华为“智能基座”系列课程

基于Mindspore-Ascend的Transformer实现“德译英”翻译实验

版本：1.0



华为技术有限公司

|  |
| --- |
| 版权所有 © 华为技术有限公司 2024。 保留一切权利。  非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。  商标声明  HW_POS_RBG_Vertical-150ppi 和其他华为商标均为华为技术有限公司的商标。  本文档提及的其他所有商标或注册商标，由各自的所有人拥有。  注意  您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。  由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。 |

|  |  |
| --- | --- |
| 华为技术有限公司 | |
| 地址： | 深圳市龙岗区坂田华为总部办公楼 邮编：518129 |
| 网址： | http://[e](http://e.huawei.com/).huawei.com |

目录

[1 华为云ModelArts环境创建 2](#_Toc181888109)

[1.1 创建ModelArts开发环境 2](#_Toc181888110)

[1.2 关闭/删除环境 5](#_Toc181888111)

[1.2.1 关闭环境 5](#_Toc181888112)

[1.2.2 删除环境 5](#_Toc181888113)

[2 基于MindSpore的Transformer实现“德译英”翻译实验 6](#_Toc181888114)

[2.1 实验介绍与Transformer概述 6](#_Toc181888115)

[2.2 实验过程 6](#_Toc181888116)

[2.2.1 依赖包安装 6](#_Toc181888117)

[2.2.2 数据处理 8](#_Toc181888118)

[2.2.3 模型结构 17](#_Toc181888119)

[2.2.4 模型训练 27](#_Toc181888120)

[2.2.5 模型推理 30](#_Toc181888121)

[2.2.6 BLEU得分 32](#_Toc181888122)

[3 课后思考题 33](#_Toc181888123)

[3.1 思考题一 33](#_Toc181888124)

[3.2 思考题二 33](#_Toc181888125)

# 华为云ModelArts环境创建

在华为云ModelArts平台上创建AI框架为Mindspore-2.2、硬件环境为昇腾处理器+ARM的开发环境。

## 创建ModelArts开发环境

进入华为云ModelArts控制台

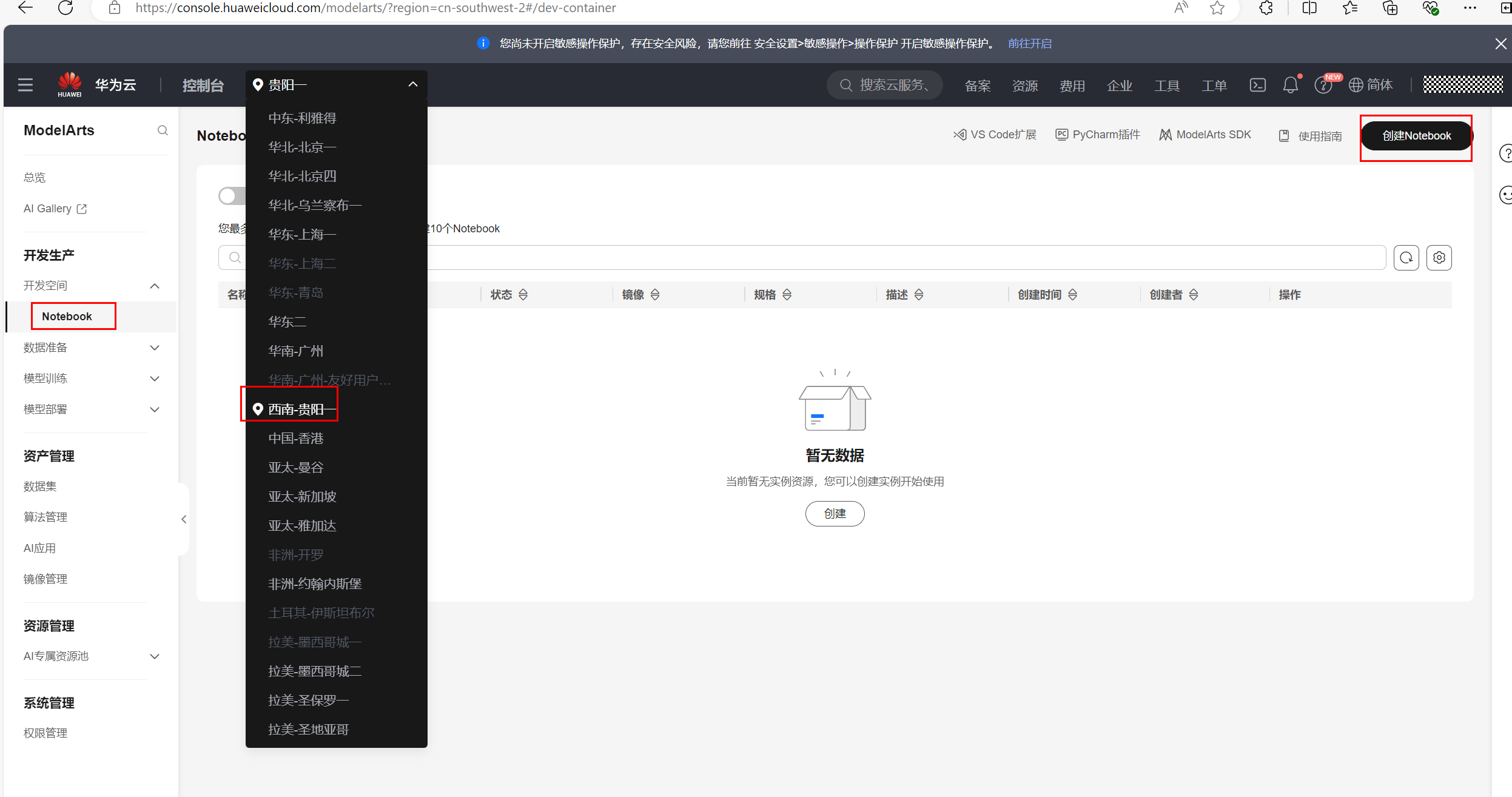
在[华为云ModelArts主页](https://www.huaweicloud.com/product/modelarts.html)，点击“管理控制台”进入ModelArts的管理页面。



华为云ModelArts主页

创建Notebook训练作业

控制台区域选择“西南-贵阳一”，在左侧菜单栏中选择“开发空间”的“Notebook”，如下图：



ModelArts控制台

在网页右上角，点击“创建Notebook”按钮，创建一个新的Notebook，其配置如下：

* 名称：自定义。
* **自动停止：打开，默认1小时。**
* 镜像：选择“公共镜像”，选择MindSpore2.2版本、Ascend+ARM的基础镜像。
* 类型：ASCEND。
* 规格：Ascend: 1\*ascend-snt9b(32G)|ARM: 24核 192GB。
* 存储配置：云硬盘EVS，128GB，可根据实验需要自定义。

