深度学习基础

实验四

版本：2.2



华为技术有限公司

|  |
| --- |
| **版权所有 © 华为技术有限公司 2022。 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**  C:\Users\jwx341670\Desktop\华为标志 Huawei Logo 2018\竖版标志Vertical Version\PNG\HW_POS_RBG_Vertical-150ppi.png 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 实验介绍 5](#_Toc95837731)

[1.1 实验目的 6](#_Toc95837732)

[1.2 实验清单 6](#_Toc95837733)

[1.3 开发平台介绍 6](#_Toc95837734)

[1.4 实验开发环境 6](#_Toc95837735)

[1.5 背景知识 6](#_Toc95837736)

[1.5.1 循环神经网络介绍 6](#_Toc95837742)

[1.5.2 注意力机制 9](#_Toc95837743)

[2 基于Mindspore的RNN-IMDB情感分析实验 10](#_Toc95837744)

[2.1 实验介绍 10](#_Toc95837745)

[2.2 实验环境要求 10](#_Toc95837746)

[2.2.1 创建实验环境 11](#_Toc95837747)

[2.3 实验总体设计 11](#_Toc95837748)

[2.4 实验过程 12](#_Toc95837749)

[2.4.1 实验准备 12](#_Toc95837755)

[2.4.2 导入数据集并预处理 14](#_Toc95837756)

[2.4.3 LSTM模型建模 21](#_Toc95837757)

[2.4.4 模型评估 26](#_Toc95837758)

[2.5 实验总结 27](#_Toc95837759)

[3 Seq2seq机器翻译实验 28](#_Toc95837760)

[3.1 实验介绍 28](#_Toc95837761)

[3.2 实验预备知识 28](#_Toc95837762)

[3.3 实验环境介绍 28](#_Toc95837763)

[3.4 实验总体设计 29](#_Toc95837764)

[3.5 实验操作 29](#_Toc95837765)

[3.6 代码详解 29](#_Toc95837766)

[3.6.1 config.py文件 29](#_Toc95837767)

[3.6.2 dataset.py文件 30](#_Toc95837768)

[3.6.3 preprocess.py文件 31](#_Toc95837769)

[3.6.4 seq2seq.py文件 33](#_Toc95837770)

[3.6.5 loss.py文件 37](#_Toc95837771)

[3.6.6 train.py文件 38](#_Toc95837772)

[3.6.7 eval.py文件 39](#_Toc95837773)

[3.6.8 main.ipynb文件 40](#_Toc95837774)

[3.7 实验总结 43](#_Toc95837781)

[3.8 思考题-汇总 43](#_Toc95837782)

[3.9 开放题（可选） 43](#_Toc95837783)

[4 附录：ModelArts开发环境搭建 44](#_Toc95837784)

# 实验介绍

循环神经网络为人工智能核心技术，本章主要围绕循环神经网络涉及的循环神经网络RNN、长短记忆网络LSTM和门控循环单元GRU而开设的实验,以及使用注意力机制做优化。本章实验难度分为中级和高级。

高级实验：Seq2seq机器翻译实验和RNN-IMDB情感分析实验

## 实验目的

本章实验的主要目的是掌握深度学习相关基础知识点，了解深度学习相关基础知识，经典全连接神经网络、卷积神经网络和对抗神经网络。掌握不同神经网络架构的设计原理，熟悉使用Mindspore深度学习框架实验深度学习实验的一般流程。

## 实验清单

表格：实验、简述、难度、软件环境、硬件环境。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **实验** | **简述** | **难度** | **软件环境** | **开发环境** |
| 基于Mindspore的RNN-IMDB情感分析实验 | 通过GloVe模型进行词嵌入，搭建LSTM网络，实现IMDB情感分析。 | 高级 | MindSpore-1.7  -Python3.7.5 | ModelArts-Ascend |
| Seq2seq机器翻译 | 基于seq2seq模型进行中英文的翻译。 | 高级 | Mindspore-1.7-python3.7 | ModelArts-Ascend |

## 开发平台介绍

参考课程ppt：《MindSpore架构介绍》、《MindSpore开发实践》。

## 实验开发环境

Mindspore

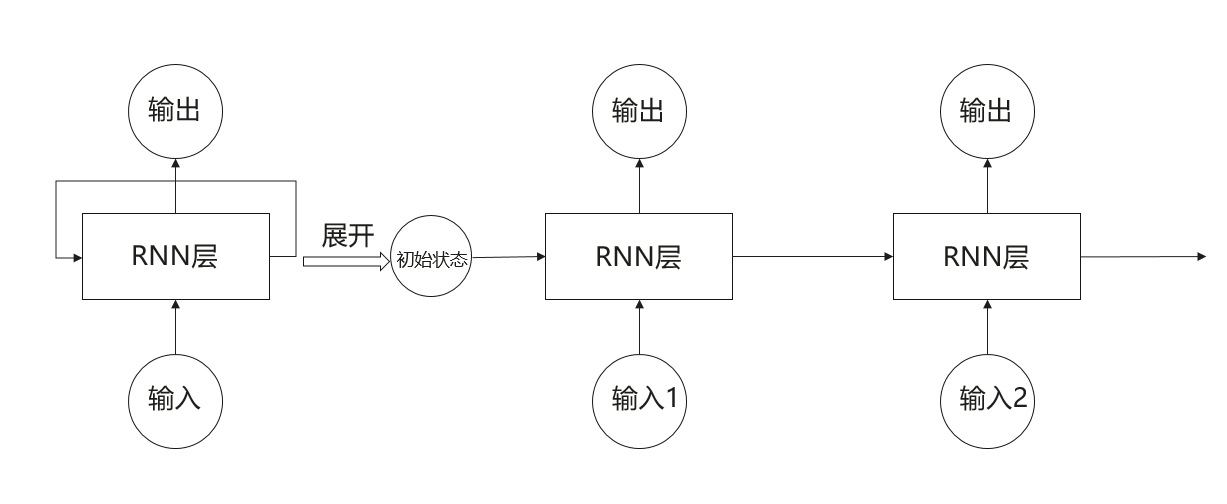
若选择在华为云ModelArts上快速搭建开发环境，可参考文末附录：ModelArts开发环境搭建。

## 背景知识



### 循环神经网络介绍

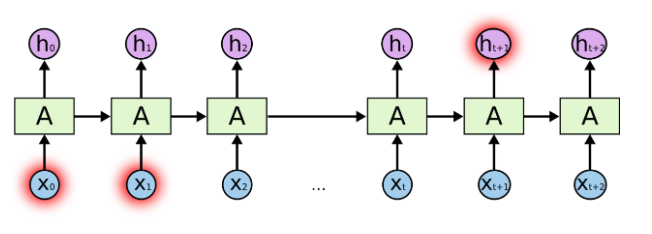
循环神经网络是深度学习与神经网络算法中主流算法之一，主要用于文本处理等与序列有关的内容，其结构如下：



循环神经网络结构图

与卷积神经网络不同，循环神经网络是图灵完备的，可以有源源不断的输出，从结构图中左侧可以看到，循环神经网络结束一个输入，产生一个输出，这一点与卷积神经网络没有区别；但是展开之后可以发现不同，右侧展开的长度与我们输入的序列长度一致，无需特意指定，比如我们输入“今天”、“天气”、“不错”，那么就会有三个对应输入1、输入2、输入3，然后第一个神经元接收了输入1之后除了产生一个对外输出，还会对后面的神经元产生一个输入，第二个神经元的输出就同时受输入1和输入2的影响。类似于卷积神经网络共享参数，循环神经网络也是共享参数的，上图右侧展开的每一个神经元内部参数是一样的，可以看做是同一神经网络结构被无限复制的结果，从而可以使用有限的参数处理任意长度的序列。

但是RNN有长期依赖问题（Long-Term Dependency），如图1-2所示，当相关信息，和当前预测位置之间得间隔不断增大时，RNN就会丧失学习到连接如此远的信息的能力。

