姓名：吕奇澹
学号：SA23006171

实验要求

实现Transformer模型以进行英译中机器翻译，其中模型的层数、维度大小以及训练数据的规模均可根据个人电脑的性能进行相应调整。采用BLEU值作为模型性能的评估指标。

实验过程

数据处理

数据集：
本研究采用WMT 2020新闻翻译任务（NEWS）中的中英文对（Chinese-English, ZH-EN）数据集。具体使用以下两个语料库数据集：
新闻评论v15（News Commentary v15），可从 https://data.statmt.org/news-commentary/v15/training/ 获取，包含中英文对（en-zh pair）共320,712条。其中，80%划分为训练集，10%划分为验证集，剩余10%用作测试集。
反向翻译新闻（Back-translated news），可从 https://data.statmt.org/wmt20/translation-task/back-translation/zh-en/ 下载，共含19,763,867条数据。从中筛选出2,000,000条数据作为训练集，另外各分配400,000条数据作为验证集和测试集。
对应数据处理的代码为：

```
import os
from sklearn.model_selection import train_test_split

def load_dataset(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()
        en_sentences = []
        zh_sentences = []
        for line in lines:
            # 跳过空行或不含制表符的行（即不包含中英文对的行）
            if '\t' not in line.strip():
                continue
            en, zh = line.strip().split('\t', 1)  # 最多分割成两部分
            if not en or not zh:  # 如果英文或中文部分为空，跳过
                continue
            en_sentences.append(en + '\n')  # 加入换行符，方便后续文件写入
            zh_sentences.append(zh + '\n')
    return en_sentences, zh_sentences

def split_dataset(en_sentences, zh_sentences, test_size=0.2):
    # 先分出80%的训练集，20%的临时测试集
    en_train, en_temp, zh_train, zh_temp = train_test_split(en_sentences,
zh_sentences, test_size=test_size, random_state=42)
    # 再将临时测试集一分为二，得到10%的验证集和10%的测试集
    en_val, en_test, zh_val, zh_test = train_test_split(en_temp, zh_temp,
test_size=0.5, random_state=42)
```

```
    return en_train, zh_train, en_val, zh_val, en_test, zh_test

def save_datasets(en_train, zh_train, en_val, zh_val, en_test, zh_test,
dataset_dir=''):
    # if not os.path.exists(dataset_dir):
    #     os.makedirs(dataset_dir)

    with open(os.path.join(dataset_dir, 'train.en'), 'w', encoding='utf-8') as file:
        file.writelines(en_train)
    with open(os.path.join(dataset_dir, 'train.zh'), 'w', encoding='utf-8') as file:
        file.writelines(zh_train)

    with open(os.path.join(dataset_dir, 'val.en'), 'w', encoding='utf-8') as file:
        file.writelines(en_val)
    with open(os.path.join(dataset_dir, 'val.zh'), 'w', encoding='utf-8') as file:
        file.writelines(zh_val)

    with open(os.path.join(dataset_dir, 'test.en'), 'w', encoding='utf-8') as file:
        file.writelines(en_test)
    with open(os.path.join(dataset_dir, 'test.zh'), 'w', encoding='utf-8') as file:
        file.writelines(zh_test)

# 加载数据集
file_path = 'news-commentary-v15.en-zh.tsv'
en_sentences, zh_sentences = load_dataset(file_path)

# 划分数据集
en_train, zh_train, en_val, zh_val, en_test, zh_test = split_dataset(en_sentences,
zh_sentences)

# 保存数据集
save_datasets(en_train, zh_train, en_val, zh_val, en_test, zh_test)
```

```
import random

# 假设文件路径
en_file_path = 'news.en'
zh_file_path = 'news.translatedto.zh'

# 读取数据
with open(en_file_path, 'r', encoding='utf-8') as en_file, open(zh_file_path, 'r',
encoding='utf-8') as zh_file:
    en_lines = en_file.readlines()
    zh_lines = zh_file.readlines()

# 检查数据长度是否相同
assert len(en_lines) == len(zh_lines), "The number of sentences in both files should
be the same."

# 将数据合并为元组列表
```

```
combined_data = list(zip(en_lines, zh_lines))

# 随机打乱数据
# random.shuffle(combined_data)

# 定义数据集的大小
train_size = 2000000
valid_test_size = 400000  # 验证集和测试集的大小相同

# 分割数据集
train_data = combined_data[:train_size]
valid_data = combined_data[train_size:train_size + valid_test_size]
test_data = combined_data[train_size + valid_test_size:train_size + 2 *
valid_test_size]

# 定义保存数据的函数
def save_data(dataset, en_filename, zh_filename):
    with open(en_filename, 'w', encoding='utf-8') as en_file, open(zh_filename, 'w',
encoding='utf-8') as zh_file:
        for en_line, zh_line in dataset:
            en_file.write(en_line)
            zh_file.write(zh_line)

# 保存数据集
save_data(train_data, '2train.en', '2train.zh')
save_data(valid_data, '2valid.en', '2valid.zh')
save_data(test_data, '2test.en', '2test.zh')

print("Datasets have been successfully split and saved.")
```

```
def is_garbage(line):
    # 假设乱码由重复的引号字符组成，我们可以检查是否有大量连续重复的字符
    return '"' in line or "'" in line or "’" in line or "“" in line or "-" in line
or "—" in line or "”" in line or "-" in line or ", " in line or "： " in line

def clean_data_sync(en_lines, zh_lines, is_garbage_func):
    # 确保两个列表的长度相同
    assert len(en_lines) == len(zh_lines), "Files do not have the same number of
lines."

    # 清除两个列表中的空行和乱码行，确保它们的索引保持同步
    cleaned_en_lines = []
    cleaned_zh_lines = []
    for en_line, zh_line in zip(en_lines, zh_lines):
        if en_line.strip() and zh_line.strip() and not is_garbage_func(en_line) and
not is_garbage_func(zh_line):
            cleaned_en_lines.append(en_line)
            cleaned_zh_lines.append(zh_line)

    return cleaned_en_lines, cleaned_zh_lines
```

```
# 文件路径
en_file_path = '2test.en'
zh_file_path = '2test.zh'

# 读取文件
with open(en_file_path, 'r', encoding='utf-8') as f_en:
    en_lines = f_en.readlines()

with open(zh_file_path, 'r', encoding='utf-8') as f_zh:
    zh_lines = f_zh.readlines()

# 清洗数据
clean_en, clean_zh = clean_data_sync(en_lines, zh_lines, is_garbage)

# 检查清洗后的数据行数是否相同，此时它们应该是相同的
assert len(clean_en) == len(clean_zh), "The number of cleaned lines does not match
after cleaning."

# 写入清洗后的数据到新文件
cleaned_en_file_path = f'cleaned_{en_file_path}'
cleaned_zh_file_path = f'cleaned_{zh_file_path}'

with open(cleaned_en_file_path, 'w', encoding='utf-8') as f_clean_en:
    f_clean_en.writelines(clean_en)

with open(cleaned_zh_file_path, 'w', encoding='utf-8') as f_clean_zh:
    f_clean_zh.writelines(clean_zh)

# 输出清洗后的数据行数和新文件的路径
print(len(clean_en), cleaned_en_file_path, cleaned_zh_file_path)
```