注意：云硬盘从Notebook实例创建成功起，直至删除成功，每GB按照规定费用收费。

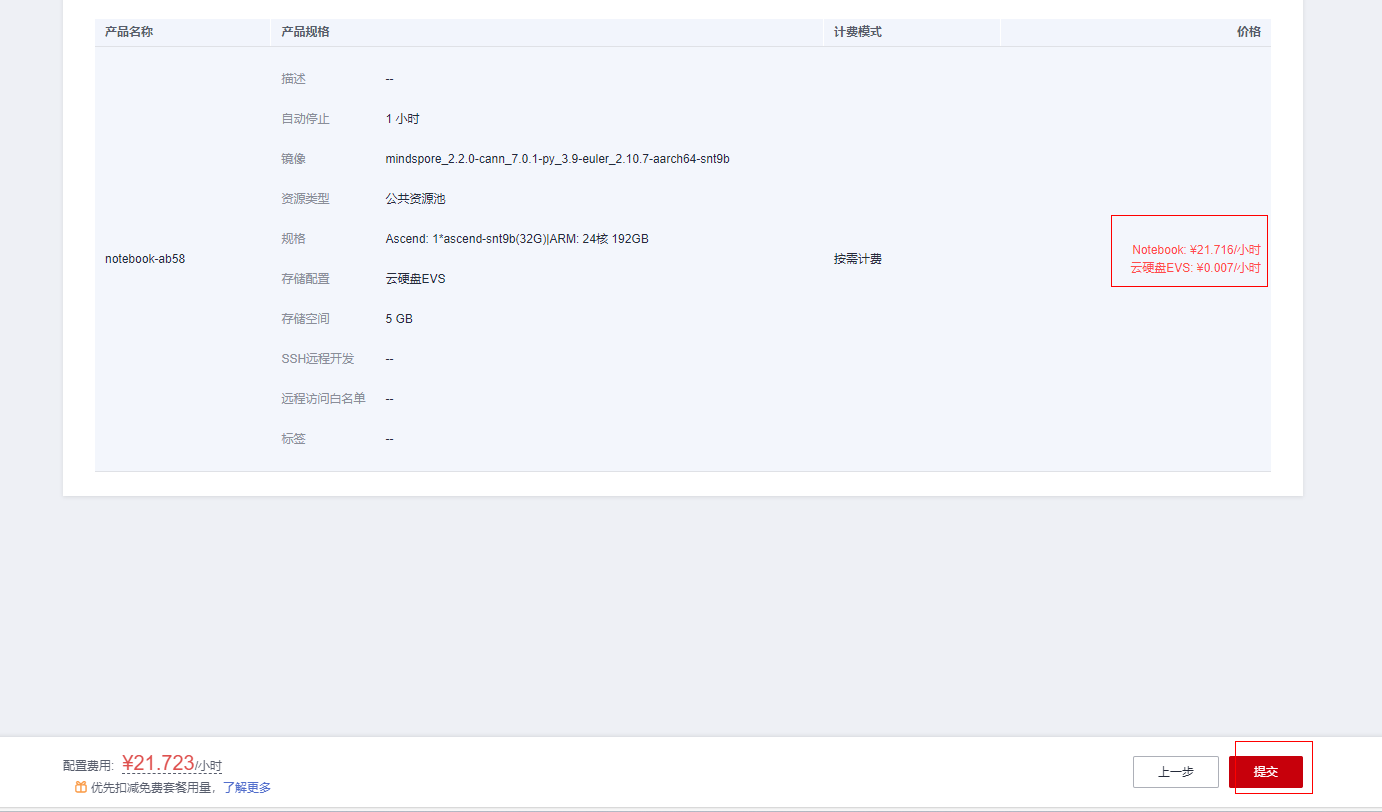
如图所示：





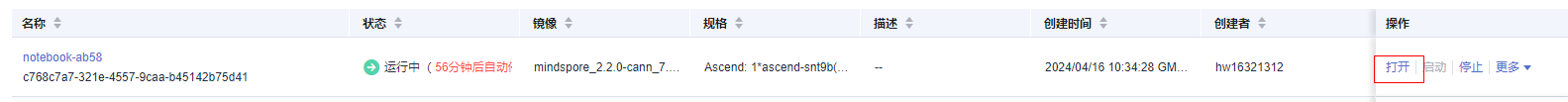
Notebook创建配置

配置完成之后“立即创建”，规格确认无误之后“提交”。



启动Notebook进入开发环境

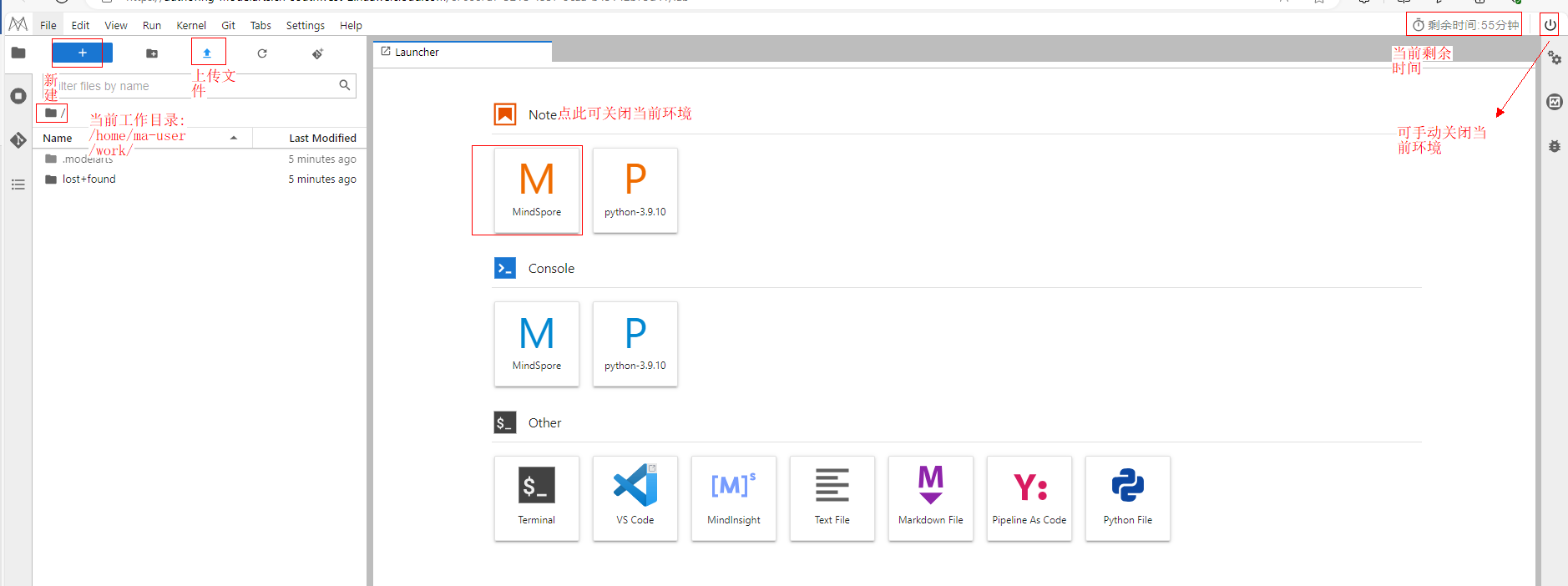
当上一步创建好Notebook状态显示为“运行中”时，在右侧操作中“打开”，即可进入在线编程页面。



Notebook实例入口

可以在此页面创建或编辑MindSpore的项目，如图所示：

\*注意：Notebook环境内上传、创建和编辑的文件均在/home/ma-user/work目录下。



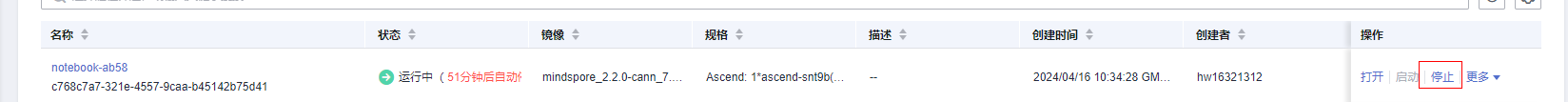
环境开发页面

## 关闭/删除环境

实验完成之后，请及时关闭或删除Notebook训练作业，避免产生不必要的资源浪费。

### 关闭环境

可点击上图右上角的“终止按钮”关闭环境，亦可在[ModelArts控制台](https://console.huaweicloud.com/modelarts/?region=cn-north-4#/dev-container)的操作栏选择“停止”操作，如下图所示：



关闭环境

### 删除环境

关闭环境后，由于云硬盘从Notebook实例创建成功起直至删除成功，每GB按照规定费用收费，因此当不再使用该云环境时，建议删除环境。

在[ModelArts控制台](https://console.huaweicloud.com/modelarts/?region=cn-north-4#/dev-container)的操作栏打开“更多”，选择“删除”操作，如下图所示：



删除环境

至此ModelArts的人工智能的训练环境搭建完成。

# 基于MindSpore的Transformer实现“德译英”翻译实验

## 实验介绍与Transformer概述

实验介绍：基于华为云ModelArts实现transformer的网络搭建，并使用Multi30k数据集进行训练，实现德语翻译英语的功能。

实验目的：提高对华为云ModelArts的熟练度，加深对transformer网络的理解。熟悉使用Mindspore训练网络的过程。

Transformer概述：

Transformer是一种基于注意力机制结构的神经网络，由Vaswani等人在2017年的论文“Attention Is All You Need”中提出，用于处理机器翻译、语言建模和文本生成等自然语言处理任务。

Transformer与传统NLP特征提取类模型的区别主要在以下两点：

* Transformer将自注意力机制和多头注意力机制的概念运用到模型中；
* 由于缺少RNN模型的时序性，Transformer引入了位置编码，在数据上而非模型中添加位置信息；

以上的处理带来了2个优点：

* 更容易并行化，训练更加高效；
* 在处理长序列的任务中表现优秀，可以快速捕捉长距离中的关联信息。

Transformer的模型结构由**编码器Encoder**和**解码器Decoder**组成，整体结构图如右图所示。

本文将从数据处理开始，顺序介绍embedding、位置编码、编码器、解码器的原理及代码实现，最后介绍网络训练和推理部分。

## 实验过程

### 依赖包安装

#### MindSpore更新

新建Notebook,并且输入以下代码下载MindSpore2.2.13

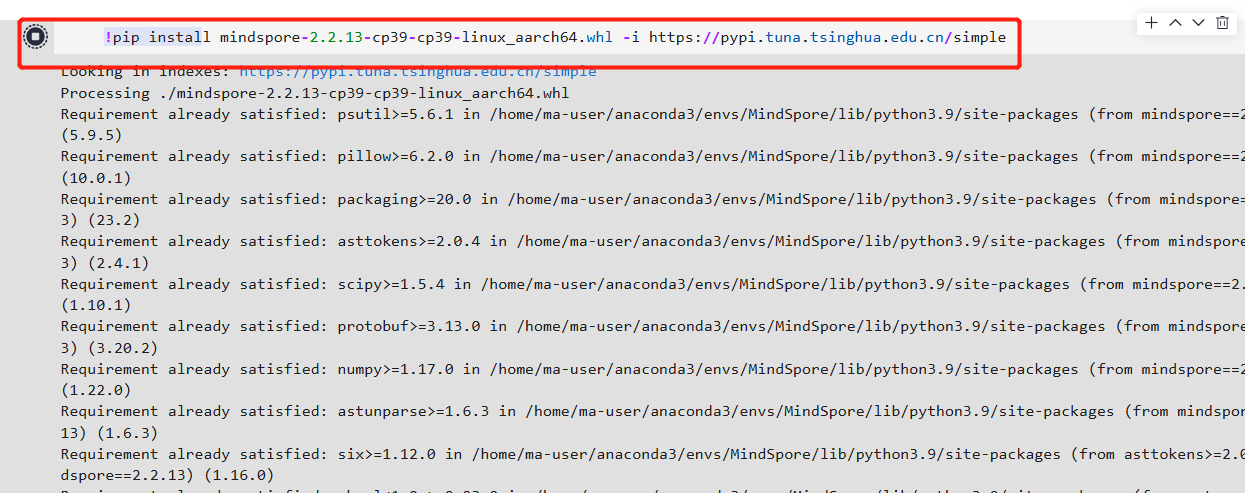
%env no\_proxy='a.test.com,127.0.0.1,2.2.2.2'