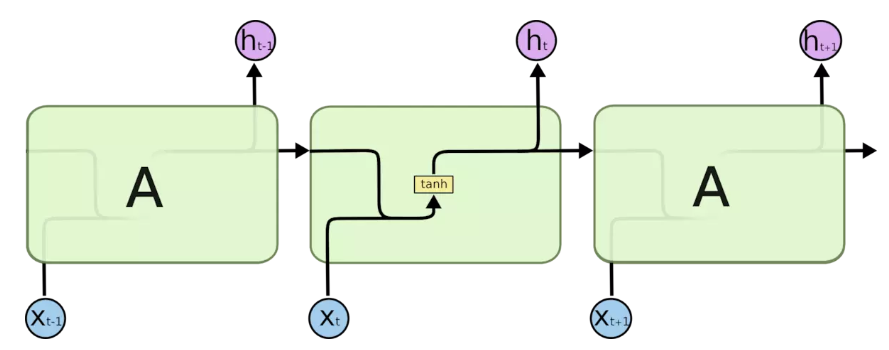


长度依赖问题

长短期记忆（LSTM）

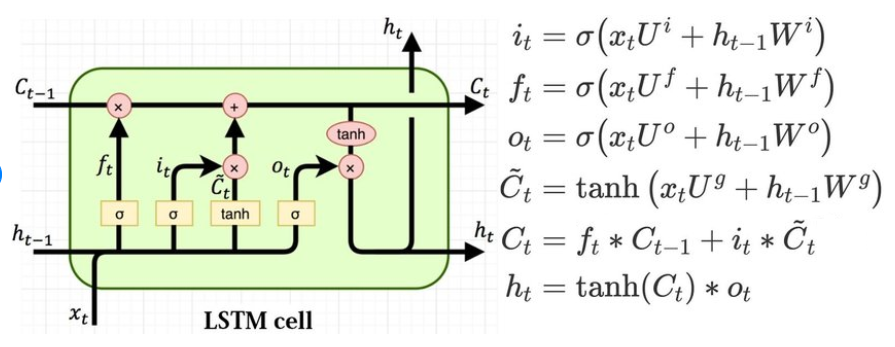
长短期记忆（Long Short-Term Memory, 简称 LSTM）是一种 RNN 特殊的类型，可以学习长期依赖信息。LSTM 由Hochreiter & Schmidhuber (1997)提出，并在近期被Alex Graves进行了改良和推广。在很多问题，LSTM 都取得相当巨大的成功，并得到了广泛的使用。LSTM 通过刻意的设计来避免长期依赖问题。记住长期的信息在实践中是 LSTM 的默认行为，而非需要付出很大代价才能获得的能力！

所有 RNN 都具有一种重复神经网络模块的链式的形式。在标准的 RNN 中，这个重复的模块只有一个非常简单的结构，例如一个 tanh 层。



标准RNN解析图

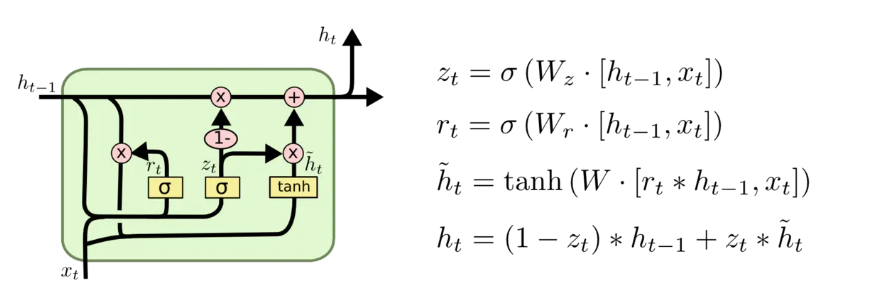
LSTM 同样是这样的结构，但是重复的模块拥有一个不同的结构。如图1-4所示，不同于单一神经网络层，LSTM以一种非常特殊的方式进行交互运算。LSTM 有通过精心设计的称作为“门”的结构来去除或者增加信息到细胞状态的能力。门是一种让信息选择式通过的方法，包含一个 sigmoid 神经网络层和一个按位的乘法操作。Sigmoid 层输出 0 到 1 之间的数值，描述每个部分有多少量可以通过。0 代表“不许任何量通过”，1 就指“允许任意量通过”。



LSTM解析图

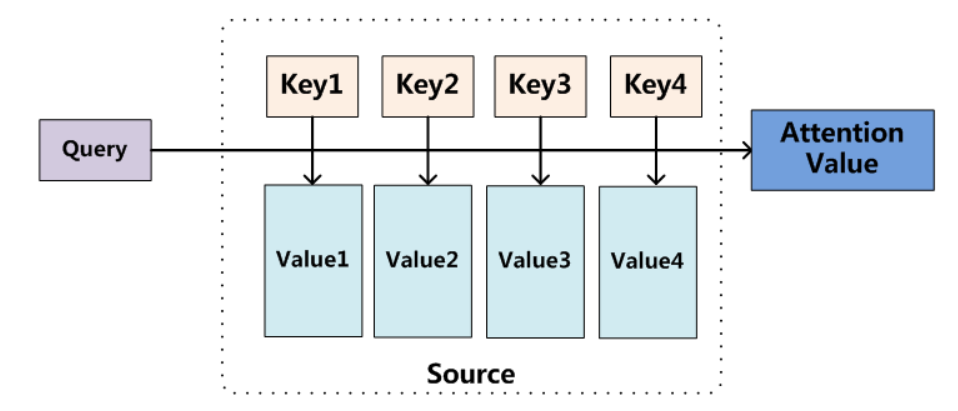
GRU

LSTM会做些许改动，改动较大的变体是 Gated Recurrent Unit (GRU)，这是由 Cho, et al. (2014) 提出。如图1-5所示，最终的模型比标准的 LSTM 模型要简单，也是非常流行的变体。



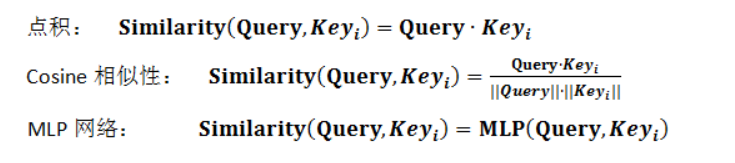
GRU解析图

### 注意力机制

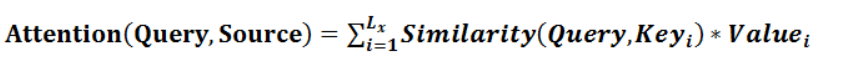


注意力机制

注意力机制Attention的本质思想是将Source中的构成元素想象成是由一系列的<Key,Value>数据对构成，此时给定Target中的某个元素Query，通过计算Query和各个Key的相似性或者相关性，得到每个Key对应Value的权重系数，然后对Value进行加权求和，即得到了最终的Attention数值。计算两者的相似性和相关性，常见的方法包括：求两者的向量点积、求两者的向量Cosine相似性或者通过再引入额外的神经网络来求值，即如下方式：



所以本质上Attention机制是对Source中元素的Value值进行加权求和，而Query和Key用来计算对应Value的权重系数。即可以将其本质思想改写为如下公式：



其中，=||Source||代表Source的长度，公式含义即如上所述。从概念上理解，把Attention仍然理解为从大量信息中有选择地筛选出少量的重要信息并聚焦到这些重要信息上，忽略大多不重要的信息。聚焦的过程体现在权重系数的计算上，权重越大越聚焦于其对应的Value值上，即权重代表了信息的重要性，而Value是其对应的信息。

# 基于Mindspore的RNN-IMDB情感分析实验

## 实验介绍

IMDB是一个与国内豆瓣比较类似的与电影相关的网站，而本次实验用到的数据集是这个网站中的一些用户评论。IMDB数据集共包含50000项影评文字，训练数据和测试数据各25000项，每一项影评文字都被标记为正面评价或负面评价，所以本实验可以看做一个基于MindSpore的二分类任务。