此时我们将下载下来的数据集可以处理为对应的数据集、验证集和测试集
（train.zh,train.en,valid.zh,valid.en,test.zh,test.en)
然后再通过config文件读入并设置超参数

```
import os
import torch
from utils.log_helper import logger_init
import logging
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
class Config():
    """
    基于Transformer架构的类Translation模型配置类
    """

    def __init__(self):
        #    数据集设置相关配置
```

```
        self.project_dir =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
        self.dataset_dir = os.path.join(self.project_dir, 'data')
        self.train_corpus_file_paths = [os.path.join(self.dataset_dir,
'cleaned_2train.zh'),  # 训练时编码器的输入
                                        os.path.join(self.dataset_dir,
'cleaned_2train.en')]  # 训练时解码器的输入
        self.val_corpus_file_paths = [os.path.join(self.dataset_dir,
'cleaned_2valid.zh'),  # 验证时编码器的输入
                                      os.path.join(self.dataset_dir,
'cleaned_2valid.en')]  # 验证时解码器的输入
        self.test_corpus_file_paths = [os.path.join(self.dataset_dir,
'cleaned_2test.zh'),
                                       os.path.join(self.dataset_dir,
'cleaned_2test.en')]
        self.min_freq = 1  # 在构建词表的过程中滤掉词（字）频小于min_freq的词（字）

        #  模型相关配置
        self.batch_size = 128
        self.d_model = 512
        self.num_head = 8
        self.num_encoder_layers = 6
        self.num_decoder_layers = 6
        self.dim_feedforward = 512
        self.dropout = 0.1
        self.beta1 = 0.9
        self.beta2 = 0.98
        self.epsilon = 10e-9
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.epochs = 10
        self.model_save_dir = os.path.join(self.project_dir, 'cache')
        if not os.path.exists(self.model_save_dir):
            os.makedirs(self.model_save_dir)
        # 日志相关
        logger_init(log_file_name='log_train',
                    log_level=logging.INFO,
                    log_dir=self.model_save_dir)
```