!wget <https://ms-release.obs.cn-north-4.myhuaweicloud.com/2.2.13/MindSpore/unified/aarch64/mindspore-2.2.13-cp39-cp39-linux_aarch64.whl>

安装MindSpore2.2.13

!pip install mindspore-2.2.13-cp39-cp39-linux\_aarch64.whl -i https://pypi.tuna.tsinghua.edu.cn/simple

运行结果：

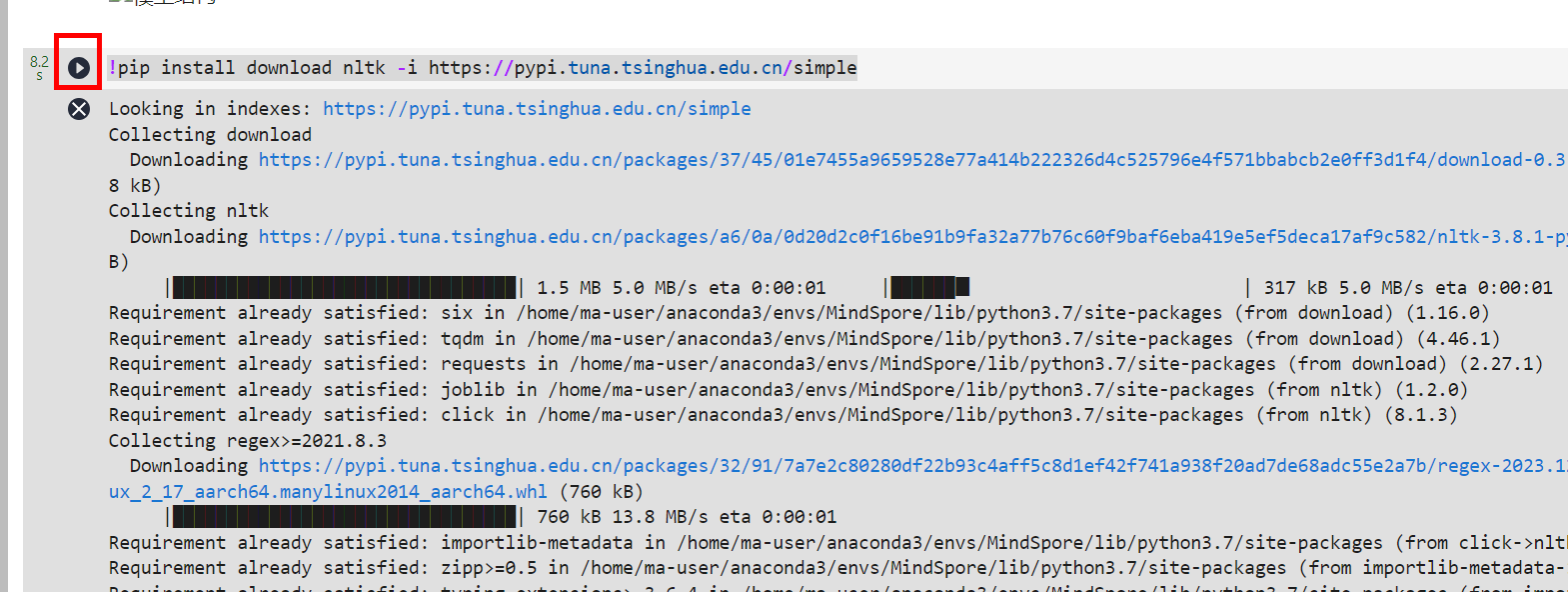


#### 安装nltk库

在执行代码前需要准备MindSpore环境，安装依赖包：

!pip install download nltk -i <https://pypi.tuna.tsinghua.edu.cn/simple>

安装输出：



### 数据处理

#### 数据集下载

我们本次使用的数据集为**Multi30K数据集**，它是一个大规模的图像-文本数据集，包含30K+图片，每张图片都有对应的德文和英文的文本描述。

在本次文本翻译任务中，德语是源语言（source languag），英语是目标语言（target language），运行下面的代码下载并展示数据集：

from download import download

import re

url = "https://modelscope.cn/api/v1/datasets/SelinaRR/Multi30K/repo?Revision=master&FilePath=Multi30K.zip"

download(url, './', kind='zip', replace=True)

datasets\_path = './datasets/'

train\_path = datasets\_path + 'train/'

valid\_path = datasets\_path + 'valid/'

test\_path = datasets\_path + 'test/'

def print\_data(data\_file\_path, print\_n=5):

print("=" \* 40 + "datasets in {}".format(data\_file\_path) + "=" \* 40)

with open(data\_file\_path, 'r', encoding='utf-8') as en\_file:

en = en\_file.readlines()[:print\_n]

for index, seq in enumerate(en):

print(index, seq.replace('\n', ''))

print\_data(train\_path + 'train.de')

print\_data(train\_path + 'train.en')

#### 数据读取

加载数据集，并进行分词，即将句子拆解为单独的词元（token，可以为字符或者单词）。一般在机器翻译类任务中，我们习惯进行单词级词元化，即每个词元要么为一个单词，要么为一个标点符号。同一个单词，不论首字母是否大写，都应该对应同一个词元，故在分词前，我们需统一将单词转换为小写。

"Hello world!" --> ["hello", "world", "!"]

接下来，我们创建数据加载器Multi30K。后期调用该类进行遍历时，每次返回当前源语言（德语）与目标语言（英语）文本描述的词元列表。

import os

class Multi30K():

"""Multi30K数据集加载器，加载Multi30K数据集并处理为一个Python迭代对象"""

def \_\_init\_\_(self, path):

self.data = self.\_load(path)

def \_load(self, path):

def tokenize(text):

text = text.rstrip()

return [tok.lower() for tok in re.findall(r'\w+|[^\w\s]', text)]

def read\_data(data\_file\_path):

with open(data\_file\_path, 'r', encoding='utf-8') as data\_file:

data = data\_file.readlines()[:-1]

return [tokenize(i) for i in data]

members = {i.split('.')[-1]: path + i for i in os.listdir(path)}

ret = [read\_data(members['de']), read\_data(members['en'])]

return list(zip(\*ret))

def \_\_getitem\_\_(self, idx):

return self.data[idx]

def \_\_len\_\_(self):

return len(self.data)

train\_dataset, valid\_dataset, test\_dataset = Multi30K(train\_path), Multi30K(valid\_path), Multi30K(test\_path)

for de, en in train\_dataset:

print(f'德文：{de}')

print(f'英文：{en}')

break

#### 构建词典

将每个词元映射到从0开始的数字索引中（为节约存储空间，可过滤掉词频低的词元），词元和数字索引所构成的集合叫做词典（vocabulary）。

以上述“Hello world!”为例，该序列组成的词典为：

{"<unk>": 0, "<pad>": 1, "<bos>": 2, "<eos>": 3, "hello": 4, "world": 5, "!": 6}

在构建词典中，我们使用了4个特殊词元。

* <unk>：未知词元（unknown），将出现次数少于一定频率的单词统一判定为未知词元；
* <bos>：起始词元（begin of sentence），用来标注一个句子的开始；
* <eos>：结束词元（end of sentence），用来标注一个句子的结束；
* <pad>：填充词元（padding），当句子长度不够时将句子填充至统一长度；

通过Vocab创建词典后，我们可以实现词元与数字索引之间的互相转换。我们可以通过调用enocde函数，获取输入词元对应的数字索引，通过调用decode函数，获取输入数字索引对应的词元。同样，我们也可以完成词元列表和数字索引列表之间的转换。

class Vocab:

"""通过词频字典，构建词典"""

special\_tokens = ['<unk>', '<pad>', '<bos>', '<eos>']

def \_\_init\_\_(self, word\_count\_dict, min\_freq=1):

self.word2idx = {}

for idx, tok in enumerate(self.special\_tokens):

self.word2idx[tok] = idx

filted\_dict = {w: c for w, c in word\_count\_dict.items() if c >= min\_freq}

for w, \_ in filted\_dict.items():

self.word2idx[w] = len(self.word2idx)

self.idx2word = {idx: word for word, idx in self.word2idx.items()}

self.bos\_idx = self.word2idx['<bos>']

self.eos\_idx = self.word2idx['<eos>']

self.pad\_idx = self.word2idx['<pad>']

self.unk\_idx = self.word2idx['<unk>']

def \_word2idx(self, word):

"""单词映射至数字索引"""

if word not in self.word2idx:

return self.unk\_idx

return self.word2idx[word]

def \_idx2word(self, idx):

"""数字索引映射至单词"""

if idx not in self.idx2word:

raise ValueError('input index is not in vocabulary.')

return self.idx2word[idx]

def encode(self, word\_or\_list):

