReader
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/datasets/
    import pyarrow.parquet as pq
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/pyarrow/p
    from .core import *
  File "/home/jack/.conda/envs/jack_zpp/lib/python3.8/site-packages/pyarrow/p
    from pyarrow._parquet import (ParquetReader, Statistics,  # noqa
ImportError: cannot import name 'FileEncryptionProperties' from 'pyarrow._par
```

在此 issue 的时间线中，提供的方法：



在这之后，我通过 conda remove datasets 卸载之后，重新使用 pip 安装，即输入命令 pip install datasets 。不过在 console 中输入 from datasets import load_dataset 之后，出现报错 DLL Not Found，而对于这一类问题，所给出的方法是卸载重装。于是我又卸载了之后重装，结果还是不行。

解决方法：后来受到文章 https://cloud.tencent.com/developer/article/1748638 启发，运行 python -m venv venv/ 创建了一个虚拟环境，然后 venv/Scripts/activate 以后，进入 /venv/Scripts 目录，输入 python 命令进入交互页面。在此处输入 import datasets from datasets import load_dataset 命令，没有报错。所以这一类错误需要在项目目录下自行新建一个 venv 虚拟环境。

```
WARNING: You are using pip version 22.0.4; however, version 23.1.2 is available.
You should consider upgrading via the 'E:\2023 Spring\ContemporaryAI\labs\Lab4\venv\Scripts
(venv) PS E:\2023 Spring\ContemporaryAI\labs\Lab4> cd venv/Scripts
(venv) PS E:\2023 Spring\ContemporaryAI\labs\Lab4\venv\Scripts> python
Python 3.9.16 (main, Jan 11 2023, 16:16:36) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import datasets
>>> from datasets import load_dataset
```

3、使用 bleu_score 时候遇到的问题

我将 model.generate 的数字直接与 diagnosis 的数字作对比，发现了如下报错：

```
  File "/usr/local/miniconda3/lib/python3.8/site-packages/nltk/translate/bleu_score.py", line 210,
    p_i = modified_precision(references, hypothesis, i)
  File "/usr/local/miniconda3/lib/python3.8/site-packages/nltk/translate/bleu_score.py", line 353,
    Counter(ngrams(reference, n)) if len(reference) >= n else Counter()
TypeError: object of type 'int' has no len()
```

报错的意思一开始没有读懂，后来发现我的输入和输出都是 int 类型的数字。

解决方法：把标准输出与候选输出的内容全部由整数转换为字符串，可以计算 bleu_score。

心得与体会：

本次实验我利用 Facebook 预训练的 BART 模型来对 train.csv 数据进行预训练，使得它学习到数据上一些重要的权重，经过预训练的模型可以用于其他数据集相关的预测工作。我也借此机会学习了 BART 模型的更多细节。实验得出的 blue 值不令人满意，对此我思考了这个现象发生的原因：1、模型与 Tokenizer 不匹配：使用了与数据集语言不匹配的预训练模

型和 tokenizer，导致无法正确处理和理解文本。2、数据缺乏语义信息：数据主要由数字组成，没有实际的语义内容，使得模型难以从中学习有意义的信息。3、对高阶 n-gram 匹配敏感：尽管生成的文本在一些 1-gram 上可能与参考文本匹配，但在高阶 n-gram 匹配上几乎没有，这直接导致 BLEU 分数非常低。4、低 Loss 与低 BLEU 分数之间的矛盾：尽管模型在训练过程中显示出低 Loss 值，但这并不意味着它在生成与参考文本紧密相关的输出方面表现良好，因此 BLEU 分数仍然很低。

在实验的过程中除了完成老师的任务，我还遇到环境配置上一些乱七八糟的问题，最后都一一解决了。这让我感受到一个项目的完成，不论是大是小，都会遇见一些对实验结果无影响但是阻碍实验进程的设备问题，虽然棘手但是需要从很多来源寻找解决办法，办法总是有的，我们需要一直有耐心。

References

[1] Xu H, Zhengyan Z, Ning D, et al. Pre-Trained Models: Past, Present and Future[J]. arXiv preprintarXiv:2106.07139, 2021.

[2] Mike Lewis, Yinhan Liu, Naman Goyal, et al. BART: Denoising Sequence-to-Sequence Pretraining for Natural Language Generation, Translation, and Comprehension arXiv preprint arXiv:1910.13461, 2019.

预训练模型的过去、现在和将来之一-预训练模型和训练模型 (51cto.com)

Pipenv vs Virtualenv vs Conda environment - 知乎 (zhihu.com)

详解 Python 虚拟环境的原理及使用-腾讯云开发者社区-腾讯云 (tencent.com)

浅谈 BLEU 评分 - Jiayue Cai's Blog (coladrill.github.io)

BART (huggingface.co)