模型结构

构建MyTransformer模型

```
import torch.nn as nn
import torch
import math


class PositionalEncoding(nn.Module):
    r"""Inject some information about the relative or absolute position of the
tokens
        in the sequence. The positional encodings have the same dimension as
```

```
        the embeddings, so that the two can be summed. Here, we use sine and cosine
        functions of different frequencies.
    .. math::
        \text{PosEncoder}(pos, 2i) = sin(pos/10000^(2i/d_model))
        \text{PosEncoder}(pos, 2i+1) = cos(pos/10000^(2i/d_model))
        \text{where pos is the word position and i is the embed idx)
    Args:
        d_model: the embed dim (required).
        dropout: the dropout value (default=0.1).
        max_len: the max. length of the incoming sequence (default=5000).
    Examples:
        #>>> pos_encoder = PositionalEncoding(d_model)
    """

    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)  # [max_len, d_model]
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  #
[max_len, 1]
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))  # [d_model/2]
        pe[:, 0::2] = torch.sin(position * div_term)  # [max_len, d_model/2]
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(1)
        self.register_buffer('pe', pe)

    def forward(self, x):  # [x_len, batch_size, d_model]
        """
        :param x: [x_len, batch_size, emb_size]
        :return: [x_len, batch_size, emb_size]
        """
        x = x + self.pe[:x.size(0), :]  # [src_len,batch_size, d_model] + [src_len,
1, d_model]
        return self.dropout(x)  # [src_len,batch_size, d_model]


class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size

    """
        :param tokens: shape : [len, batch_size]
        :return: shape: [len, batch_size, emb_size]
        """

    def forward(self, tokens):
        return self.embedding(tokens) * math.sqrt(self.emb_size)
```

```
from torch.nn.init import xavier_uniform_
import torch.nn.functional as F
from torch.nn import Parameter
import torch.nn as nn
import copy
import torch

is_print_shape = True


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

        """
        :param d_model:  d_k = d_v = d_model/nhead = 64, 模型中向量的维度，论文默认值为
512
        :param nhead:                多头注意力机制中多头的数量，论文默认为值 8
        :param num_encoder_layers:   encoder堆叠的数量，也就是论文中的N，论文默认值为6
        :param num_decoder_layers:   decoder堆叠的数量，也就是论文中的N，论文默认值为6
        :param dim_feedforward:      全连接中向量的维度，论文默认值为 2048
        :param dropout:              丢弃率，论文中的默认值为 0.1
        """

        #  ================ 编码部分 =====================
        encoder_layer = MyTransformerEncoderLayer(d_model, nhead, dim_feedforward,
dropout)
        encoder_norm = nn.LayerNorm(d_model)
        self.encoder = MyTransformerEncoder(encoder_layer, num_encoder_layers,
encoder_norm)

        # ================ 解码部分 =====================
        decoder_layer = MyTransformerDecoderLayer(d_model, nhead, dim_feedforward,
dropout)
        decoder_norm = nn.LayerNorm(d_model)
        self.decoder = MyTransformerDecoder(decoder_layer, num_decoder_layers,
decoder_norm)

        self._reset_parameters()

        self.d_model = d_model
        self.nhead = nhead

    def _reset_parameters(self):
        r"""Initiate parameters in the transformer model."""
        """
        初始化
        """
        for p in self.parameters():
            if p.dim() > 1:
```

```
                xavier_uniform_(p)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None,
                memory_mask=None, src_key_padding_mask=None,
                tgt_key_padding_mask=None, memory_key_padding_mask=None):
        """
        :param src:   [src_len,batch_size,embed_dim]
        :param tgt:  [tgt_len, batch_size, embed_dim]
        :param src_mask:  None
        :param tgt_mask:  [tgt_len, tgt_len]
        :param memory_mask: None
        :param src_key_padding_mask: [batch_size, src_len]
        :param tgt_key_padding_mask: [batch_size, tgt_len]
        :param memory_key_padding_mask:  [batch_size, src_len]
        :return: [tgt_len, batch_size, num_heads * kdim] <==>
[tgt_len,batch_size,embed_dim]
        """
        memory = self.encoder(src, mask=src_mask,
src_key_padding_mask=src_key_padding_mask)
        # [src_len, batch_size, num_heads * kdim] <==>
[src_len,batch_size,embed_dim]
        output = self.decoder(tgt=tgt, memory=memory, tgt_mask=tgt_mask,
memory_mask=memory_mask,
                              tgt_key_padding_mask=tgt_key_padding_mask,
                              memory_key_padding_mask=memory_key_padding_mask)
        return output  # [tgt_len, batch_size, num_heads * kdim] <==>
[tgt_len,batch_size,embed_dim]

    def generate_square_subsequent_mask(self, sz):
        r"""Generate a square mask for the sequence. The masked positions are filled
with float('-inf').
            Unmasked positions are filled with float(0.0).
        """
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask
== 1, float(0.0))
        return mask  # [sz,sz]


class MyTransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
        super(MyTransformerEncoderLayer, self).__init__()
        """
        :param d_model:         d_k = d_v = d_model/nhead = 64, 模型中向量的维度，论文默
认值为 512
        :param nhead:           多头注意力机制中多头的数量，论文默认为值 8
        :param dim_feedforward: 全连接中向量的维度，论文默认值为 2048
        :param dropout:         丢弃率，论文中的默认值为 0.1


        """
        self.self_attn = MyMultiheadAttention(d_model, nhead, dropout=dropout)
```

```
        # Implementation of Feedforward model
        self.dropout1 = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(d_model)

        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.activation = nn.ReLU()

        self.dropout2 = nn.Dropout(dropout)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, src, src_mask=None, src_key_padding_mask=None):
        """
        :param src: 编码部分的输入，形状为 [src_len,batch_size, embed_dim]
        :param src_mask:  None
        :param src_key_padding_mask:  编码部分输入的padding情况，形状为 [batch_size,
src_len]
        :return
        """
        src2 = self.self_attn(src, src, src, attn_mask=src_mask,
                              key_padding_mask=src_key_padding_mask, )[0]  # 计算多头
注意力
        # src2: [src_len,batch_size,num_heads*kdim] num_heads*kdim = embed_dim
        src = src + self.dropout1(src2)  # 残差连接
        src = self.norm1(src)  # [src_len,batch_size,num_heads*kdim]

        src2 = self.activation(self.linear1(src))  #
[src_len,batch_size,dim_feedforward]
        src2 = self.linear2(self.dropout(src2))  #
[src_len,batch_size,num_heads*kdim]
        src = src + self.dropout2(src2)
        src = self.norm2(src)
        return src  # [src_len, batch_size, num_heads * kdim] <==>
[src_len,batch_size,embed_dim]


class MyTransformerEncoder(nn.Module):
    def __init__(self, encoder_layer, num_layers, norm=None):
        super(MyTransformerEncoder, self).__init__()
        """
        encoder_layer: 就是包含有多头注意力机制的一个编码层
        num_layers: 克隆得到多个encoder layers 论文中默认为6
        norm: 归一化层

        """
        self.layers = _get_clones(encoder_layer, num_layers)  # 克隆得到多个encoder
layers 论文中默认为6
        self.num_layers = num_layers
        self.norm = norm
```

```
    def forward(self, src, mask=None, src_key_padding_mask=None):
        """
        :param src: 编码部分的输入，形状为 [src_len,batch_size, embed_dim]
        :param mask:  编码部分输入的padding情况，形状为 [batch_size, src_len]
        :return:# [src_len, batch_size, num_heads * kdim] <==>
[src_len,batch_size,embed_dim]
        """
        output = src
        for mod in self.layers:
            output = mod(output, src_mask=mask,
                         src_key_padding_mask=src_key_padding_mask)  # 多个encoder
layers层堆叠后的前向传播过程
        if self.norm is not None:
            output = self.norm(output)
        return output  # [src_len, batch_size, num_heads * kdim] <==>
[src_len,batch_size,embed_dim]


def _get_clones(module, N):
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])


class MyTransformerDecoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
        super(MyTransformerDecoderLayer, self).__init__()
        """
        :param d_model:         d_k = d_v = d_model/nhead = 64, 模型中向量的维度，论文默
认值为 512