"""将单个单词或单词数组映射至单个数字索引或数字索引数组"""

if isinstance(word\_or\_list, list):

return [self.\_word2idx(i) for i in word\_or\_list]

return self.\_word2idx(word\_or\_list)

def decode(self, idx\_or\_list):

"""将单个数字索引或数字索引数组映射至单个单词或单词数组"""

if isinstance(idx\_or\_list, list):

return [self.\_idx2word(i) for i in idx\_or\_list]

return self.\_idx2word(idx\_or\_list)

def \_\_len\_\_(self):

return len(self.word2idx)

使用collections中的Counter和OrderedDict统计英/德语每个单词在整体文本中出现的频率。构建词频字典，然后再将词频字典转为词典。其中，收录所有源语言（德语）词元的词典为de\_vocab，收录所有目标语言（英语）词元的词典为en\_vocab。

在分配数字索引时有一个小技巧：常用的词元对应数值较小的索引，这样可以节约检索时间。

from collections import Counter, OrderedDict

def build\_vocab(dataset):

de\_words, en\_words = [], []

for de, en in dataset:

de\_words.extend(de)

en\_words.extend(en)

de\_count\_dict = OrderedDict(sorted(Counter(de\_words).items(), key=lambda t: t[1], reverse=True))

en\_count\_dict = OrderedDict(sorted(Counter(en\_words).items(), key=lambda t: t[1], reverse=True))

return Vocab(de\_count\_dict, min\_freq=2), Vocab(en\_count\_dict, min\_freq=2)

de\_vocab, en\_vocab = build\_vocab(train\_dataset)

print('Unique tokens in de vocabulary:{} and en vocabulary:{}\n'.format(len(de\_vocab), len(en\_vocab)))

str\_seq\_en = ['two', 'young', ',', 'white', 'males', 'are', 'outside', 'near', 'many', 'bushes', '.']

print("word:{}\nindex:{}\n".format(str\_seq\_en, en\_vocab.encode(str\_seq\_en)))

index = [5, 6, 7, 8, 9, 10]

print("index:{}\nword:{}".format(index, en\_vocab.decode(index)))

输出：

Unique tokens in de vocabulary:7882 and en vocabulary:5898

word:['two', 'young', ',', 'white', 'males', 'are', 'outside', 'near', 'many', 'bushes', '.']

index:[16, 24, 15, 25, 776, 17, 57, 80, 204, 1305, 5]

index:[5, 6, 7, 8, 9, 10]

word:['.', 'in', 'the', 'on', 'man', 'is']

从上面的打印可以看出，我们完成了从词元到索引、从索引到词元之间的转换。

#### 数据迭代器

数据处理的最后一步是创建数据迭代器。截至目前，我们已经通过数据加载器Multi30K将源语言（德语）与目标语言（英语）的文本描述转换为词元序列，并构建了词元与数字索引一一对应的词典，接下来，需要将词元序列转换为数字索引序列。

还是以“Hello world!”为例，我们逐步演示数据迭代器中的操作

1、我们将表示开始和结束的特殊词元<bos>和<eos>分别添加在每个词元序列的句首和句尾。

处理前：["hello", "world", "!"]

处理后：["<bos>", "hello", "world", "!", "<eos>"]

2、统一序列长度（超出长度的进行截断，未达到长度的通过填充<pad>进行补齐）,同时记录序列的有效长度。此处假定统一的长度为7。

处理前：["<bos>", "hello", "world", "!", "<eos>"]

处理后：["<bos>", "hello", "world", "!", "<eos>", "<pad>", "<pad>"]， valid length = 5

3、最后，对文本序列进行批处理。对于每个batch中的序列，通过调用词典中的encode为序列中的所有词元找到其对应的数字索引，将结果以Tensor的形式返回。

处理前：["<bos>", "hello", "world", "!", "<eos>", "<pad>", "<pad>"]

处理后：Tensor([2, 4, 5, 6, 3, 1, 1])

import mindspore

class Iterator():

"""创建数据迭代器"""

def \_\_init\_\_(self, dataset, de\_vocab, en\_vocab, batch\_size, max\_len=32, drop\_reminder=False):

self.dataset = dataset

self.de\_vocab = de\_vocab

self.en\_vocab = en\_vocab

self.batch\_size = batch\_size

self.max\_len = max\_len

self.drop\_reminder = drop\_reminder

length = len(self.dataset) // batch\_size

self.len = length if drop\_reminder else length + 1 # 批量数量

def \_\_call\_\_(self):

def pad(idx\_list, vocab, max\_len):

"""统一序列长度，并记录有效长度"""

idx\_pad\_list, idx\_len = [], []

for i in idx\_list:

if len(i) > max\_len - 2:

idx\_pad\_list.append([vocab.bos\_idx] + i[:max\_len-2] + [vocab.eos\_idx])

idx\_len.append(max\_len)

else:

idx\_pad\_list.append([vocab.bos\_idx] + i + [vocab.eos\_idx] + [vocab.pad\_idx] \* (max\_len - len(i) - 2))

idx\_len.append(len(i) + 2)

return idx\_pad\_list, idx\_len

def sort\_by\_length(src, trg):

"""根据src的字段长度进行排序"""

data = zip(src, trg)

data = sorted(data, key=lambda t: len(t[0]), reverse=True)

return zip(\*list(data))

def encode\_and\_pad(batch\_data, max\_len):

"""将批量中的文本数据转换为数字索引，并统一每个序列的长度"""

src\_data, trg\_data = zip(\*batch\_data)

src\_idx = [self.de\_vocab.encode(i) for i in src\_data]

trg\_idx = [self.en\_vocab.encode(i) for i in trg\_data]

src\_idx, trg\_idx = sort\_by\_length(src\_idx, trg\_idx)

src\_idx\_pad, src\_len = pad(src\_idx, de\_vocab, max\_len)

trg\_idx\_pad, \_ = pad(trg\_idx, en\_vocab, max\_len)

return src\_idx\_pad, src\_len, trg\_idx\_pad

for i in range(self.len):

if i == self.len - 1 and not self.drop\_reminder:

batch\_data = self.dataset[i \* self.batch\_size:]

else:

batch\_data = self.dataset[i \* self.batch\_size: (i+1) \* self.batch\_size]

src\_idx, src\_len, trg\_idx = encode\_and\_pad(batch\_data, self.max\_len)

yield mindspore.Tensor(src\_idx, mindspore.int32), \

mindspore.Tensor(src\_len, mindspore.int32), \

mindspore.Tensor(trg\_idx, mindspore.int32)

def \_\_len\_\_(self):

return self.len

train\_iterator = Iterator(train\_dataset, de\_vocab, en\_vocab, batch\_size=128, max\_len=32, drop\_reminder=True)

valid\_iterator = Iterator(valid\_dataset, de\_vocab, en\_vocab, batch\_size=128, max\_len=32, drop\_reminder=False)

test\_iterator = Iterator(test\_dataset, de\_vocab, en\_vocab, batch\_size=1, max\_len=32, drop\_reminder=False)

for src\_idx, src\_len, trg\_idx in train\_iterator():

print(f'src\_idx.shape:{src\_idx.shape}\n{src\_idx}\nsrc\_len.shape:{src\_len.shape}\n{src\_len}\ntrg\_idx.shape:{trg\_idx.shape}\n{trg\_idx}')

break

输出：

src\_idx.shape:(128, 32)

[[ 2 5 13 ... 1 1 1]

[ 2 5 13 ... 1 1 1]

[ 2 5 13 ... 1 1 1]

...

[ 2 5 52 ... 1 1 1]

[ 2 8 37 ... 1 1 1]

[ 2 5 33 ... 1 1 1]]

src\_len.shape:(128,)

[27 25 24 24 23 23 23 23 22 22 22 21 21 21 21 21 20 20 20 20 20 19 19 19

18 18 18 18 18 18 18 18 17 17 17 17 17 17 17 17 17 17 16 16 16 16 16 16

16 16 16 16 15 15 15 15 15 15 15 15 15 15 15 14 14 14 14 14 14 14 14 14

14 14 14 14 13 13 13 13 13 13 13 13 13 12 12 12 12 12 12 12 12 12 12 12

12 12 12 12 12 12 12 12 11 11 11 11 11 11 11 11 11 11 10 10 10 10 10 10

10 9 9 9 9 9 9 8]

trg\_idx.shape:(128, 32)

[[ 2 4 2243 ... 1 1 1]

[ 2 4 9 ... 1 1 1]

[ 2 4 9 ... 1 1 1]

...