实验数据：

从华为云对象存储服务（OBS）获取

华为云开通了相应的数据存储服务OBS可直接通过链接进行数据集下载。

数据集链接：<https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/aclImdb_v1.tar.gz>

从斯坦福大学网站获取

数据集链接：<http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz>

实验模型：

该实验会使用GloVe作为实验所需的预训练模型，用于词嵌入，提取词的特征。GloVe的全称叫Global Vectors for Word Representation，它是一个基于全局词频统计（count-based & overall statistics）的词表征（word representation）工具，它可以把一个单词表达成一个由实数组成的向量，这些向量捕捉到了单词之间一些语义特性，比如相似性（similarity）、类比性（analogy）等。我们通过对向量的运算，比如欧几里得距离或者cosine相似度，可以计算出两个单词之间的语义相似性。

从华为云对象存储服务（OBS）获取

华为云开通了相应的数据存储服务OBS可直接通过链接进行模型下载。

模型链接：

https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/glove.zip

本实验的数据和模型会在实验代码里进行下载。

## 实验环境要求

ModelArts平台：Mindspore-python3.7-aarch64

### 创建实验环境

进入ModelArts开发环境

参考文末附录，创建ModelArts上的开发环境Notebook并进入。

在Notebook中下载数据集和代码

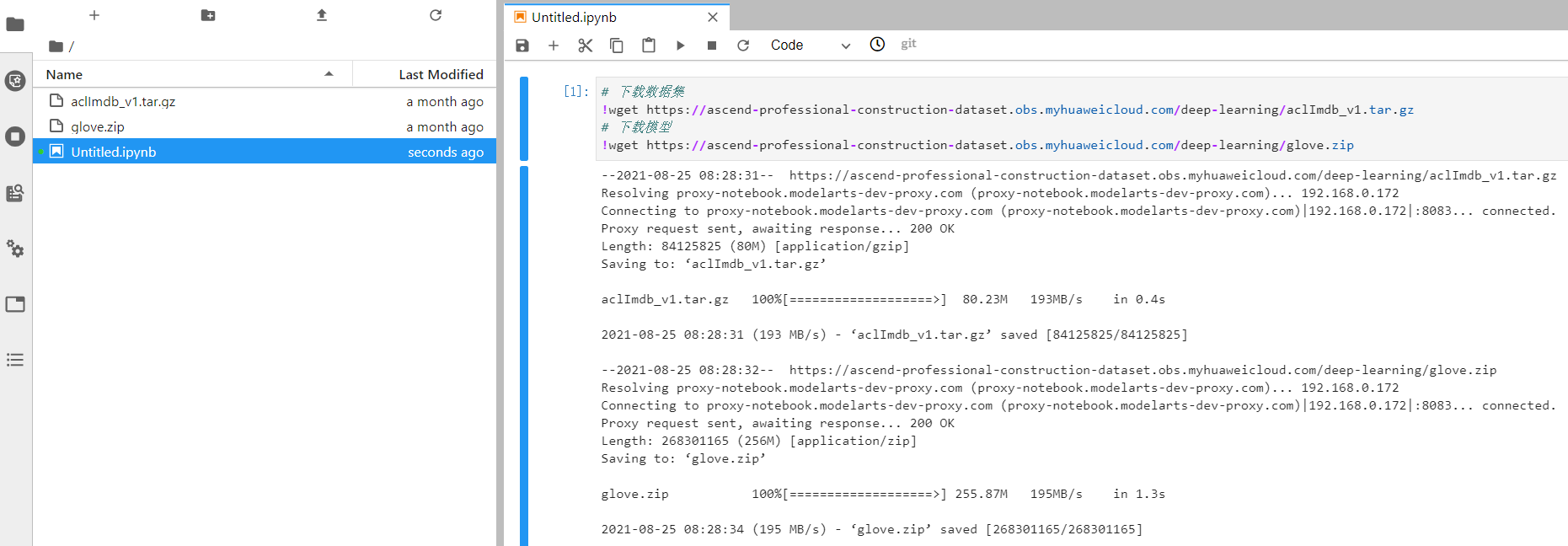
由于原始文件比较大，因此需要在Notebook环境中下载数据集aclImdb\_v1.tar.gz和代码文件glove.zip。

# 下载数据集

!wget https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/aclImdb\_v1.tar.gz

# 下载模型

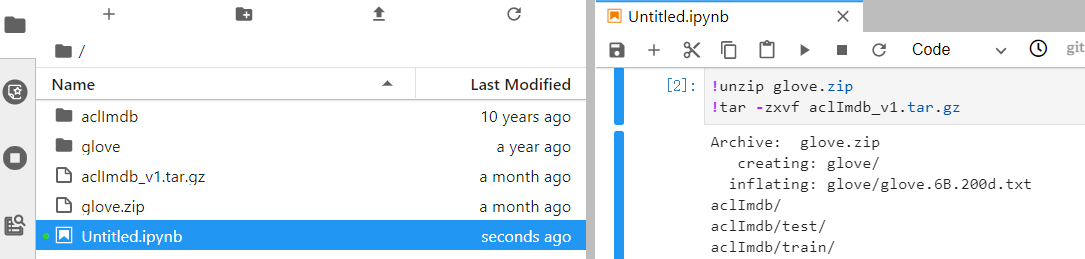
!wget https://ascend-professional-construction-dataset.obs.myhuaweicloud.com/deep-learning/glove.zip



解压数据和代码文件

!unzip glove.zip

!tar -zxvf aclImdb\_v1.tar.gz



## 实验总体设计

这里我们将学习一种特殊的循环神经网络—长短记忆网络（LSTM），与卷积神经网络不同，循环神经网络可以把前一个时序的输出作为输入的一部分，以此来产生新的输出，是图灵完备的。本实验主要介绍使用MindSpore在Ascend环境下训练并测试LSTM模型。

该实验主要步骤包括：

导入实验所需模块

加载数据集，进行数据处理。

定义网络。

定义优化器和损失函数。

使用网络训练数据，生成模型。

得到模型之后，使用验证数据集，查看模型精度情况。

## 实验过程



### 实验准备

下载实验所需模块gensim

创建Notebook，出入如下命令下载gensim模块。

!pip install gensim

导入实验所需模块

os模块主要用于对系统路径和文件进行处理。shutil模块主要用于文件夹删除等操作。Numpy模块主要用于数据的基本运算操作。config模块是配置基于LSTM构建SentimentNet网络所需相关参数。MindSpore相关模块主要用于搭建网络、调用优化器、读取数据集和将数据集处理成网络的标准输入格式。

import os

import math

import gensim

import argparse

import numpy as np

import mindspore.dataset as ds

import mindspore.ops as ops

from itertools import chain

from easydict import EasyDict as edict

from mindspore import Model

from mindspore import Tensor, nn, context, Parameter, ParameterTuple

from mindspore.nn import Accuracy

from mindspore.mindrecord import FileWriter

from mindspore.common.initializer import initializer

from mindspore.train.serialization import load\_checkpoint, load\_param\_into\_net

from mindspore.train.callback import CheckpointConfig, ModelCheckpoint, TimeMonitor, LossMonitor

删除文件夹

为了避免多次运行程序发现错误，需删除前一次运行程序所生成的文件夹：

import shutil

if os.path.exists('./preprocess'):

shutil.rmtree('./preprocess')

传入必要信息

使用parser模块，传入运行必要的信息，如数据集存放路径，GloVe存放路径，这样的好处是，对于经常变化的配置，可以在运行代码时输入，使用更加灵活。

preprocess：是否预处理数据集，默认为否。

aclimdb\_path：数据集存放路径。

glove\_path：GloVe文件存放路径。

preprocess\_path：预处理数据集的结果文件夹。

ckpt\_path：CheckPoint文件路径。

pre\_trained：预加载CheckPoint文件。

device\_target：指定Ascend环境。

parser = argparse.ArgumentParser(description='MindSpore LSTM Example')

parser.add\_argument('--preprocess', type=str, default='false', choices=['true', 'false'],

help='whether to preprocess data.')