        :param nhead:           多头注意力机制中多头的数量，论文默认为值 8
        :param dim_feedforward: 全连接中向量的维度，论文默认值为 2048
        :param dropout:         丢弃率，论文中的默认值为 0.1
        """
        self.self_attn = MyMultiheadAttention(embed_dim=d_model, num_heads=nhead,
dropout=dropout)
        # 解码部分输入序列之间的多头注意力（也就是论文结构图中的Masked Multi-head attention)
        self.multihead_attn = MyMultiheadAttention(embed_dim=d_model,
num_heads=nhead, dropout=dropout)
        # 编码部分输出（memory）和解码部分之间的多头注意力机制。
        # Implementation of Feedforward model

        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

        self.activation = nn.ReLU()
```

```
    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None,
tgt_key_padding_mask=None,
                memory_key_padding_mask=None):
        """
        :param tgt:  解码部分的输入，形状为 [tgt_len,batch_size, embed_dim]
        :param memory: 编码部分的输出（memory）, [src_len,batch_size,embed_dim]
        :param tgt_mask: 注意力Mask输入，用于掩盖当前position之后的信息, [tgt_len,
tgt_len]
        :param memory_mask: 编码器-解码器交互时的注意力掩码，一般为None
        :param tgt_key_padding_mask: 解码部分输入的padding情况，形状为 [batch_size,
tgt_len]
        :param memory_key_padding_mask: 编码部分输入的padding情况，形状为 [batch_size,
src_len]
        :return:
        """
        tgt2 = self.self_attn(tgt, tgt, tgt,  # [tgt_len,batch_size, embed_dim]
                              attn_mask=tgt_mask,
                              key_padding_mask=tgt_key_padding_mask)[0]
        # 解码部分输入序列之间'的多头注意力（也就是论文结构图中的Masked Multi-head attention)

        tgt = tgt + self.dropout1(tgt2)  # 接着是残差连接
        tgt = self.norm1(tgt)  # [tgt_len,batch_size, embed_dim]

        tgt2 = self.multihead_attn(tgt, memory, memory,  # [tgt_len, batch_size,
embed_dim]
                                   attn_mask=memory_mask,
                                   key_padding_mask=memory_key_padding_mask)[0]

        # 解码部分的输入经过多头注意力后同编码部分的输出（memory）通过多头注意力机制进行交互
        tgt = tgt + self.dropout2(tgt2)  # 残差连接
        tgt = self.norm2(tgt)  # [tgt_len, batch_size, embed_dim]

        tgt2 = self.activation(self.linear1(tgt))  # [tgt_len, batch_size,
dim_feedforward]
        tgt2 = self.linear2(self.dropout(tgt2))  # [tgt_len, batch_size, embed_dim]
        # 最后的两层全连接
        tgt = tgt + self.dropout3(tgt2)
        tgt = self.norm3(tgt)
        return tgt  # [tgt_len, batch_size, num_heads * kdim] <==>
[tgt_len,batch_size,embed_dim]


class MyTransformerDecoder(nn.Module):
    def __init__(self, decoder_layer, num_layers, norm=None):
        super(MyTransformerDecoder, self).__init__()
        self.layers = _get_clones(decoder_layer, num_layers)
        self.num_layers = num_layers
        self.norm = norm

    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None,
tgt_key_padding_mask=None,
```

```
                memory_key_padding_mask=None):
        """
        :param tgt: 解码部分的输入，形状为 [tgt_len,batch_size, embed_dim]
        :param memory: 编码部分最后一层的输出 [src_len,batch_size, embed_dim]
        :param tgt_mask: 注意力Mask输入，用于掩盖当前position之后的信息，[tgt_len,
tgt_len]
        :param memory_mask: 编码器-解码器交互时的注意力掩码，一般为None
        :param tgt_key_padding_mask: 解码部分输入的padding情况，形状为 [batch_size,
tgt_len]
        :param memory_key_padding_mask: 编码部分输入的padding情况，形状为 [batch_size,
src_len]
        :return:
        """
        output = tgt  # [tgt_len,batch_size, embed_dim]

        for mod in self.layers:  # 这里的layers就是N层解码层堆叠起来的
            output = mod(output, memory,
                         tgt_mask=tgt_mask,
                         memory_mask=memory_mask,
                         tgt_key_padding_mask=tgt_key_padding_mask,
                         memory_key_padding_mask=memory_key_padding_mask)
        if self.norm is not None:
            output = self.norm(output)

        return output  # [tgt_len, batch_size, num_heads * kdim] <==>
[tgt_len,batch_size,embed_dim]


class MyMultiheadAttention(nn.Module):
    """
    多头注意力机制的计算公式为（就是论文第5页的公式）：
    .. math::
        \text{MultiHead}(Q, K, V) = \text{Concat}(head_1,\dots,head_h)W^O
        \text{where} head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)
    """

    def __init__(self, embed_dim, num_heads, dropout=0., bias=True):
        super(MyMultiheadAttention, self).__init__()
        """
        :param embed_dim:   词嵌入的维度，也就是前面的d_model参数，论文中的默认值为512
        :param num_heads:   多头注意力机制中多头的数量，也就是前面的nhead参数， 论文默认值为 8
        :param dropout:
        :param bias:        最后对多头的注意力（组合）输出进行线性变换时，是否使用偏置
        """
        self.embed_dim = embed_dim  # 前面的d_model参数
        self.head_dim = embed_dim // num_heads  # head_dim 指的就是d_k,d_v
        self.kdim = self.head_dim
        self.vdim = self.head_dim

        self.num_heads = num_heads  # 多头个数
        self.dropout = dropout
```

```
        assert self.head_dim * num_heads == self.embed_dim, "embed_dim 除以 num_heads 必须为整数"
        # 上面的限制条件就是论文中的  d_k = d_v = d_model/n_head 条件

        self.q_proj = nn.Linear(embed_dim, embed_dim, bias=bias)  # embed_dim = kdim * num_heads
        # 这里第二个维度之所以是embed_dim，实际上这里是同时初始化了num_heads个W_q堆叠起来的，也就是num_heads个头
        self.k_proj = nn.Linear(embed_dim, embed_dim, bias=bias)  # W_k,  embed_dim = kdim * num_heads
        self.v_proj = nn.Linear(embed_dim, embed_dim, bias=bias)  # W_v,  embed_dim = vdim * num_heads
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        # 最后将所有的Z组合起来的时候，也是一次性完成，  embed_dim = vdim * num_heads
        self._reset_parameters()

    def _reset_parameters(self):
        """
        以特定方式来初始化参数
        :return:
        """
        for p in self.parameters():
            if p.dim() > 1:
                xavier_uniform_(p)

    def forward(self, query, key, value, attn_mask=None, key_padding_mask=None):
        """
        在论文中，编码时query, key, value 都是同一个输入，  解码时 输入的部分也都是同一个输入，
        解码和编码交互时 key,value指的是 memory, query指的是tgt
        :param query: # [tgt_len, batch_size, embed_dim], tgt_len 表示目标序列的长度
        :param key:  #  [src_len, batch_size, embed_dim], src_len 表示源序列的长度
        :param value: # [src_len, batch_size, embed_dim], src_len 表示源序列的长度
        :param attn_mask: # [tgt_len,src_len] or [num_heads*batch_size,tgt_len, src_len]
                    一般只在解码时使用，为了并行一次喂入所有解码部分的输入，所以要用mask来进行掩盖当前时刻之后的位置信息
        :param key_padding_mask: [batch_size, src_len], src_len 表示源序列的长度
        :return:
        attn_output: [tgt_len, batch_size, embed_dim]
        attn_output_weights: # [batch_size, tgt_len, src_len]
        """
        return multi_head_attention_forward(query, key, value, self.num_heads,
                                            self.dropout,
                                            out_proj=self.out_proj,
                                            training=self.training,
                                            key_padding_mask=key_padding_mask,
                                            q_proj=self.q_proj,
                                            k_proj=self.k_proj,
                                            v_proj=self.v_proj,
                                            attn_mask=attn_mask)
```

```
def multi_head_attention_forward(query,  # [tgt_len,batch_size, embed_dim]
                                 key,  # [src_len, batch_size, embed_dim]
                                 value,  # [src_len, batch_size, embed_dim]
                                 num_heads,
                                 dropout_p,
                                 out_proj,  # [embed_dim = vdim * num_heads,
embed_dim = vdim * num_heads]
                                 training=True,
                                 key_padding_mask=None,  #
[batch_size,src_len/tgt_len]
                                 q_proj=None,  # [embed_dim,kdim * num_heads]
                                 k_proj=None,  # [embed_dim, kdim * num_heads]
                                 v_proj=None,  # [embed_dim, vdim * num_heads]
                                 attn_mask=None,  # [tgt_len,src_len] or
[num_heads*batch_size,tgt_len, src_len]
                                 ):
    q = q_proj(query)
    #  [tgt_len,batch_size, embed_dim] x [embed_dim,kdim * num_heads] =
[tgt_len,batch_size,kdim * num_heads]

    k = k_proj(key)
    # [src_len, batch_size, embed_dim] x [embed_dim, kdim * num_heads] = [src_len,
batch_size, kdim * num_heads]

    v = v_proj(value)