[ 2 4 55 ... 1 1 1]

[ 2 4 38 ... 1 1 1]

[ 2 4 35 ... 1 1 1]]

从上面的打印可以看出，每个batch有128个句子，每个句子的有效长度不尽相同，但总长度都被统一处理成了32。

#### Word embedding

词元token的索引index无法准确表示词的含义，为此我们引入词向量来表示词元。word embedding操作就是把词元的索引映射到词向量的过程。刚开始训练时，词向量是随机初始化的，训练后期才趋于准确，因此下面的代码在执行时的结果具有随机性。另外，不同的任务对词向量的维度要求也不一样，这里的维度可以简单理解为词性分析。

from mindspore import nn, Tensor

word\_index = Tensor([21, 28, 49, 12, 275, 119, 49, 23, 54, 32])

src\_emb = nn.Embedding(len(de\_vocab), 4)

enc\_outputs = src\_emb(word\_index)

print(enc\_outputs)

输出：

[[-1.45623935e-02 -4.86097159e-03 -9.34928539e-04 4.68974002e-03]

[ 1.07922601e-02 6.97764102e-04 -2.59567145e-02 1.30578419e-02]

[ 1.39968535e-02 -7.42947636e-03 5.32284239e-03 8.29133112e-03]

[ 1.28781004e-02 3.31685552e-03 1.06496066e-02 3.63264419e-03]

[-9.77428909e-03 -1.71048939e-03 -4.34428919e-03 -1.99727481e-03]

[-8.76351260e-03 -8.25243909e-03 -8.53217952e-03 1.70198118e-03]

[ 1.39968535e-02 -7.42947636e-03 5.32284239e-03 8.29133112e-03]

[-6.72905520e-03 3.95659590e-03 1.86168291e-02 -1.55664689e-03]

[-6.32177899e-03 4.99482779e-03 -1.60118807e-02 1.24117211e-02]

[ 1.45842796e-02 5.51572209e-03 7.22815603e-05 3.75842326e-03]]

#### 位置编码

位置编码𝑃𝐸的形状与经过word embedding后的输出𝑋相同，对于索引为[pos, 2i]的元素，以及索引为[pos, 2i+1]的元素，位置编码的计算如下：

在下面的代码中，我们实现了位置编码，输入经过word embedding后的结果𝑋，输出添加位置信息后的结果𝑋+𝑃𝐸。

import mindspore

from mindspore import nn, ops, Tensor

from mindspore import dtype as mstype

from mindspore import numpy as mnp

class PositionalEncoding(nn.Cell):

"""位置编码"""

def \_\_init\_\_(self, d\_model, dropout\_p=0.1, max\_len=100):

super().\_\_init\_\_()

self.dropout = nn.Dropout(1 - dropout\_p)

self.pe = ops.Zeros()((max\_len, d\_model), mstype.float32)

pos = mnp.arange(0, max\_len, dtype=mstype.float32).view((-1, 1))

angle = ops.pow(10000.0, mnp.arange(0, d\_model, 2, dtype=mstype.float32)/d\_model)

self.pe[:, 0::2] = ops.sin(pos/angle)

self.pe[:, 1::2] = ops.cos(pos/angle)

def construct(self, x):

batch\_size = x.shape[0]

pe = self.pe.expand\_dims(0)

pe = ops.broadcast\_to(pe, (batch\_size, -1, -1))

x = x + pe[:, :x.shape[1], :]

return self.dropout(x)

x = ops.Zeros()((1, 10, 4), mstype.float32)

pe = PositionalEncoding(4)

print(pe(x))

输出：

[[-2.5319790e-03 6.3022482e-03 1.1523547e-02 1.3428002e-02]

[ 1.6073976e-02 -2.6973659e-05 3.6076386e-03 -4.6042274e-03]

[-1.1364570e-02 3.3582787e-03 -6.7372052e-03 4.4691828e-03]

[-7.9177227e-03 2.1175742e-03 6.7918161e-03 -3.0649467e-03]

[-1.1215101e-03 7.3979357e-03 9.2820851e-03 -1.2527270e-02]

[-6.8438123e-03 2.1198750e-02 -3.9475216e-03 7.3993159e-04]

[-1.1364570e-02 3.3582787e-03 -6.7372052e-03 4.4691828e-03]

[ 1.9603861e-03 6.7669908e-03 1.6084902e-02 -9.3285274e-03]

[-1.7430523e-02 -3.3243808e-03 -1.3079916e-02 -7.4667772e-03]

[ 2.2126625e-03 -6.2489207e-03 3.1161383e-03 1.4870527e-02]]

### 模型结构

#### 注意力机制：

相同的一句话，不同的人听的时候侧重点也可能不同。在自然语言处理中，根据任务内容的不同，句子中需要重点关注的部分也会不同。为此，我们引入注意力机制来判断在执行某个任务时，词在句子中的重要性，并通过注意力分数来表示词的重要程度。分数越高，说明该词对完成该任务的重要性越大。 在计算注意力分数时，我们主要参考三个因素：**query**、**key**和**value**。

* query：任务内容
* key：索引/标签（帮助定位到答案）
* value：答案

注意力分数的计算公式为：

同时，为了避免query（）和key()本身的“大小”影响到相似度的计算，我们需要在点乘后除以

在如下代码中，我们实现了scaled dot-product attention的计算， 调用类后，返回的是加权后的value（output）以及注意力权重（attn）。

class ScaledDotProductAttention(nn.Cell):

def \_\_init\_\_(self, dropout\_p=0.):

super().\_\_init\_\_()

self.softmax = nn.Softmax()

self.dropout = nn.Dropout(1-dropout\_p)

self.sqrt = ops.Sqrt()

def construct(self, query, key, value, attn\_mask=None):

"""scaled dot product attention"""

embed\_size = query.shape[-1]

scaling\_factor = self.sqrt(Tensor(embed\_size, mstype.float16))

attn = ops.matmul(query, key.swapaxes(-2, -1) / scaling\_factor)

if attn\_mask is not None:

attn = attn.masked\_fill(attn\_mask, -1e9)

attn = self.softmax(attn)

attn = self.dropout(attn)

output = ops.matmul(attn, value)

return (output, attn)

attention = ScaledDotProductAttention()

q\_s = k\_s = v\_s = ops.ones((128, 8, 32, 64), mindspore.float32)

attn\_mask = ops.ones((128, 8, 32, 32), mindspore.float32)

attn\_mask = mindspore.ops.gt(attn\_mask, attn\_mask)

output, attn = attention(q\_s, k\_s, v\_s, attn\_mask)

print(output.shape, attn.shape)

输出：

(128, 8, 32, 64) (128, 8, 32, 32)dis这里有error的报错，如何规避

#### 多头注意力（Multi-Head Attention）

多头注意力是注意力机制的扩展，它可以使模型通过不同的方式关注输入序列的不同部分，从而提升模型的训练效果。

不同于之前一次计算整体输入的注意力分数，多头注意力是多次计算，每次计算输入序列中某一部分的注意力分数，最后再将结果进行整合。

多头注意力通过对输入的embedding乘以不同的权重参数、和，将其映射到多个小维度空间中，我们称之为“头”（head），每个头部会并行计算自己的自注意力分数。

、和为可学习的权重参数。一般为了平衡计算成本，我们会取。

在获得多组自注意力分数后，我们将结果拼接到一起，得到多头注意力的最终输出。为可学习的权重参数，用于将拼接后的多头注意力输出映射回原来的维度。

简单来说，在多头注意力中，每个头部可以'解读'输入内容的不同方面，比如：捕捉全局依赖关系、关注特定语境下的词元、识别词和词之间的语法关系等。

class MultiHeadAttention(nn.Cell):

def \_\_init\_\_(self, d\_model, d\_k, n\_heads, dropout\_p=0.):

super().\_\_init\_\_()

self.n\_heads = n\_heads

self.d\_k = d\_k

self.W\_Q = nn.Dense(d\_model, d\_k \* n\_heads)

self.W\_K = nn.Dense(d\_model, d\_k \* n\_heads)

self.W\_V = nn.Dense(d\_model, d\_k \* n\_heads)

self.W\_O = nn.Dense(n\_heads \* d\_k, d\_model)

self.attention = ScaledDotProductAttention(dropout\_p=dropout\_p)

def construct(self, query, key, value, attn\_mask):

"""

query: [batch\_size, len\_q, d\_model]

key: [batch\_size, len\_k, d\_model]

value: [batch\_size, len\_k, d\_model]

attn\_mask: [batch\_size, seq\_len, seq\_len]

"""

batch\_size = query.shape[0]

q\_s = self.W\_Q(query).view(batch\_size, -1, self.n\_heads, self.d\_k)

k\_s = self.W\_K(key).view(batch\_size, -1, self.n\_heads, self.d\_k)

v\_s = self.W\_V(value).view(batch\_size, -1, self.n\_heads, self.d\_k)

q\_s = q\_s.transpose((0, 2, 1, 3))

k\_s = k\_s.transpose((0, 2, 1, 3))

v\_s = v\_s.transpose((0, 2, 1, 3))

attn\_mask = attn\_mask.expand\_dims(1)

attn\_mask = ops.tile(attn\_mask, (1, self.n\_heads, 1, 1))

context, attn = self.attention(q\_s, k\_s, v\_s, attn\_mask)

context = context.transpose((0, 2, 1, 3)).view((batch\_size, -1, self.n\_heads \* self.d\_k))

output = self.W\_O(context)

return output, attn

enc\_input = ops.ones((128, 32, 512), mindspore.float32)

attn\_mask = ops.ones((128, 32, 32), mindspore.float32)

attn\_mask = mindspore.ops.gt(attn\_mask, attn\_mask)

mha = MultiHeadAttention(512, 64, 8)

output, attn = mha(enc\_input, enc\_input, enc\_input, attn\_mask)

print("output.shape:{}\nattn\_mask.shape:{}\n".format(output.shape, attn.shape))

输出：

output.shape:(128, 32, 512)

attn\_mask.shape:(128, 8, 32, 32)

在上面介绍[数据迭代器](https://authoring-modelarts-cnnorth4.huaweicloud.com/417784cd-f677-4565-b538-0038b0caae7d/lab/workspaces/auto-r/tree/transformer-code%20(1).ipynb#%E6%95%B0%E6%8D%AE%E8%BF%AD%E4%BB%A3%E5%99%A8)时有提到，不同的句子token个数也不尽相同，我们为了统一模型输入的长度，使用<pad>占位符补齐一些稍短的文本。

处理前："Hello world!"