parser.add\_argument('--aclimdb\_path', type=str, default="./aclImdb",

help='path where the dataset is stored.')

parser.add\_argument('--glove\_path', type=str, default="./glove",

help='path where the GloVe is stored.')

parser.add\_argument('--preprocess\_path', type=str, default="./preprocess",

help='path where the pre-process data is stored.')

parser.add\_argument('--ckpt\_path', type=str, default="./",

help='the path to save the checkpoint file.')

parser.add\_argument('--pre\_trained', type=str, default=None,

help='the pretrained checkpoint file path.')

parser.add\_argument('--device\_target', type=str, default="Ascend", choices=['GPU', 'CPU','Ascend'],

help='the target device to run, support "GPU", "CPU","Ascend". Default: "Ascend".')

args = parser.parse\_args(['--device\_target', 'Ascend', '--preprocess', 'true'])

# 设置运行环境，若需要保持IR，可以设置save\_graphs=True，device\_target用于指定运行的硬件。

context.set\_context(

mode=context.GRAPH\_MODE,

save\_graphs=False,

device\_target='Ascend')

定义超参数

lstm\_cfg = edict({

# 分类类别为2

'num\_classes': 2,

# 学习率

'learning\_rate': 0.1,

# 动量系数

'momentum': 0.9,

# 训练次数

'num\_epochs': 10,

# 批大小

'batch\_size': 32,

# 词向量长度为200

'embed\_size': 200,

# 隐藏层神经元数量

'num\_hiddens': 128,

# 隐藏层数量

'num\_layers': 1,

# 是否双向，建议单向，双向训练会花费很长时间

'bidirectional': False,

# 多少次训练保存模型

'save\_checkpoint\_steps': 390\*5,

'keep\_checkpoint\_max': 10

})

cfg = lstm\_cfg

### 导入数据集并预处理

本次实验采用IMDB影评数据集作为实验数据。在网上下载数据集放到当前工作目录下；下载glove文件并解压glove文件到当前工作目录下，该模型总共读取400000个单词，每个单词压缩成200维度的词向量。

数据集预处理

定义ImdbParser类解析文本数据集，包括编码、分词、对齐、处理GloVe原始数据，使之能够适应网络结构。

class ImdbParser():

"""

将aclImdb数据解析为特征和标签。

sentence->tokenized->encoded->padding->features

"""

def \_\_init\_\_(self, imdb\_path, glove\_path, embed\_size=200):

# 分训练集和测试集

self.\_\_segs = ['train', 'test']

# 正例为1，负例为0

self.\_\_label\_dic = {'pos': 1, 'neg': 0}

# 设置数据集路径，词向量长度，模型路径

self.\_\_imdb\_path = imdb\_path

self.\_\_glove\_dim = embed\_size

self.\_\_glove\_file = os.path.join(glove\_path, 'glove.6B.' + str(self.\_\_glove\_dim) + 'd.txt')

# 属性

self.\_\_imdb\_datas = {}

self.\_\_features = {}

self.\_\_labels = {}

self.\_\_vacab = {}

self.\_\_word2idx = {}

self.\_\_weight\_np = {}

self.\_\_wvmodel = None

def parse(self):

"""

将imdb数据解析到内存中

"""

self.\_\_wvmodel = gensim.models.KeyedVectors.load\_word2vec\_format(self.\_\_glove\_file)

for seg in self.\_\_segs:

self.\_\_parse\_imdb\_datas(seg)

self.\_\_parse\_features\_and\_labels(seg)

self.\_\_gen\_weight\_np(seg)

def \_\_parse\_imdb\_datas(self, seg):

"""

从模型 txt文件读取数据

"""

data\_lists = []

for label\_name, label\_id in self.\_\_label\_dic.items():

sentence\_dir = os.path.join(self.\_\_imdb\_path, seg, label\_name)

for file in os.listdir(sentence\_dir):

# 编码格式位uft8，因此在编译glove文件的时候也需要按照utf8的格式保存。

with open(os.path.join(sentence\_dir, file), mode='r', encoding='utf8') as f:

sentence = f.read().replace('\n', '')

data\_lists.append([sentence, label\_id])

self.\_\_imdb\_datas[seg] = data\_lists

def \_\_parse\_features\_and\_labels(self, seg):

"""

解析特征和标签

"""

features = []

labels = []

for sentence, label in self.\_\_imdb\_datas[seg]:

features.append(sentence)

labels.append(label)

self.\_\_features[seg] = features

self.\_\_labels[seg] = labels

# 更新特性到tokenized

self.\_\_updata\_features\_to\_tokenized(seg)

# 解析vacab

self.\_\_parse\_vacab(seg)

# 编码特征

self.\_\_encode\_features(seg)

# padding

self.\_\_padding\_features(seg)

def \_\_updata\_features\_to\_tokenized(self, seg):

tokenized\_features = []

for sentence in self.\_\_features[seg]:

# 把所有字母转化为小写，并且以空格作为分隔符，该实验使用单词作为token。

tokenized\_sentence = [word.lower() for word in sentence.split(" ")]

tokenized\_features.append(tokenized\_sentence)

self.\_\_features[seg] = tokenized\_features

def \_\_parse\_vacab(self, seg):

# 解析词汇

tokenized\_features = self.\_\_features[seg]

vocab = set(chain(\*tokenized\_features))

self.\_\_vacab[seg] = vocab

# 单词标号，例如: {'hello': 1, 'world':111, ... '<unk>': 0}

word\_to\_idx = {word: i + 1 for i, word in enumerate(vocab)}

# 在测试集中未遇到的单词会被归为<unk>，在字典中标号为0

word\_to\_idx['<unk>'] = 0

self.\_\_word2idx[seg] = word\_to\_idx

def \_\_encode\_features(self, seg):

""" 将字编码到索引 """

word\_to\_idx = self.\_\_word2idx['train']

encoded\_features = []

for tokenized\_sentence in self.\_\_features[seg]:

encoded\_sentence = []

for word in tokenized\_sentence:

encoded\_sentence.append(word\_to\_idx.get(word, 0))

encoded\_features.append(encoded\_sentence)

self.\_\_features[seg] = encoded\_features

def \_\_padding\_features(self, seg, maxlen=200, pad=0):

""" 将所有特征填充到相同长度，默认指定长度为200 """

padded\_features = []

for feature in self.\_\_features[seg]:

if len(feature) >= maxlen:

# 如果一条评论的长度大于指定长度，则取前几个单词，大于指定长度的直接不要了。

padded\_feature = feature[:maxlen]

else:

# 如果一条评论的长度小于指定长度，在评论的末尾填充0直到指定长度，0在字典中的意思是<unk>

padded\_feature = feature

while len(padded\_feature) < maxlen:

padded\_feature.append(pad)

padded\_features.append(padded\_feature)

self.\_\_features[seg] = padded\_features

def \_\_gen\_weight\_np(self, seg):

"""

通过gensim产生权重

"""

weight\_np = np.zeros((len(self.\_\_word2idx[seg]), self.\_\_glove\_dim), dtype=np.float32)

for word, idx in self.\_\_word2idx[seg].items():

if word not in self.\_\_wvmodel:

continue

word\_vector = self.\_\_wvmodel.get\_vector(word)

weight\_np[idx, :] = word\_vector

self.\_\_weight\_np[seg] = weight\_np

def get\_datas(self, seg):

"""