    # [src_len, batch_size, embed_dim] x [embed_dim, vdim * num_heads] = [src_len,
batch_size, vdim * num_heads]
    if is_print_shape:
        print("" + "=" * 80)
        print("进入多头注意力计算:")
        print(
            f"\t 多头num_heads = {num_heads}, d_model={query.size(-1)}, d_k = d_v =
d_model/num_heads={query.size(-1) // num_heads}")
        print(f"\t query的shape([tgt_len, batch_size, embed_dim]):{query.shape}")
        print(f"\t  W_q 的shape([embed_dim,kdim * num_heads]):
{q_proj.weight.shape}")
        print(f"\t   Q  的shape([tgt_len, batch_size,kdim * num_heads]):{q.shape}")
        print("\t" + "-" * 70)

        print(f"\t  key 的shape([src_len,batch_size, embed_dim]):{key.shape}")
        print(f"\t  W_k 的shape([embed_dim,kdim * num_heads]):
{k_proj.weight.shape}")
        print(f"\t   K  的shape([src_len,batch_size,kdim * num_heads]):{k.shape}")
        print("\t" + "-" * 70)

        print(f"\t value的shape([src_len,batch_size, embed_dim]):{value.shape}")
        print(f"\t  W_v 的shape([embed_dim,vdim * num_heads]):
{v_proj.weight.shape}")
        print(f"\t   V  的shape([src_len,batch_size,vdim * num_heads]):{v.shape}")
        print("\t" + "-" * 70)
        print("\t ***** 注意，这里的W_q, W_k, W_v是多个head同时进行计算的. 因此，Q,K,V分别也
是包含了多个head的q,k,v堆叠起来的结果 *****")
```

```
    tgt_len, bsz, embed_dim = query.size()  # [tgt_len,batch_size, embed_dim]
    src_len = key.size(0)
    head_dim = embed_dim // num_heads  # num_heads * head_dim = embed_dim
    scaling = float(head_dim) ** -0.5
    q = q * scaling  # [query_len,batch_size,kdim * num_heads]

    if attn_mask is not None:  # [tgt_len,src_len] or [num_heads*batch_size,tgt_len,
src_len]
        if attn_mask.dim() == 2:
            attn_mask = attn_mask.unsqueeze(0)  # [1, tgt_len,src_len]
            if list(attn_mask.size()) != [1, query.size(0), key.size(0)]:
                raise RuntimeError('The size of the 2D attn_mask is not correct.')
        elif attn_mask.dim() == 3:
            if list(attn_mask.size()) != [bsz * num_heads, query.size(0),
key.size(0)]:
                raise RuntimeError('The size of the 3D attn_mask is not correct.')
        # 现在 atten_mask 的维度就变成了3D

    q = q.contiguous().view(tgt_len, bsz * num_heads, head_dim).transpose(0, 1)
    # [batch_size * num_heads,tgt_len,kdim]
    # 因为前面是num_heads个头一起参与的计算，所以这里要进行一下变形，以便于后面计算。 且同时交换了
0，1两个维度
    k = k.contiguous().view(-1, bsz * num_heads, head_dim).transpose(0, 1)  #
[batch_size * num_heads,src_len,kdim]
    v = v.contiguous().view(-1, bsz * num_heads, head_dim).transpose(0, 1)  #
[batch_size * num_heads,src_len,vdim]
    attn_output_weights = torch.bmm(q, k.transpose(1, 2))
    # [batch_size * num_heads,tgt_len,kdim] x [batch_size * num_heads, kdim,
src_len]
    # =  [batch_size * num_heads, tgt_len, src_len]  这就num_heads个QK相乘后的注意力矩阵

    if attn_mask is not None:
        attn_output_weights += attn_mask  # [batch_size * num_heads, tgt_len,
src_len]

    if key_padding_mask is not None:
        attn_output_weights = attn_output_weights.view(bsz, num_heads, tgt_len,
src_len)
        # 变成 [batch_size, num_heads, tgt_len, src_len]的形状
        attn_output_weights = attn_output_weights.masked_fill(
            key_padding_mask.unsqueeze(1).unsqueeze(2),
            float('-inf'))  #
        # 扩展维度，key_padding_mask从[batch_size,src_len]变成[batch_size,1,1,src_len]
        # 然后再对attn_output_weights进行填充
        attn_output_weights = attn_output_weights.view(bsz * num_heads, tgt_len,
                                                       src_len)  # [batch_size *
num_heads, tgt_len, src_len]

    attn_output_weights = F.softmax(attn_output_weights, dim=-1)  # [batch_size *
num_heads, tgt_len, src_len]
```

```
    attn_output_weights = F.dropout(attn_output_weights, p=dropout_p,
training=training)
    attn_output = torch.bmm(attn_output_weights, v)
    # Z = [batch_size * num_heads, tgt_len, src_len]  x  [batch_size *
num_heads,src_len,vdim]
    # = # [batch_size * num_heads,tgt_len,vdim]
    # 这就num_heads个Attention(Q,K,V)结果

    attn_output = attn_output.transpose(0, 1).contiguous().view(tgt_len, bsz,
embed_dim)
    # 先transpose成 [tgt_len, batch_size* num_heads ,kdim]
    # 再view成 [tgt_len,batch_size,num_heads*kdim]
    attn_output_weights = attn_output_weights.view(bsz, num_heads, tgt_len, src_len)

    Z = out_proj(attn_output)
    # 这里就是多个z  线性组合成Z  [tgt_len,batch_size,embed_dim]
    if is_print_shape:
        print(f"\t 多头注意力中,多头计算结束后的形状（堆叠）为
([tgt_len,batch_size,num_heads*kdim]){attn_output.shape}")
        print(f"\t 多头计算结束后，再进行线性变换时的权重w_o的形状为([num_heads*vdim,
num_heads*vdim  ]){out_proj.weight.shape}")
        print(f"\t 多头线性变化后的形状为([tgt_len,batch_size,embed_dim]) {Z.shape}")
    return Z, attn_output_weights.sum(dim=1) / num_heads  # average attention
weights over heads
```

```
import torch.nn as nn
import torch
from model.MyTransformer import MyTransformer
from torch.nn import Transformer as MyTransformer
from model.Embedding import PositionalEncoding, TokenEmbedding


class TranslationModel(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(TranslationModel, self).__init__()
        self.my_transformer = MyTransformer(d_model=d_model,
                                            nhead=nhead,
                                            num_encoder_layers=num_encoder_layers,
                                            num_decoder_layers=num_decoder_layers,
                                            dim_feedforward=dim_feedforward,
                                            dropout=dropout)
        self.pos_embedding = PositionalEncoding(d_model=d_model, dropout=dropout)
        self.src_token_embedding = TokenEmbedding(src_vocab_size, d_model)
        self.tgt_token_embedding = TokenEmbedding(tgt_vocab_size, d_model)
        self.classification = nn.Linear(d_model, tgt_vocab_size)
        self._reset_parameters()
```

```
    def forward(self, src=None, tgt=None, src_mask=None,
                tgt_mask=None, memory_mask=None, src_key_padding_mask=None,
                tgt_key_padding_mask=None, memory_key_padding_mask=None):
        """

        :param src: Encoder的输入 [src_len,batch_size]
        :param tgt: Decoder的输入 [tgt_len,batch_size]
        :param src_key_padding_mask: 用来Mask掉Encoder中不同序列的padding部分,
[batch_size, src_len]
        :param tgt_key_padding_mask: 用来Mask掉Decoder中不同序列的padding部分
[batch_size, tgt_len]
        :param memory_key_padding_mask: 用来Mask掉Encoder输出的memory中不同序列的padding
部分 [batch_size, src_len]
        :return:
        """
        src_embed = self.src_token_embedding(src)  # [src_len, batch_size,
embed_dim]
        src_embed = self.pos_embedding(src_embed)  # [src_len, batch_size,
embed_dim]
        tgt_embed = self.tgt_token_embedding(tgt)  # [tgt_len, batch_size,
embed_dim]
        tgt_embed = self.pos_embedding(tgt_embed)  # [tgt_len, batch_size,
embed_dim]

        outs = self.my_transformer(src=src_embed, tgt=tgt_embed, src_mask=src_mask,
                                   tgt_mask=tgt_mask, memory_mask=memory_mask,
                                   src_key_padding_mask=src_key_padding_mask,
                                   tgt_key_padding_mask=tgt_key_padding_mask,
                                   memory_key_padding_mask=memory_key_padding_mask)
        # [tgt_len,batch_size,embed_dim]
        logits = self.classification(outs)  # [tgt_len,batch_size,tgt_vocab_size]
        return logits

    def encoder(self, src):
        src_embed = self.src_token_embedding(src)  # [src_len, batch_size,
embed_dim]
        src_embed = self.pos_embedding(src_embed)  # [src_len, batch_size,
embed_dim]
        memory = self.my_transformer.encoder(src_embed)
        return memory

    def decoder(self, tgt, memory):
        tgt_embed = self.tgt_token_embedding(tgt)  # [tgt_len, batch_size,
embed_dim]
        tgt_embed = self.pos_embedding(tgt_embed)  # [tgt_len, batch_size,
embed_dim]
        outs = self.my_transformer.decoder(tgt_embed, memory=memory)  #
[tgt_len,batch_size,embed_dim]
        return outs

    def _reset_parameters(self):
        r"""Initiate parameters in the transformer model."""
```

```
        """
        初始化
        """
        for p in self.parameters():
            if p.dim() > 1:
                nn.init.xavier_uniform_(p)
```