处理后：<bos> hello world ! <eos> <pad> <pad>

这些<pad>占位符没有任何意义，不应该参与注意力分数计算中。为此我们在注意力中加入了padding掩码，即识别输入序列中的<pad>占位符，保证计算时这些位置对应的注意力分数为0。

def get\_attn\_pad\_mask(seq\_q, seq\_k, pad\_idx):

"""注意力掩码：识别序列中的<pad>占位符

Args:

seq\_q (Tensor): query序列，shape = [batch size, query len]

seq\_k (Tensor): key序列，shape = [batch size, key len]

pad\_idx (Tensor): key序列<pad>占位符对应的数字索引

"""

batch\_size, len\_q = seq\_q.shape

batch\_size, len\_k = seq\_k.shape

pad\_attn\_mask = ops.equal(seq\_k, pad\_idx)

pad\_attn\_mask = pad\_attn\_mask.expand\_dims(1)

pad\_attn\_mask = ops.broadcast\_to(pad\_attn\_mask, (batch\_size, len\_q, len\_k))

return pad\_attn\_mask

q = k = Tensor([[1, 1, 0, 0]], mstype.float32)

pad\_idx = 0

mask = get\_attn\_pad\_mask(q, k, pad\_idx)

print(mask)

输出：

[[[False False True True]

[False False True True]

[False False True True]

[False False True True]]]

#### 基于位置的前馈神经网络（Position-Wise Feed-Forwardf Network）

基于位置的前馈神经网络被用来对输入中的每个位置进行非线性变换。它由两个线性层组成，层与层之间需要经过ReLU激活函数。

相比固定的ReLU函数，基于位置的前馈神经网络可以处理更加复杂的关系，并且由于前馈网络是基于位置的，可以捕获到不同位置的信息，并为每个位置提供不同的转换。

class PoswiseFeedForward(nn.Cell):

def \_\_init\_\_(self, d\_ff, d\_model, dropout\_p=0.):

super().\_\_init\_\_()

self.linear1 = nn.Dense(d\_model, d\_ff)

self.linear2 = nn.Dense(d\_ff, d\_model)

self.dropout = nn.Dropout(1-dropout\_p)

self.relu = nn.ReLU()

def construct(self, x):

"""前馈神经网络

x: [batch\_size, seq\_len, d\_model]

"""

x = self.linear1(x)

x = self.relu(x)

x = self.dropout(x)

output = self.linear2(x)

return output

x = ops.ones((128, 32, 512), mstype.float32)

ffn = PoswiseFeedForward(2048, 512)

print(ffn(x).shape)

输出：

(128, 32, 512)

#### Add&Norm

Add & Norm层本质上是残差连接后紧接了一个LayerNorm层。

* Add：残差连接，帮助缓解网络退化问题，注意需要满足𝑥与SubLayer(𝑥)的形状一致；
* Norm：Layer Norm，层归一化，帮助模型更快地进行收敛；

class AddNorm(nn.Cell):

def \_\_init\_\_(self, d\_model, dropout\_p=0.):

super().\_\_init\_\_()

self.layer\_norm = nn.LayerNorm((d\_model, ), epsilon=1e-5)

self.dropout = nn.Dropout(1-dropout\_p)

def construct(self, x, residual):

return self.layer\_norm(self.dropout(x) + residual)

#### 编码网络

##### Encoder Layer

我们首先实现encoder中的一个层。

class EncoderLayer(nn.Cell):

def \_\_init\_\_(self, d\_model, n\_heads, d\_ff, dropout\_p=0.):

super().\_\_init\_\_()

d\_k = d\_model // n\_heads

if d\_k \* n\_heads != d\_model:

raise ValueError(f"The `d\_model` {d\_model} can not be divisible by `num\_heads` {n\_heads}.")

self.enc\_self\_attn = MultiHeadAttention(d\_model, d\_k, n\_heads, dropout\_p)

self.pos\_ffn = PoswiseFeedForward(d\_ff, d\_model, dropout\_p)

self.add\_norm1 = AddNorm(d\_model, dropout\_p)

self.add\_norm2 = AddNorm(d\_model, dropout\_p)

def construct(self, enc\_inputs, enc\_self\_attn\_mask):

"""

enc\_inputs: [batch\_size, src\_len, d\_model]

enc\_self\_attn\_mask: [batch\_size, src\_len, src\_len]

"""

residual = enc\_inputs

enc\_outputs, attn = self.enc\_self\_attn(enc\_inputs, enc\_inputs, enc\_inputs, enc\_self\_attn\_mask)

enc\_outputs = self.add\_norm1(enc\_outputs, residual)

residual = enc\_outputs

enc\_outputs = self.pos\_ffn(enc\_outputs)

enc\_outputs = self.add\_norm2(enc\_outputs, residual)

return enc\_outputs, attn

x = ops.ones((128, 32, 512), mstype.float32)

mask = Tensor([False]).broadcast\_to((128, 32, 32))

encoder\_layer = EncoderLayer(512, 8, 2018)

output, attn = encoder\_layer(x, mask)

print(output.shape, attn.shape)

输出

(128, 32, 512) (128, 8, 32, 32)

##### 编码器Encoder

将上面实现的encoder层堆叠n\_layers次，并添加wording embedding与positional encoding。

class Encoder(nn.Cell):

def \_\_init\_\_(self, src\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.):

super().\_\_init\_\_()

self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model)

self.pos\_emb = PositionalEncoding(d\_model, dropout\_p)

self.layers = nn.CellList([EncoderLayer(d\_model, n\_heads, d\_ff, dropout\_p)] \* n\_layers)

self.scaling\_factor = ops.Sqrt()(Tensor(d\_model, mstype.float32))

def construct(self, enc\_inputs, src\_pad\_idx):

"""enc\_inputs : [batch\_size, src\_len]

"""

enc\_outputs = self.src\_emb(enc\_inputs.astype(mstype.int32))

enc\_outputs = self.pos\_emb(enc\_outputs \* self.scaling\_factor)

enc\_self\_attn\_mask = get\_attn\_pad\_mask(enc\_inputs, enc\_inputs, src\_pad\_idx)

enc\_self\_attns = []

for layer in self.layers:

enc\_outputs, enc\_self\_attn = layer(enc\_outputs, enc\_self\_attn\_mask)

enc\_self\_attns.append(enc\_self\_attn)

return enc\_outputs, enc\_self\_attns

encoder = Encoder(len(de\_vocab), 512, 8, 2048, 6)

for src\_idx, src\_len, trg\_idx in train\_iterator():

enc\_outputs, enc\_self\_attns = encoder(src\_idx, de\_vocab.pad\_idx)

print("enc\_outputs.shape:{}\n".format(enc\_outputs.shape))

break

输出：

enc\_outputs.shape:(128, 32, 512)

解码器将编码器输出的上下文序列转换为目标序列的预测结果，该输出将在模型训练中与真实目标输出Y进行比较，计算损失。

不同于编码器，每个Decoder层中包含两层多头注意力机制,并在最后多出一个线性层，输出对目标序列的预测结果。

第一层：计算目标序列的注意力分数的**掩码多头自注意力**；

第二层：用于计算上下文序列与目标序列对应关系，其中Decoder掩码多头注意力的输出作为query，Encoder的输出（上下文序列）作为key和value；

带掩码的多头注意力

在处理目标序列的输入时，t时刻的模型只能“观察”直到t-1时刻的所有词元，后续的词语不应该一并输入Decoder中。

为了保证在t时刻，只有t-1个词元作为输入参与多头注意力分数的计算，我们需要在第一个多头注意力中额外增加一个时间掩码，使目标序列中的词随时间发展逐个被暴露出来。

该注意力掩码可通过三角矩阵实现，对角线以上的词元表示为不参与注意力计算的词元，标记为1。

0 1 1 1 1

0 0 1 1 1

0 0 0 1 1

0 0 0 0 1

该掩码一般被称作subsequent mask。

最后，将subsequent mask和padding mask合并为一个整体的掩码，确保模型既不会注意到t时刻以后的词元，也不会关注为<pad>的词元。

def get\_attn\_subsequent\_mask(seq\_q, seq\_k):

"""生成时间掩码，使decoder在第t时刻只能看到序列的前t-1个元素

Args:

seq\_q (Tensor): query序列，shape = [batch size, len\_q]

seq\_k (Tensor): key序列，shape = [batch size, len\_k]

"""

batch\_size, len\_q = seq\_q.shape

batch\_size, len\_k = seq\_k.shape

ones = ops.ones((batch\_size, len\_q, len\_k), mindspore.float32)

subsequent\_mask = mnp.triu(ones, k=1)

return subsequent\_mask

q = k = ops.ones((1, 4), mstype.float32)

mask = get\_attn\_subsequent\_mask(q, k)

print(mask)

输出：

[[[0. 1. 1. 1.]

[0. 0. 1. 1.]

[0. 0. 0. 1.]

[0. 0. 0. 0.]]]