返回特征，标签和权重

"""

features = np.array(self.\_\_features[seg]).astype(np.int32)

labels = np.array(self.\_\_labels[seg]).astype(np.int32)

weight = np.array(self.\_\_weight\_np[seg])

return features, labels, weight

数据集转换为MindRecord格式

函数\_convert\_to\_mindrecord中weight.txt为数据预处理后自动生成的weight参数信息文件。

定义convert\_to\_mindrecord函数将数据集格式转换为MindRecord格式，便于MindSpore读取。

def \_convert\_to\_mindrecord(data\_home, features, labels, weight\_np=None, training=True):

"""

将IMDB转化为mindrecord数据

"""

if weight\_np is not None:

np.savetxt(os.path.join(data\_home, 'weight.txt'), weight\_np)

# write mindrecord

schema\_json = {"id": {"type": "int32"},

"label": {"type": "int32"},

"feature": {"type": "int32", "shape": [-1]}}

data\_dir = os.path.join(data\_home, "aclImdb\_train.mindrecord")

if not training:

data\_dir = os.path.join(data\_home, "aclImdb\_test.mindrecord")

def get\_imdb\_data(features, labels):

# 从IMDB数据集从读取数据

data\_list = []

for i, (label, feature) in enumerate(zip(labels, features)):

data\_json = {"id": i,

"label": int(label),

"feature": feature.reshape(-1)}

data\_list.append(data\_json)

return data\_list

writer = FileWriter(data\_dir, shard\_num=4)

data = get\_imdb\_data(features, labels)

writer.add\_schema(schema\_json, "nlp\_schema")

writer.add\_index(["id", "label"])

writer.write\_raw\_data(data)

writer.commit()

def convert\_to\_mindrecord(embed\_size, aclimdb\_path, preprocess\_path, glove\_path):

"""

将IMDB数据集转化为mindrecord数据格式

"""

parser = ImdbParser(aclimdb\_path, glove\_path, embed\_size)

parser.parse()

if not os.path.exists(preprocess\_path):

# 如果preprocess文件夹不存在，则新建文件夹

print(f"preprocess path {preprocess\_path} is not exist")

os.makedirs(preprocess\_path)

# 训练集

train\_features, train\_labels, train\_weight\_np = parser.get\_datas('train')

\_convert\_to\_mindrecord(preprocess\_path, train\_features, train\_labels, train\_weight\_np)

#

test\_features, test\_labels, \_ = parser.get\_datas('test')

\_convert\_to\_mindrecord(preprocess\_path, test\_features, test\_labels, training=False)

调用convert\_to\_mindrecord函数将数据集格式转换为MindRecord格式。

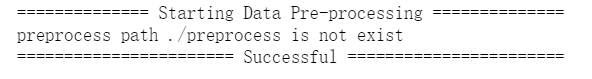
if args.preprocess == "true":

print("============== Starting Data Pre-processing ==============")

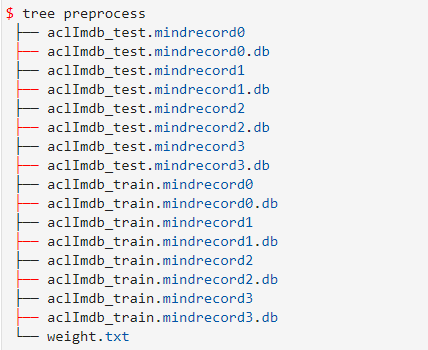
convert\_to\_mindrecord(cfg.embed\_size, args.aclimdb\_path, args.preprocess\_path, args.glove\_path)

print("======================= Successful =======================")

输出结果：



转换成功后会在preprocess目录下生成MindRecord文件，通常该操作在数据集不变的情况下，无需每次训练都执行，此时preprocess文件目录如下所示：



以上各文件中：

名称包含aclImdb\_train.mindrecord为转换后的MindRecord格式的训练数据集。

名称包含aclImdb\_test.mindrecord为转换后的MindRecord格式的测试数据集。

weight.txt为预处理后自动生成的weight参数信息文件。

创建训练集

定义创建数据集函数lstm\_create\_dataset，创建训练集ds\_train。

def lstm\_create\_dataset(data\_home, batch\_size, repeat\_num=1, training=True):

"""数据操作"""

# 设置种子，伪随机

ds.config.set\_seed(1)

data\_dir = os.path.join(data\_home, "aclImdb\_train.mindrecord0")

if not training:

data\_dir = os.path.join(data\_home, "aclImdb\_test.mindrecord0")

data\_set = ds.MindDataset(data\_dir, columns\_list=["feature", "label"], num\_parallel\_workers=4)

# 数据洗牌

data\_set = data\_set.shuffle(buffer\_size=data\_set.get\_dataset\_size())

# 数据分批

data\_set = data\_set.batch(batch\_size=batch\_size, drop\_remainder=True)

data\_set = data\_set.repeat(count=repeat\_num)

return data\_set

ds\_train = lstm\_create\_dataset(args.preprocess\_path, cfg.batch\_size)

ds\_eval = lstm\_create\_dataset(args.preprocess\_path, cfg.batch\_size, training=False)

打印预处理后的数据

打印经过预处理的数据，打印一个batch中的标签以及该batch中第一条数据的特征。

# 打印第一批的标签以及第一批第一个数据的特征

iterator = ds\_train.create\_dict\_iterator().\_get\_next()

first\_batch\_label = iterator["label"].asnumpy()

first\_batch\_first\_feature = iterator["feature"].asnumpy()[0]

print(f"The first batch contains label below:\n{first\_batch\_label}\n")

print(f"The feature of the first item in the first batch is below vector:\n{first\_batch\_first\_feature}")

输出结果：

可以看到该实验中，规定了每条数据的长度为200，若该条数据的词数大于200，将截取最后200个词，若该条数据的词数小于200，将会在后面补0。

The first batch contains label below:

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

The feature of the first item in the first batch is below vector:

[ 48279 236021 161257 50881 133197 163821 7632 240767 154912 11213

18002 142178 236021 244063 233488 213225 2703 33497 7632 240767

236021 133197 128255 152310 192793 139944 120529 154385 79370 167304

112392 106818 245320 111306 45225 22418 192621 127774 245320 111306

102311 240767 8611 192621 27226 226181 75201 240767 245441 105177

177813 197017 3278 180122 48279 236021 127905 161257 110503 25678

136070 79370 45173 196677 220623 192621 240767 239270 74125 33756

217408 161257 12095 175851 7632 14460 35179 86130 133197 239270

25134 108842 43094 128313 113337 231263 158565 131063 12095 128475

86743 77052 217408 4243 210354 246703 226181 195563 133197 29456

226763 216613 207971 169103 79370 167304 83761 34132 240767 65124

226181 240767 14070 139944 161257 248165 139944 161257 69855 226181

237465 226181 139944 59963 192621 240767 216678 201184 226181 170778

240767 77811 7632 240767 94721 79370 167304 110503 139712 32448

240767 55623 186527 161257 240767 154176 7632 139944 43956 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0]

### LSTM模型建模

初始化网络参数及网络状态

定义lstm\_default\_state函数来初始化网络参数及网络状态。

# 定义需要单层LSTM小算子堆叠的设备类型。

STACK\_LSTM\_DEVICE = ["CPU"]

# 短期内存（h）和长期内存（c）初始化为0

# 定义lstm\_default\_state函数来初始化网络参数及网络状态。

def lstm\_default\_state(batch\_size, hidden\_size, num\_layers, bidirectional):

"""初始化默认输入."""

num\_directions = 2 if bidirectional else 1

h = Tensor(np.zeros((num\_layers \* num\_directions, batch\_size, hidden\_size)).astype(np.float32))

c = Tensor(np.zeros((num\_layers \* num\_directions, batch\_size, hidden\_size)).astype(np.float32))

return h, c

def stack\_lstm\_default\_state(batch\_size, hidden\_size, num\_layers, bidirectional):

"""init default input."""

num\_directions = 2 if bidirectional else 1

h\_list = c\_list = []

for \_ in range(num\_layers):

h\_list.append(Tensor(np.zeros((num\_directions, batch\_size, hidden\_size)).astype(np.float32)))

c\_list.append(Tensor(np.zeros((num\_directions, batch\_size, hidden\_size)).astype(np.float32)))

h, c = tuple(h\_list), tuple(c\_list)

return h, c

# 针对不同的场景，自定义单层LSTM小算子堆叠，来实现多层LSTM大算子功能。

class StackLSTM(nn.Cell):

"""

Stack multi-layers LSTM together.