我们构建了一个最基础的Multi-Head self-attention Transformer model，用以进行本次实验的机器翻译任务

模型训练

我们使用acc作为训练的指标，bleu作为验证集和测试集的指标，我们使用了动态的学习率包括warmup&learning rate decay来帮助模型稳定训练，训练代码如下：

```
from copy import deepcopy
from config.config import Config
from model.TranslationModel import TranslationModel
from utils.data_helpers import LoadEnglishGermanDataset, my_tokenizer
from nltk.translate.bleu_score import corpus_bleu
import torch
import time
import os
import logging
import matplotlib.pyplot as plt

os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ['CUDA_VISIBLE_DEVICES'] = '0'

class CustomSchedule(object):
    def __init__(self, d_model, warmup_steps=4000, optimizer=None):
        super(CustomSchedule, self).__init__()
        self.d_model = torch.tensor(d_model, dtype=torch.float32)
        self.warmup_steps = warmup_steps
        self.steps = 1.
        self.optimizer = optimizer

    def step(self):
        arg1 = self.steps ** -0.5
        arg2 = self.steps * (self.warmup_steps ** -1.5)
        self.steps += 1.
        lr = (self.d_model ** -0.5) * min(arg1, arg2)
        for p in self.optimizer.param_groups:
            p['lr'] = lr
        return lr

def accuracy(logits, y_true, PAD_IDX):
    y_pred = logits.transpose(0, 1).argmax(axis=2).reshape(-1)
    y_true = y_true.transpose(0, 1).reshape(-1)
    acc = y_pred.eq(y_true)
```

```
    mask = torch.logical_not(y_true.eq(PAD_IDX))
    acc = acc.logical_and(mask)
    correct = acc.sum().item()
    total = mask.sum().item()
    return float(correct) / total, correct, total

def evaluate(config, valid_iter, model, data_loader, loss_fn):
    model.eval()
    total_loss = 0
    hypotheses = []  # 存储所有预测翻译
    references = []  # 存储所有参考翻译
    with torch.no_grad():
        for idx, (src, tgt) in enumerate(valid_iter):
            src = src.to(config.device)
            tgt = tgt.to(config.device)  # [tgt_len, batch_size]
            tgt_input = tgt[:-1, :]  # 解码部分的输入

            src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = \
                data_loader.create_mask(src, tgt_input, device=config.device)

            logits = model(src=src, tgt=tgt_input, src_mask=src_mask,
tgt_mask=tgt_mask,
                           src_key_padding_mask=src_padding_mask,
tgt_key_padding_mask=tgt_padding_mask,
                           memory_key_padding_mask=src_padding_mask)

            tgt_out = tgt[1:, :]  # 解码部分的真实值
            loss = loss_fn(logits.reshape(-1, logits.shape[-1]),
tgt_out.reshape(-1))
            total_loss += loss.item()

            # 将预测和参考翻译转换为单词列表
            y_pred = logits.argmax(-1).transpose(0, 1)
            for i in range(y_pred.shape[0]):
                hypotheses.append([data_loader.en_vocab.itos[word] for word in
y_pred[i].cpu().numpy()])
                tgt_list = tgt[1:, i].cpu().numpy()
                references.append([[data_loader.en_vocab.itos[word] for word in
tgt_list]])

    bleu_score = corpus_bleu(references, hypotheses)
    model.train()
    return total_loss / len(valid_iter), bleu_score

def train_model(config):
    logging.info("############载入数据集############")
    data_loader = LoadEnglishGermanDataset(config.train_corpus_file_paths,
                                           batch_size=config.batch_size,
                                           tokenizer=my_tokenizer,
                                           min_freq=config.min_freq)
    logging.info("############划分数据集############")
    train_iter, valid_iter, test_iter = \
```

```
        data_loader.load_train_val_test_data(config.train_corpus_file_paths,
                                             config.val_corpus_file_paths,
                                             config.test_corpus_file_paths)
    logging.info("############初始化模型#############")
    translation_model = TranslationModel(src_vocab_size=len(data_loader.de_vocab),
                                         tgt_vocab_size=len(data_loader.en_vocab),
                                         d_model=config.d_model,
                                         nhead=config.num_head,

num_encoder_layers=config.num_encoder_layers,

num_decoder_layers=config.num_decoder_layers,
                                         dim_feedforward=config.dim_feedforward,
                                         dropout=config.dropout)
    model_save_path = os.path.join(config.model_save_dir, 'model.pkl')
    if os.path.exists(model_save_path):
        loaded_paras = torch.load(model_save_path)
        translation_model.load_state_dict(loaded_paras)
        logging.info("#### 成功载入已有模型，进行追加训练...")
    translation_model = translation_model.to(config.device)
    loss_fn = torch.nn.CrossEntropyLoss(ignore_index=data_loader.PAD_IDX)

    optimizer = torch.optim.Adam(translation_model.parameters(),
                                 lr=0.,
                                 betas=(config.beta1, config.beta2),
eps=config.epsilon)
    lr_scheduler = CustomSchedule(config.d_model, optimizer=optimizer)
    translation_model.train()

    train_losses = []
    valid_losses = []

    for epoch in range(config.epochs):
        losses = 0
        start_time = time.time()
        for idx, (src, tgt) in enumerate(train_iter):
            src = src.to(config.device)
            tgt = tgt.to(config.device)
            tgt_input = tgt[:-1, :]
            src_mask, tgt_mask, src_padding_mask, tgt_padding_mask \
                = data_loader.create_mask(src, tgt_input, config.device)
            logits = translation_model(
                src=src,
                tgt=tgt_input,
                src_mask=src_mask,
                tgt_mask=tgt_mask,
                src_key_padding_mask=src_padding_mask,
                tgt_key_padding_mask=tgt_padding_mask,
                memory_key_padding_mask=src_padding_mask)
            optimizer.zero_grad()
            tgt_out = tgt[1:, :]
```

```
            loss = loss_fn(logits.reshape(-1, logits.shape[-1]),
tgt_out.reshape(-1))
            loss.backward()
            lr_scheduler.step()
            optimizer.step()
            losses += loss.item()
            acc, _, _ = accuracy(logits, tgt_out, data_loader.PAD_IDX)
            msg = f"Epoch: {epoch}, Batch[{idx}/{len(train_iter)}], Train loss :
{loss.item():.3f}, Train acc: {acc}"
            logging.info(msg)
        end_time = time.time()
        train_loss = losses / len(train_iter)
        train_losses.append(train_loss)
        valid_loss, valid_bleu = evaluate(config, valid_iter, translation_model,
data_loader, loss_fn)
        valid_losses.append(valid_loss)
        msg = f"Epoch: {epoch}, Train loss: {train_loss:.3f}, Valid loss:
{valid_loss:.3f}, Valid BLEU: {valid_bleu:.3f}, Epoch time = {(end_time -
start_time):.3f}s"
        logging.info(msg)
        if epoch % 2 == 0:
            state_dict = deepcopy(translation_model.state_dict())
            torch.save(state_dict, model_save_path)

    # 绘制损失曲线
    plt.plot(train_losses, label='Training Loss')
    plt.plot(valid_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()
    plt.savefig('fis.png')

    # 在测试集上测试模型
    test_loss, test_bleu = evaluate(config, test_iter, translation_model,
data_loader, loss_fn)
    print(f"Test Loss: {test_loss}, Test BLEU: {test_bleu}")

if __name__ == '__main__':
    config = Config()
    train_model(config)
```