#### 解码网络

##### Decoder Layer

首先实现Decoder中的一个层。

class DecoderLayer(nn.Cell):

def \_\_init\_\_(self, d\_model, n\_heads, d\_ff, dropout\_p=0.):

super().\_\_init\_\_()

d\_k = d\_model // n\_heads

if d\_k \* n\_heads != d\_model:

raise ValueError(f"The `d\_model` {d\_model} can not be divisible by `num\_heads` {n\_heads}.")

self.dec\_self\_attn = MultiHeadAttention(d\_model, d\_k, n\_heads, dropout\_p)

self.dec\_enc\_attn = MultiHeadAttention(d\_model, d\_k, n\_heads, dropout\_p)

self.pos\_ffn = PoswiseFeedForward(d\_ff, d\_model, dropout\_p)

self.add\_norm1 = AddNorm(d\_model, dropout\_p)

self.add\_norm2 = AddNorm(d\_model, dropout\_p)

self.add\_norm3 = AddNorm(d\_model, dropout\_p)

def construct(self, dec\_inputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask):

"""

dec\_inputs: [batch\_size, trg\_len, d\_model]

enc\_outputs: [batch\_size, src\_len, d\_model]

dec\_self\_attn\_mask: [batch\_size, trg\_len, trg\_len]

dec\_enc\_attn\_mask: [batch\_size, trg\_len, src\_len]

"""

residual = dec\_inputs

dec\_outputs, dec\_self\_attn = self.dec\_self\_attn(dec\_inputs, dec\_inputs, dec\_inputs, dec\_self\_attn\_mask)

dec\_outputs = self.add\_norm1(dec\_outputs, residual)

residual = dec\_outputs

dec\_outputs, dec\_enc\_attn = self.dec\_enc\_attn(dec\_outputs, enc\_outputs, enc\_outputs, dec\_enc\_attn\_mask)

dec\_outputs = self.add\_norm2(dec\_outputs, residual)

residual = dec\_outputs

dec\_outputs = self.pos\_ffn(dec\_outputs)

dec\_outputs = self.add\_norm3(dec\_outputs, residual)

return dec\_outputs, dec\_self\_attn, dec\_enc\_attn

x = y = ops.ones((128, 32, 512), mstype.float32)

mask1 = mask2 = Tensor([False]).broadcast\_to((128, 32, 32))

decoder\_layer = DecoderLayer(512, 8, 2048)

output, attn1, attn2 = decoder\_layer(x, y, mask1, mask2)

print(output.shape, attn1.shape, attn2.shape)

输出：

(128, 32, 512) (128, 8, 32, 32) (128, 8, 32, 32)

##### 解码器Decoder

将上面实现的DecoderLayer堆叠n\_layer次，添加word embedding与positional encoding，以及最后的线性层。

输出的dec\_outputs为对目标序列的预测。

class Decoder(nn.Cell):

def \_\_init\_\_(self, trg\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.):

super().\_\_init\_\_()

self.trg\_emb = nn.Embedding(trg\_vocab\_size, d\_model)

self.pos\_emb = PositionalEncoding(d\_model, dropout\_p)

self.layers = nn.CellList([DecoderLayer(d\_model, n\_heads, d\_ff) for \_ in range(n\_layers)])

self.projection = nn.Dense(d\_model, trg\_vocab\_size)

self.scaling\_factor = ops.Sqrt()(Tensor(d\_model, mstype.float32))

def construct(self, dec\_inputs, enc\_inputs, enc\_outputs, src\_pad\_idx, trg\_pad\_idx):

"""

dec\_inputs: [batch\_size, trg\_len]

enc\_inputs: [batch\_size, src\_len]

enc\_outputs: [batch\_size, src\_len, d\_model]

"""

dec\_outputs = self.trg\_emb(dec\_inputs.astype(mstype.int32))

dec\_outputs = self.pos\_emb(dec\_outputs \* self.scaling\_factor)

dec\_self\_attn\_pad\_mask = get\_attn\_pad\_mask(dec\_inputs, dec\_inputs, trg\_pad\_idx)

dec\_self\_attn\_subsequent\_mask = get\_attn\_subsequent\_mask(dec\_inputs, dec\_inputs)

dec\_self\_attn\_mask = ops.gt((dec\_self\_attn\_pad\_mask + dec\_self\_attn\_subsequent\_mask), 0)

dec\_enc\_attn\_mask = get\_attn\_pad\_mask(dec\_inputs, enc\_inputs, src\_pad\_idx)

dec\_self\_attns, dec\_enc\_attns = [], []

for layer in self.layers:

dec\_outputs, dec\_self\_attn, dec\_enc\_attn = layer(dec\_outputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask)

dec\_self\_attns.append(dec\_self\_attn)

dec\_enc\_attns.append(dec\_enc\_attn)

dec\_outputs = self.projection(dec\_outputs)

return dec\_outputs, dec\_self\_attns, dec\_enc\_attns

#### Transformer

将实现的Encoder与Decoder组合起来。

class Transformer(nn.Cell):

def \_\_init\_\_(self, encoder, decoder):

super().\_\_init\_\_()

self.encoder = encoder

self.decoder = decoder

self.encoder.to\_float(mindspore.float16)

self.decoder.to\_float(mindspore.float16)

def construct(self, enc\_inputs, dec\_inputs, src\_pad\_idx, trg\_pad\_idx):

"""

enc\_inputs: [batch\_size, src\_len]

dec\_inputs: [batch\_size, trg\_len]

"""

enc\_outputs, enc\_self\_attns = self.encoder(enc\_inputs, src\_pad\_idx)

dec\_outputs, dec\_self\_attns, dec\_enc\_attns = self.decoder(dec\_inputs, enc\_inputs, enc\_outputs, src\_pad\_idx, trg\_pad\_idx)

dec\_logits = dec\_outputs.view((-1, dec\_outputs.shape[-1]))

return dec\_logits, enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns

### 模型训练

#### 定义超参数&实例化Transformer模型

src\_vocab\_size = len(de\_vocab)

trg\_vocab\_size = len(en\_vocab)

src\_pad\_idx = de\_vocab.pad\_idx

trg\_pad\_idx = en\_vocab.pad\_idx

d\_model = 512

d\_ff = 2048

n\_layers = 6

n\_heads = 8

encoder = Encoder(src\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.1)

decoder = Decoder(trg\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.1)

model = Transformer(encoder, decoder)

#### 定义损失函数与优化器

* 损失函数：定义如何计算模型输出(logits)与目标(targets)之间的误差，这里可以使用交叉熵损失（CrossEntropyLoss）
* 优化器：MindSpore将模型优化算法的实现称为**优化器**。优化器内部定义了模型的参数优化过程（即梯度如何更新至模型参数），所有优化逻辑都封装在优化器对象中。

loss\_fn = nn.CrossEntropyLoss(ignore\_index=trg\_pad\_idx)

optimizer = nn.Adam(model.trainable\_params(), learning\_rate=0.0001)

#### 模型训练逻辑

1、Network+loss function直接构造正向函数

2、函数变换，获得梯度计算（反向传播）函数

3、构造训练过程函数

4、调用函数进行训练

#### 定义前向网络计算逻辑

在训练过程中，表示句子结尾的<eos>占位符应是被模型预测出来，而不是作为模型的输入，所以在处理Decoder的输入时，我们需要移除目标序列最末的<eos>占位符。

其中，𝑥𝑖代表目标序列中第i个表示实际内容的词元。

我们期望最终的输出包含表示句末的<eos>，不包含表示句首的<bos>，所以在计算损失时，需要同样去除的目标序列的句首<bos>占位符，再进行比较。

其中，𝑦𝑖表示预测的第i个实际内容词元。

def forward(enc\_inputs, dec\_inputs):

"""前向网络

enc\_inputs: [batch\_size, src\_len]

dec\_inputs: [batch\_size, trg\_len]

"""

logits, \_, \_, \_ = model(enc\_inputs, dec\_inputs[:, :-1], src\_pad\_idx, trg\_pad\_idx)

targets = dec\_inputs[:, 1:].view(-1)

loss = loss\_fn(logits, targets)

return loss

#### 定义梯度计算函数

为了优化模型参数，需要求参数对loss的导数。我们调用mindspore.ops.value\_and\_grad函数，来获得function的微分函数。

常用到的参数有三种：

* fn：待求导的函数；
* grad\_position：指定求导输入位置的索引；
* weights：指定求导的参数；

由于使用Cell封装神经网络模型，模型参数为Cell的内部属性，此时我们不需要使用grad\_position指定对函数输入求导，因此将其配置为None。对模型参数求导时，我们使用weights参数，使用model.trainable\_params()方法从Cell中取出可以求导的参数。

grad\_fn = ops.value\_and\_grad(forward, None, optimizer.parameters)