"""

def \_\_init\_\_(self,

input\_size,

hidden\_size,

num\_layers=1,

has\_bias=True,

batch\_first=False,

dropout=0.0,

bidirectional=False):

super(StackLSTM, self).\_\_init\_\_()

self.num\_layers = num\_layers

self.batch\_first = batch\_first

self.transpose = ops.Transpose()

# direction number

num\_directions = 2 if bidirectional else 1

# input\_size list

input\_size\_list = [input\_size]

for i in range(num\_layers - 1):

input\_size\_list.append(hidden\_size \* num\_directions)

# layers

layers = []

for i in range(num\_layers):

layers.append(nn.LSTMCell(input\_size=input\_size\_list[i],

hidden\_size=hidden\_size,

has\_bias=has\_bias,

batch\_first=batch\_first,

bidirectional=bidirectional,

dropout=dropout))

# weights

weights = []

for i in range(num\_layers):

# weight size

weight\_size = (input\_size\_list[i] + hidden\_size) \* num\_directions \* hidden\_size \* 4

if has\_bias:

bias\_size = num\_directions \* hidden\_size \* 4

weight\_size = weight\_size + bias\_size

# numpy weight

stdv = 1 / math.sqrt(hidden\_size)

w\_np = np.random.uniform(-stdv, stdv, (weight\_size, 1, 1)).astype(np.float32)

# lstm weight

weights.append(Parameter(initializer(Tensor(w\_np), w\_np.shape), name="weight" + str(i)))

#

self.lstms = layers

self.weight = ParameterTuple(tuple(weights))

def construct(self, x, hx):

"""construct"""

if self.batch\_first:

x = self.transpose(x, (1, 0, 2))

# stack lstm

h, c = hx

hn = cn = None

for i in range(self.num\_layers):

x, hn, cn, \_, \_ = self.lstms[i](x, h[i], c[i], self.weight[i])

if self.batch\_first:

x = self.transpose(x, (1, 0, 2))

return x, (hn, cn)

创建网络

使用Cell方法，定义网络结构（SentimentNet网络）。

class SentimentNet(nn.Cell):

"""定义网络结构"""

def \_\_init\_\_(self,

vocab\_size,

embed\_size,

num\_hiddens,

num\_layers,

bidirectional,

num\_classes,

weight,

batch\_size):

super(SentimentNet, self).\_\_init\_\_()

# 将词映射成词向量

self.embedding = nn.Embedding(vocab\_size,

embed\_size,

embedding\_table=weight)

self.embedding.embedding\_table.requires\_grad = False

self.trans = ops.Transpose()

self.perm = (1, 0, 2)

# 该实验用了LSTM网络

if context.get\_context("device\_target") in STACK\_LSTM\_DEVICE:

# stack lstm by user

self.encoder = StackLSTM(input\_size=embed\_size,

hidden\_size=num\_hiddens,

num\_layers=num\_layers,

has\_bias=True,

bidirectional=bidirectional,

dropout=0.0)

self.h, self.c = stack\_lstm\_default\_state(batch\_size, num\_hiddens, num\_layers, bidirectional)

else:

# standard lstm

self.encoder = nn.LSTM(input\_size=embed\_size,

hidden\_size=num\_hiddens,

num\_layers=num\_layers,

has\_bias=True,

bidirectional=bidirectional,

dropout=0.0)

self.h, self.c = lstm\_default\_state(batch\_size, num\_hiddens, num\_layers, bidirectional)

self.concat = ops.Concat(1)

# LSTM层后接一个全连接层用于分类

if bidirectional:

self.decoder = nn.Dense(num\_hiddens \* 4, num\_classes)

else:

self.decoder = nn.Dense(num\_hiddens \* 2, num\_classes)

def construct(self, inputs):

# input：(64,500,300)

embeddings = self.embedding(inputs)

embeddings = self.trans(embeddings, self.perm)

output, \_ = self.encoder(embeddings, (self.h, self.c))

# states[i] size(64,200) -> encoding.size(64,400)

encoding = self.concat((output[0], output[199]))

outputs = self.decoder(encoding)

return outputs

调用函数构建网络。

embedding\_table = np.loadtxt(os.path.join(args.preprocess\_path, "weight.txt")).astype(np.float32)

network = SentimentNet(vocab\_size=embedding\_table.shape[0],

embed\_size=cfg.embed\_size,

num\_hiddens=cfg.num\_hiddens,

num\_layers=cfg.num\_layers,

bidirectional=cfg.bidirectional,

num\_classes=cfg.num\_classes,

weight=Tensor(embedding\_table),

batch\_size=cfg.batch\_size)

定义损失函数以及优化器

# 使用交叉熵损失（带softmax）作为损失函数

loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# 使用动量优化器

opt = nn.Momentum(network.trainable\_params(), cfg.learning\_rate, cfg.momentum)

定义EvalCallBack类用于网络训练

from mindspore.train.callback import Callback

class EvalCallBack(Callback):

def \_\_init\_\_(self, model, eval\_dataset, eval\_per\_epoch, epoch\_per\_eval):

self.model = model

self.eval\_dataset = eval\_dataset

self.eval\_per\_epoch = eval\_per\_epoch

self.epoch\_per\_eval = epoch\_per\_eval

def epoch\_end(self, run\_context):

cb\_param = run\_context.original\_args()

cur\_epoch = cb\_param.cur\_epoch\_num

if cur\_epoch % self.eval\_per\_epoch == 0:

acc = self.model.eval(self.eval\_dataset, dataset\_sink\_mode=False)

self.epoch\_per\_eval["epoch"].append(cur\_epoch)

self.epoch\_per\_eval["acc"].append(acc["acc"])

print(acc) # 用于打印每个epoch的验证准确率

训练模型并保存

将创建好的网络、损失函数、评估指标、优化器等参数装入模型中对模型进行训练。加载训练数据集（`ds\_train`）并配置好`CheckPoint`生成信息，然后使用`model.train`接口，进行模型训练；根据输出可以看到loss值随着训练逐步降低，最后达到0.3-0.4左右。

# 训练，验证模型

import time

model = Model(network, loss, opt, {'acc': Accuracy()})

loss\_cb = LossMonitor()

print("============== Starting Training ==============")

start = time.time()# 记录训练，验证所花费的时间

config\_ck = CheckpointConfig(save\_checkpoint\_steps=ds\_train.get\_dataset\_size(),

keep\_checkpoint\_max=cfg.keep\_checkpoint\_max)

ckpoint\_cb = ModelCheckpoint(prefix="lstm", directory=args.ckpt\_path, config=config\_ck)

time\_cb = TimeMonitor(data\_size=ds\_train.get\_dataset\_size())

if args.device\_target == "CPU":

epoch\_per\_eval = {"epoch": [], "acc": []}

eval\_cb = EvalCallBack(model, ds\_eval, 1, epoch\_per\_eval)

model.train(cfg.num\_epochs, ds\_train, callbacks=[time\_cb, ckpoint\_cb, loss\_cb,eval\_cb], dataset\_sink\_mode=False)

else:

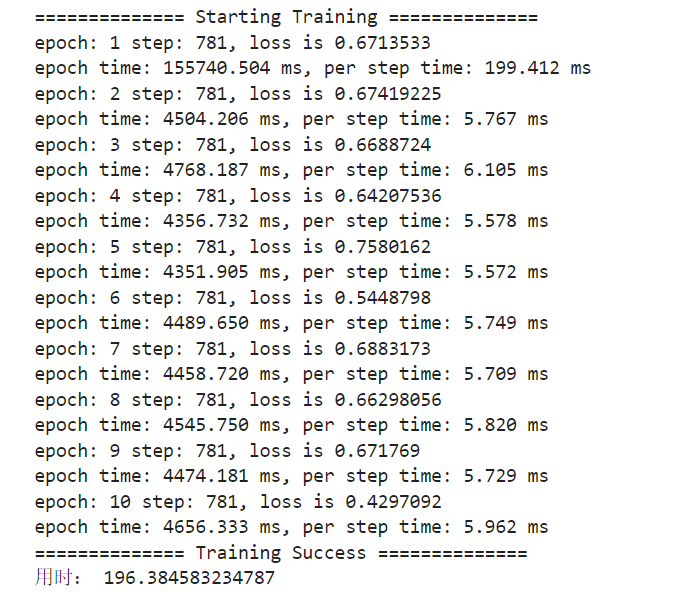
model.train(cfg.num\_epochs, ds\_train, callbacks=[time\_cb, ckpoint\_cb, loss\_cb])

print("============== Training Success ==============")

end = time.time()

print("用时：",end-start)

输出结果：



### 模型评估

创建并加载验证数据集（`ds\_eval`），加载由训练保存的CheckPoint文件，进行验证，查看模型质量。

# 测试模型

args.ckpt\_path = f'./lstm-{cfg.num\_epochs}\_781.ckpt'

print("============== Starting Testing ==============")

ds\_eval = lstm\_create\_dataset(args.preprocess\_path, cfg.batch\_size, training=False)

param\_dict = load\_checkpoint(args.ckpt\_path)

load\_param\_into\_net(network, param\_dict)

if args.device\_target == "CPU":

acc = model.eval(ds\_eval, dataset\_sink\_mode=False)

else:

acc = model.eval(ds\_eval)

print("============== {} ==============".format(acc))

输出结果：



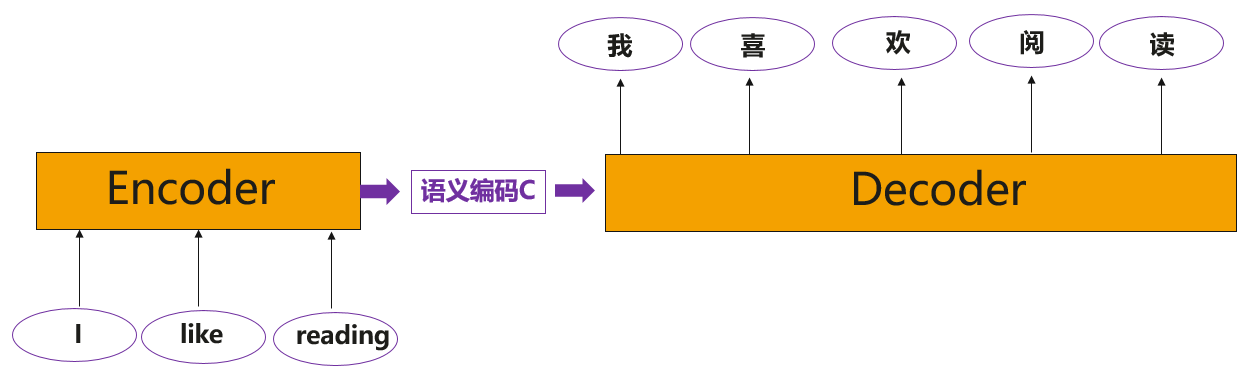
## 实验总结

本章提供了一个基于MindSpore框架的情感分析实验。通过本次体验全面了解了如何使用MindSpore进行自然语言中处理情感分类问题，理解了如何通过定义和初始化基于LSTM的SentimentNet网络进行训练模型及验证正确率。

# Seq2seq机器翻译实验

## 实验介绍

翻译任务在日常生活应用广泛，如手机中有各种翻译软件，可以满足人们交流、阅读的需求。本实验基于seq2seq编码器-解码器框架，结合GRU单元实现英文转中文的翻译任务，框架示意图如下：



编码器-解码器框架示意图

## 实验预备知识

有文本预处理的基础。

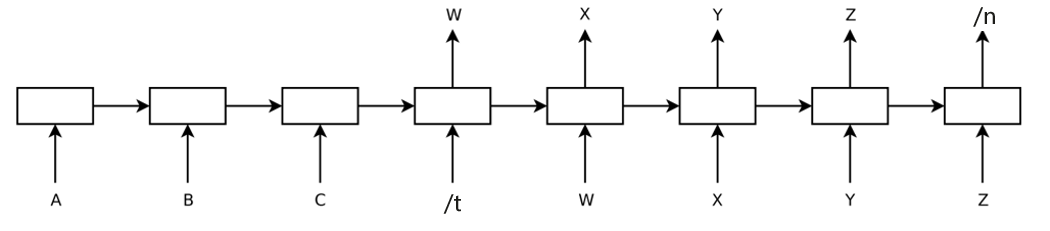
有相应Python语言的编程基础。

有神经网络LSTM的理论基础。

## 实验环境介绍

ModelArts平台：Mindspore-python3.7-aarch64

## 实验总体设计



实验总体设计示意图

如图所示，encoder接受输入ABC，编码后将语义信息传递给decoder，decoder经过解码后获得wxyz，/t为起始符，/n为结束符。

## 实验操作

参照实验步骤2.2.1，在ModelArts创建实验环境

在Notebook中下载数据集和代码

!wget https://ascend-professional-construction-dataset.obs.cn-north-4.myhuaweicloud.com/deep-learning/translation.zip

解压数据和代码文件

!unzip translation.zip

## 代码详解

本节将详细介绍实验的设计与实现。

### config.py文件

from easydict import EasyDict as edict # 以属性的方式去访问字典的值

# 实验的参数设定表

cfg = edict({

'en\_vocab\_size': 1154, # 英文字典的大小，也就是英文的 subword 的个数

'ch\_vocab\_size': 1116, # 中文字典的大小

'max\_seq\_length': 10, # 字数的个数

'hidden\_size': 1024, # 隐藏单元数

'batch\_size': 16, # 批尺寸大小

'eval\_batch\_size': 1,

'learning\_rate': 0.001, # 学习率

'momentum': 0.9, # 动量优化器参数

'num\_epochs': 15,# 训练全部数据集迭代次数

'save\_checkpoint\_steps': 125, # 每隔这么多步骤保存检查点

'keep\_checkpoint\_max': 10, # 要保留的最近检查点文件的最大数量.当新文件被创建时,旧文件被删除.如果为None或0,则保留所有检查点文件.默认为5(也就是保留5个最近的检查点文件.)

'dataset\_path':'./preprocess', # 预处理路径

'ckpt\_save\_path':'./ckpt', # 储存模型的位置

'checkpoint\_path':'./ckpt/gru-15\_125.ckpt' # 储存检查点的位置

})