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt


class CustomSchedule(nn.Module):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()
```

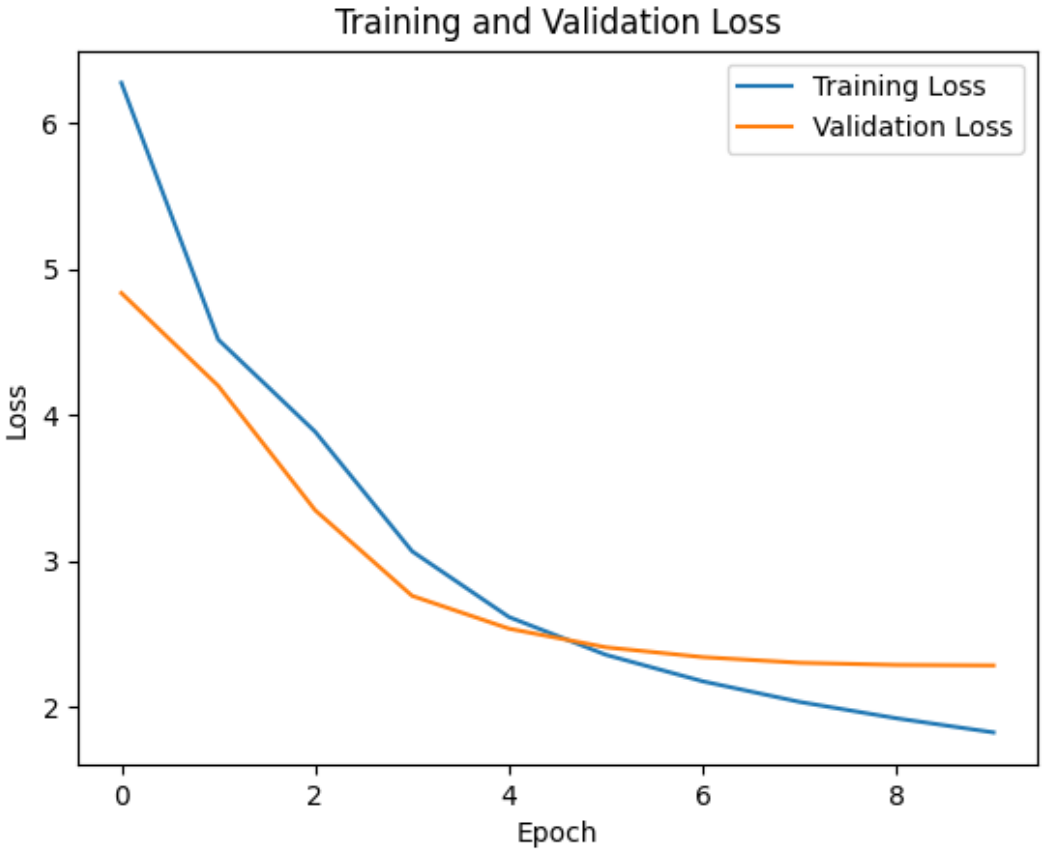
```
        self.d_model = torch.tensor(d_model, dtype=torch.float32)
        self.warmup_steps = warmup_steps
        self.step = 1.

    def __call__(self):
        arg1 = self.step ** -0.5
        arg2 = self.step * (self.warmup_steps ** -1.5)
        self.step += 1.
        return (self.d_model ** -0.5) * min(arg1, arg2)


if __name__ == '__main__':
    for d in [[256, 4000], [512, 4000], [512, 8000]]:
        lr_list = []
        lr = CustomSchedule(d[0], warmup_steps=d[1])
        for i in range(20000):
            lr_list.append(lr())
        plt.plot(lr_list, label=f"d_model = {d[0]}, warm_up = {d[1]}")
    plt.legend()
    plt.xlabel("Steps")
    plt.ylabel("Learning rate")
    plt.show()
```

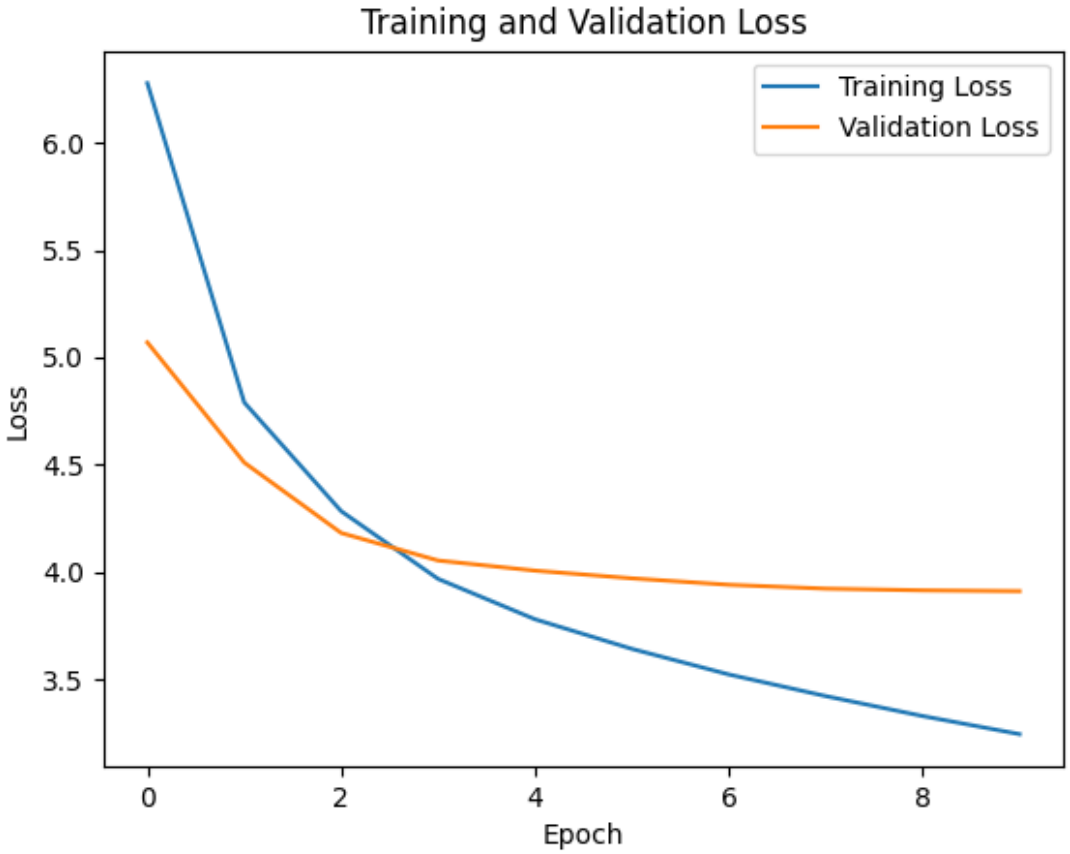
实验结果

在 News Commentary v15 的测试集bleu为：Test Loss: 2.2762728822035867, Test BLEU: 0.11357219915268317



在 Back-translated news 的测试集bleu为：Test Loss: 3.9635218110154655, Test BLEU: 0.04071266889789966

对应的曲线为



最终模型的test bleu score并不高，说明了机器翻译的结果与参考翻译得结果一致度也不是很高，从损失曲线也能看出一定的过拟合现象，可能需要针对性再做一些防止过拟合的手段来得到更好的结果。

结果展示

我们展示 Back-translated news上前50个的结果

```
Building prefix dict from the default dictionary ...
[2024-02-06 04:32:46] - DEBUG: Building prefix dict from the default dictionary ...
Loading model from cache /tmp/jieba.cache
[2024-02-06 04:32:46] - DEBUG: Loading model from cache /tmp/jieba.cache
Loading model cost 0.934 seconds.
[2024-02-06 04:32:47] - DEBUG: Loading model cost 0.934 seconds.
Prefix dict has been built successfully.
[2024-02-06 04:32:47] - DEBUG: Prefix dict has been built successfully.
中文：我们所做的一切是残骸。
翻译： What is all of the way .
英语：All we did locate was wreckage.