#### 定义模型训练函数

from tqdm import tqdm

def train(iterator, epoch=0):

model.set\_train(True)

num\_batches = len(iterator)

total\_loss = 0

total\_steps = 0

with tqdm(total=num\_batches, unit='step', desc='Train ') as t:

for src, src\_len, trg in iterator():

loss, grads = grad\_fn(src, trg)

optimizer(grads)

total\_loss += loss.asnumpy()

total\_steps += 1

curr\_loss = total\_loss / total\_steps

t.set\_postfix({'loss': f'{curr\_loss:.2f}', 'epoch': f'{epoch:03}'})

t.update(1)

return total\_loss / total\_steps

#### 定义模型评估逻辑

在评估中，仅需正向计算loss，无需更新模型参数,故模型状态需设置为训练model.set\_train(False)。

def evaluate(iterator):

model.set\_train(False)

num\_batches = len(iterator)

total\_loss = 0

total\_steps = 0

with tqdm(total=num\_batches, unit='step', desc='Evaluate') as t:

for src, \_, trg in iterator():

loss = forward(src, trg)

total\_loss += loss.asnumpy()

total\_steps += 1

curr\_loss = total\_loss / total\_steps

t.set\_postfix({'loss': f'{curr\_loss:.2f}'})

t.update(1)

return total\_loss / total\_steps

#### 模型训练

数据集遍历迭代，一次完整的数据集遍历成为一个epoch。我们逐个epoch打印训练的损失值和评估精度，并通过save\_checkpoint保存评估精度最高的ckpt文件（transformer.ckpt）到home\_path/.mindspore\_examples/transformer.ckpt。

from mindspore import save\_checkpoint

num\_epochs = 10

best\_valid\_loss = float('inf')

ckpt\_file\_name = './transformer.ckpt'

for i in range(num\_epochs):

train\_loss = train(train\_iterator, i)

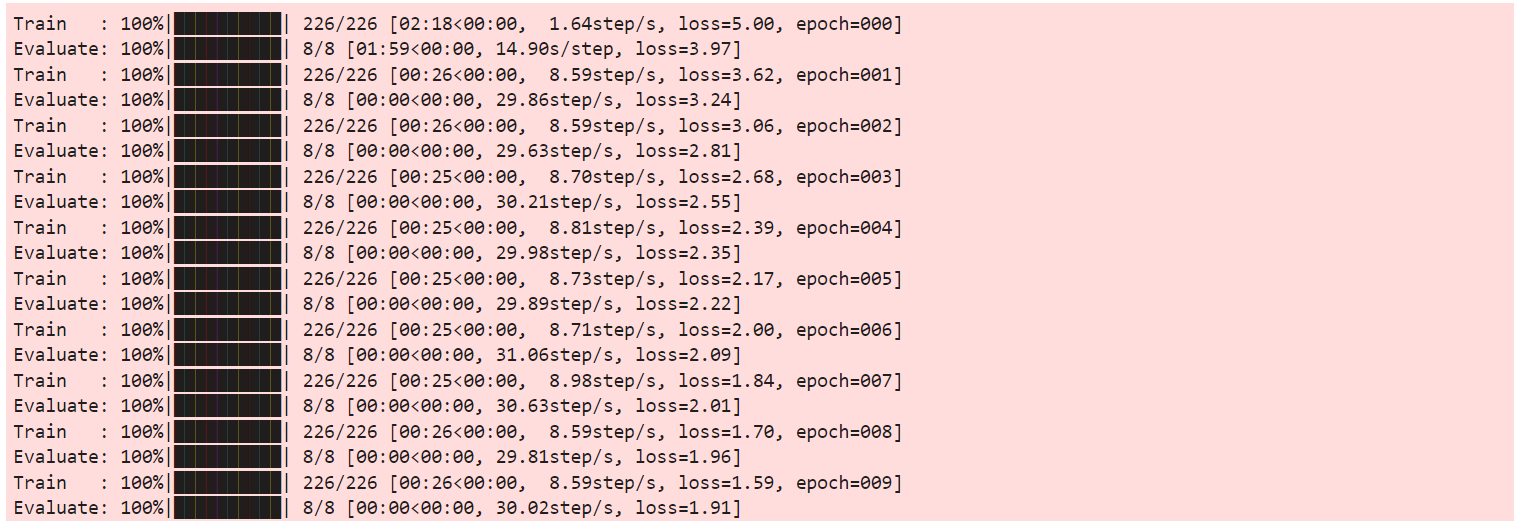
valid\_loss = evaluate(valid\_iterator)

if valid\_loss < best\_valid\_loss:

best\_valid\_loss = valid\_loss

save\_checkpoint(model, ckpt\_file\_name)

输出：



### 模型推理

首先，通过load\_checkpoint与load\_param\_into\_net将训练好的模型参数加载入新实例化的模型中。

from mindspore import load\_checkpoint, load\_param\_into\_net

encoder = Encoder(src\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.1)

decoder = Decoder(trg\_vocab\_size, d\_model, n\_heads, d\_ff, n\_layers, dropout\_p=0.1)

new\_model = Transformer(encoder, decoder)

param\_dict = load\_checkpoint(ckpt\_file\_name)

load\_param\_into\_net(new\_model, param\_dict)

推理过程中无需对模型参数进行更新，所以这里model.set\_train(False)。

我们输入一个德文语句，期望可以返回翻译好的英文语句。

首先通过Encoder提取德文序列中的特征信息，并将其传输至Decoder。Decoder最开始的输入为起始占位符<bos>，每次会根据输入预测下一个出现的单词，并对输入进行更新，直到预测出终止占位符<eos>。

def inference(sentence, max\_len=32):

"""模型推理：输入一个德语句子，输出翻译后的英文句子

enc\_inputs: [batch\_size(1), src\_len]

"""

new\_model.set\_train(False)

if isinstance(sentence, str):

tokens = [tok.lower() for tok in re.findall(r'\w+|[^\w\s]', sentence.rstrip())]

else:

tokens = [token.lower() for token in sentence]

if len(tokens) > max\_len - 2:

src\_len = max\_len

tokens = ['<bos>'] + tokens[:max\_len - 2] + ['<eos>']

else:

src\_len = len(tokens) + 2

tokens = ['<bos>'] + tokens + ['<eos>'] + ['<pad>'] \* (max\_len - src\_len)

indexes = de\_vocab.encode(tokens)

enc\_inputs = Tensor(indexes, mstype.float32).expand\_dims(0)

enc\_outputs, \_ = new\_model.encoder(enc\_inputs, src\_pad\_idx)

dec\_inputs = Tensor([[en\_vocab.bos\_idx]], mstype.float32)

max\_len = enc\_inputs.shape[1]

for \_ in range(max\_len):

dec\_outputs, \_, \_ = new\_model.decoder(dec\_inputs, enc\_inputs, enc\_outputs, src\_pad\_idx, trg\_pad\_idx)

dec\_logits = dec\_outputs.view((-1, dec\_outputs.shape[-1]))

dec\_logits = dec\_logits[-1, :]

pred = dec\_logits.argmax(axis=0).expand\_dims(0).expand\_dims(0)

pred = pred.astype(mstype.float32)

dec\_inputs = ops.concat((dec\_inputs, pred), axis=1)

if int(pred.asnumpy()[0]) == en\_vocab.eos\_idx:

break

trg\_indexes = [int(i) for i in dec\_inputs.view(-1).asnumpy()]

eos\_idx = trg\_indexes.index(en\_vocab.eos\_idx) if en\_vocab.eos\_idx in trg\_indexes else -1

trg\_tokens = en\_vocab.decode(trg\_indexes[1:eos\_idx])

return trg\_tokens

以测试数据集中的第一组语句为例，进行测试。

example\_idx = 0

src = test\_dataset[example\_idx][0]

trg = test\_dataset[example\_idx][1]

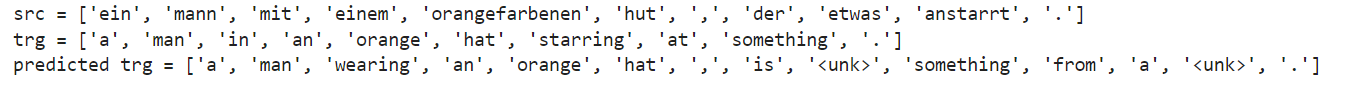
pred\_trg = inference(src)

print(f'src = {src}')

print(f'trg = {trg}')

print(f"predicted trg = {pred\_trg}")

输出：



### BLEU得分

双语替换评测得分（bilingual evaluation understudy，BLEU）为衡量文本翻译模型生成出来的语句好坏的一种算法，它的核心在于评估机器翻译的译文 predpred 与人工翻译的参考译文 labellabel 的相似度。通过对机器译文的片段与参考译文进行比较，计算出各个片段的的分数，并配以权重进行加和，基本规则为：

1、惩罚过短的预测，即如果机器翻译出来的译文相对于人工翻译的参考译文过于短小，则命中率越高，需要施加更多的惩罚；

2、对长段落匹配更高的权重，即如果出现长段落的完全命中，说明机器翻译的译文更贴近人工翻译的参考译文；

BLEU的公式如下：

* len(label)：人工翻译的译文长度
* len(pred)：机器翻译的译文长度
* p\_n：n-gram的精度

我们可以调用nltk中的corpus\_bleu函数来计算BLEU。

from nltk.translate.bleu\_score import corpus\_bleu

def calculate\_bleu(dataset, max\_len=50):

trgs = []

pred\_trgs = []

for data in dataset[:10]:

src = data[0]

trg = data[1]

pred\_trg = inference(src, max\_len)

pred\_trgs.append(pred\_trg)

trgs.append([trg])

return corpus\_bleu(trgs, pred\_trgs)

bleu\_score = calculate\_bleu(test\_dataset)

print(f'BLEU score = {bleu\_score\*100:.2f}')

输出：

BLEU score = 39.71

# 课后思考题

## 思考题一

基于Mindspore-Ascend的Transformer实现“中译英”。

## 思考题二

调整实验代码，实现基于Mindspore-Ascend的Transformer“英译德”翻译。