中文：这种家务对扁桃体肌肉和肩部肌肉是很好的。
翻译： The good is with a hip and bad luck .
英语：This chore is great for toning arms and shoulder muscles.

中文：他们闻到火葬场里烧焦的人肉的气味。
翻译： They smell wall birds .
英语：They smelled the... burning human flesh coming from the crematoria.
```

中文：Conwy 县坠机事件中的2名妇女死亡。
翻译： Two children are in court case two five officers crash cases two men driver cases
英语：Woman dies, two injured in Conwy county crash

中文：IIR
翻译： Josh
英语：IR.

中文：基克允许用户保持匿名。
翻译： The will protect them .
英语：Kik allows users to remain anonymous.

中文：自行车盗窃是开放式校园政策的头号牺牲品。
翻译： The policy is a factor in the fund .
英语：Bike thefts are the number one casualty of open campus policy.

中文：Rushcliffe
翻译： Josh
英语：Rushcliffe

中文：他是71岁。
翻译： He .
英语：He was 71.

中文：上面显示的是渲染它可能是什么样的东西。
翻译： What looks .
英语：A rendering of what it may have looked like is shown above.

中文：这种松露不能被分享。
翻译： The can .
英语：The truffles could not be shared.

中文：他在实地的才干应进一步探讨。
翻译： He too during a ban .
英语：His talent for the field should be explored further.

中文：唯一的选择
翻译： The only
英语：The only choice

中文：联合王国国防部
翻译： The UK
英语：UK Ministry of Defence

中文：作为支持球队这个赛季最积极的方面是什么?
翻译： Is the team in the best ?
英语：As a supporter what has been the most positive aspect about your team this season?

中文：医疗保险不包括救护车。

翻译： The health or medical doctors were called .
英语：Ambulances are not covered by Medicare.

中文：这幅画高于预期的售价3500万至5000万美元。
翻译： The $ 70 million is over the year .
英语：The painting sold above its expected sale price of $35 million to $50 million.

中文：房地产市场在所有年龄段都有优势。
翻译： The most standard .
英语：The real estate mogul also has a commanding advantage across all age groups.

中文：凯特·温斯林在金球奖和巴弗斯的比赛中赢得了最佳的支持。
翻译： Kate and the black won for the award for the Danish nomination .
英语：Kate Winslet stands a strong chance of taking home best supporting actress after winning in the category both at the Golden Globes and the Baftas.

中文：关于 Beyan 视频的争议
翻译： Google
英语：Controversy over Beyonce video

中文：什叶派的拉博夫是一个以艺术的名义进入电梯的人。
翻译： The Nazi is a Catholic name in the Nazi statue .
英语：Shia LaBeouf is trapping himself inside an elevator in the name of art.

中文：他们的研究提供了关于独立公投的一些令人着迷的信息和观察。
翻译： They and new evidence from the various language .
英语：Their study offers some fascinating nuggets of information and observation about the independence referendum.

中文：该协会将仇恨团体定义为攻击基于种族、性取向和宗教等核心特征的组织的组织。
翻译： The includes religious , through background , background , the appeal of religious and religious movement to ban .
英语：The SPLC defines hate groups as organizations that attack people based on central characteristics such as race, sexual orientation and religion.

中文：Bernat Mosgue / AP 照片
翻译： Photo AP
英语：Bernat Armangue / AP Photo

中文：巴拉克·奥巴马和家人在复活节
翻译： Obama and
英语：Barack Obama and Family on Easter Sunday

中文：这给我们造成了极大的不安。
翻译： This deep us .
英语：It caused us a huge amount of concern.

中文：20世纪80年代通过了一个由电话运营商和国家运营商控制的连接节点的封闭网络。
翻译： The is owned by every owned owned by a group of the National between the company and the German company .

英语：Adopted in the 1980s as a closed network with connecting nodes controlled by phone carriers and national operators, SS7 directs mobile traffic from cellphone towers to the Internet.

中文：我可以打子弹。
翻译： I hit a hand .
英语：I can hit shots.

中文：此举使消费者受益于标准的国内天然气关税。
翻译： The protect water to supply consumers to supply prices .
英语：The moves benefit customers on a standard domestic gas tariff.

中文：她是邪恶的化身!
翻译： She is !
英语：She is evil personified!

中文：失去子女补助金的人数可能在五年内上升50%。
翻译： The 50 children will be 50 to five per cent in their life .
英语：The number losing child benefit might rise by 50% within five years.

中文：前妻子保罗·麦卡特尼的妻子希瑟·米尔斯和王菲的歌手汤姆·帕克被任命为顶替者。
翻译： The wife Grace and former singer named singer , was named alongside wife John .
英语：Heather Mills, the former wife of Sir Paul McCartney, and The Wanted singer Tom Parker have been drafted in as replacements.

中文：我的厨房画廊。
翻译： My The library
英语：My kitchen gallery: Georgia Levy

中文：有些人在压力过大或过度兴奋时往往会打嗝。
翻译： Some often or angry .
英语：Some people tend to hiccup when stressed or overexcited.

中文：凯特·厄普顿热气球升空
翻译： Jennifer Maria
英语：Kate Upton Takes a Hot Air Balloon

中文：上次比赛
翻译： Last
英语：Last contest

中文：她们有自己的背景和历史上的女性和女权运动。
翻译： They and a character in other people .
英语：They have some kind of background and history of women and feminism.

中文：她还在德黑兰度过了童年的一部分。
翻译： She also during the Nazi World War II .
英语：She also spent part of her childhood in Tehran.

中文：这一次没有男朋友。
翻译： This never .

```
英语：No boyfriend this time around...

中文：我们采取疟疾预防措施。
翻译： We order .
英语：We take malaria prophylaxis.

中文：388g 箱
翻译： £ 15
英语：388g box

中文：塞德里克·福特在工厂里杀害了三人
翻译： Billy dead
英语：Cedric Ford killed three people at factory

中文：你什么时候开始意识到人们觉得你很有趣?
翻译： What you like when she was ?
英语：When did you begin to realise that people found you funny?

中文：按新闻协会
翻译： By Press
英语：By Press Association

中文：由此产生的与美国的争吵已被莫斯科利用。
翻译： The Latin in Russia .
英语：The resulting row with the US has been exploited by Moscow.

中文：Coacin 4
翻译： after
英语：Costac4

中文：我们把这个家带到了加拿大。
翻译： We . to Canada .
英语：We brought this home for Canada.

中文：DNA 只是我们身体的另一个可能出错。
翻译： It only a human can not .
英语：DNA is just another bit of our body that might go wrong.

中文：没有人能相信。
翻译： Nobody .
英语：Nobody could believe it.

中文：虾髻
翻译： Josh
英语：Prawn bun cha
```

实验收获

1. 了解了基本的tansformer的原理和使用方法
2. 了解了bleu score等基本的机器翻译指标

3. 对机器翻译任务有了一个感性的认知