### dataset.py文件

import os # 导入标准库OS

import re

import sys

import numpy as np

from mindspore import dataset as ds

# 得到目标操作（通过encoder-decoder得到相应的输入输出）

def target\_operation(encoder\_data, decoder\_data):

encoder\_data = encoder\_data[1:]

target\_data = decoder\_data[1:]

decoder\_data = decoder\_data[:-1]

return encoder\_data, decoder\_data, target\_data

# 验证操作

def eval\_operation(encoder\_data, decoder\_data):

encoder\_data = encoder\_data[1:]

decoder\_data = decoder\_data[:-1]

return encoder\_data, decoder\_data

# 得到训练数据集

def create\_dataset(data\_home, batch\_size, repeat\_num=1, is\_training=True, device\_num=1, rank=0):

if is\_training:

data\_dir = os.path.join(data\_home, "gru\_train.mindrecord") # 合并路径

else:

data\_dir = os.path.join(data\_home, "gru\_eval.mindrecord") #

data\_set = ds.MindDataset(data\_dir, columns\_list=["encoder\_data","decoder\_data"], num\_parallel\_workers=4,

num\_shards=device\_num, shard\_id=rank) # 通过训练分别得到encoder和decoder的数据集

if is\_training: # 训练阶段

operations = target\_operation # 调用得到目标数据

data\_set = data\_set.map(operations=operations, input\_columns=["encoder\_data","decoder\_data"],

output\_columns=["encoder\_data","decoder\_data","target\_data"],

column\_order=["encoder\_data","decoder\_data","target\_data"])

else: # 验证阶段

operations = eval\_operation

data\_set = data\_set.map(operations=operations, input\_columns=["encoder\_data","decoder\_data"],

output\_columns=["encoder\_data","decoder\_data"],

column\_order=["encoder\_data","decoder\_data"])

data\_set = data\_set.shuffle(buffer\_size=data\_set.get\_dataset\_size()) # 打乱数据集

data\_set = data\_set.batch(batch\_size=batch\_size, drop\_remainder=True) # 将数据集分批

data\_set = data\_set.repeat(count=repeat\_num) # 重复数据集

return data\_set

### preprocess.py文件

import os

import re

import sys

import random

import numpy as np

import unicodedata # 使用unicodedata模块先将文本标准化

from mindspore import dataset as ds

from mindspore.mindrecord import FileWriter

# 预备特殊字元，在开头添加 <SOS>，在结尾添加 <EOS>

EOS = "<eos>"

SOS = "<sos>"

MAX\_SEQ\_LEN=10

# 多用于那些需要包含音调的字符体系中，Unicode体系中，使用Decompose(分离)分别存储字符(U+0043)本身和音调(U+0327)本身。

# 从给定的字符串中删除重音符号。 输入文本是unicode字符串，返回带有重音符号的输入字符串，作为unicode。

# normalize() 第一个参数指定字符串标准化的方式。 NFD表示字符应该分解为多个组合字符表示。

def unicodeToAscii(s):

return ''.join(

c for c in unicodedata.normalize('NFD', s)

if unicodedata.category(c) != 'Mn'

)

# 标准化处理字符串

def normalizeString(s):

s = s.lower().strip() # lower将整个字符串改为小写；strip删除字符串前后的空白。

s = unicodeToAscii(s) # 调用函数将Unicode转化成Ascii

s = re.sub(r"([.!?])", r" \1", s)

s = re.sub(r"[^a-zA-Z.!?]+", r" ", s) # 将符号“.!?”前用空格隔开

return s

def prepare\_data(data\_path, vocab\_save\_path, max\_seq\_len):

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

data = f.read() # 读取文件

# 得到文件中的内容

data = data.split('\n')

data = data[:2000]

# 拆分英文句子和中文句子

en\_data = [normalizeString(line.split('\t')[0]) for line in data] # 得到标准化处理的英文句子

ch\_data = [line.split('\t')[1] for line in data] # 得到中文句子

# 获取单词并存储

en\_vocab = set(' '.join(en\_data).split(' ')) # 获取不重复的英文单词

id2en = [EOS] + [SOS] + list(en\_vocab) # 英文单词表中加上两个始末特殊字元

en2id = {c:i for i,c in enumerate(id2en)} # 遍历所有英文单词组合为一个索引序列

en\_vocab\_size = len(id2en) # 查看英文单词个数

np.savetxt(os.path.join(vocab\_save\_path, 'en\_vocab.txt'), np.array(id2en), fmt='%s') # 将英文单词表保存

ch\_vocab = set(''.join(ch\_data)) # 获取不重复的中文单词

id2ch = [EOS] + [SOS] + list(ch\_vocab) # 中文单词表中加上两个始末特殊字元

ch2id = {c:i for i,c in enumerate(id2ch)} # 遍历所有中文单词组合为一个索引序列，即获取每个单词的id

ch\_vocab\_size = len(id2ch) # 查看中文单词个数

np.savetxt(os.path.join(vocab\_save\_path, 'ch\_vocab.txt'), np.array(id2ch), fmt='%s') # 将中文单词表保存

# 将中英文句子转换为单词ids组合 --> [SOS] + sentences ids + [EOS]

en\_num\_data = np.array([[1] + [int(en2id[en]) for en in line.split(' ')] + [0] for line in en\_data])

ch\_num\_data = np.array([[1] + [int(ch2id[ch]) for ch in line] + [0] for line in ch\_data])

# 将上述句子的索引ID组合长度延长到自定义的max\_length

for i in range(len(en\_num\_data)):

num = max\_seq\_len + 1 - len(en\_num\_data[i])

if(num >= 0):

en\_num\_data[i] += [0]\*num

else:

en\_num\_data[i] = en\_num\_data[i][:max\_seq\_len] + [0]

for i in range(len(ch\_num\_data)):

num = max\_seq\_len + 1 - len(ch\_num\_data[i])

if(num >= 0):

ch\_num\_data[i] += [0]\*num

else:

ch\_num\_data[i] = ch\_num\_data[i][:max\_seq\_len] + [0]

return en\_num\_data, ch\_num\_data, en\_vocab\_size, ch\_vocab\_size

# 转换保存mindspore的中英文单词表

def convert\_to\_mindrecord(data\_path, mindrecord\_save\_path, max\_seq\_len):

en\_num\_data, ch\_num\_data, en\_vocab\_size, ch\_vocab\_size = prepare\_data(data\_path, mindrecord\_save\_path, max\_seq\_len)

data\_list\_train = []

for en, de in zip(en\_num\_data, ch\_num\_data):

en = np.array(en).astype(np.int32) # 将英文句子ID强制转换为指定的整数类型。

de = np.array(de).astype(np.int32) # 将中文句子ID强制转换为指定的整数类型。

data\_json = {"encoder\_data": en.reshape(-1),

"decoder\_data": de.reshape(-1)}

data\_list\_train.append(data\_json) # 将英文作为编码器，中文作为解码器加入

data\_list\_eval = random.sample(data\_list\_train, 20)

data\_dir = os.path.join(mindrecord\_save\_path, "gru\_train.mindrecord") # 把目录和文件名合成一个路径.

writer = FileWriter(data\_dir) # 用于将用户定义的原始数据写入MindRecord File系列。

schema\_json = {"encoder\_data": {"type": "int32", "shape": [-1]},

"decoder\_data": {"type": "int32", "shape": [-1]}} # 设计编码器和解码器架构

writer.add\_schema(schema\_json, "gru\_schema") # 添加架构，如果成功添加架构，则返回架构ID，或引发异常。

writer.write\_raw\_data(data\_list\_train) # 默认情况下，写入原始数据，生成MindRecord File的顺序对，并根据预定义的模式对数据进行校验。

writer.commit() # 将数据刷新到磁盘并生成相应的db文件。

data\_dir = os.path.join(mindrecord\_save\_path, "gru\_eval.mindrecord")

writer = FileWriter(data\_dir)

writer.add\_schema(schema\_json, "gru\_schema")

writer.write\_raw\_data(data\_list\_eval)

writer.commit()

print("en\_vocab\_size: ", en\_vocab\_size) # 打印出英文单词长度

print("ch\_vocab\_size: ", ch\_vocab\_size) # 打印出中文单词长度

return en\_vocab\_size, ch\_vocab\_size

if \_\_name\_\_=='\_\_main\_\_':

convert\_to\_mindrecord("src/cmn\_zhsim.txt", './preprocess', MAX\_SEQ\_LEN)

### seq2seq.py文件

"""Seq2Seq构建"""

import math

import numpy as np

from mindspore import Tensor

from mindspore import Tensor, Parameter

import mindspore.nn as nn

import mindspore.ops.operations as P

import mindspore.common.dtype as mstype

from src.loss import NLLLoss

# 定义GRU中的权重和偏置

def gru\_default\_state(batch\_size, input\_size, hidden\_size, num\_layers=1, bidirectional=False):

'''GRU（LSTM的变体）的权重初始化'''

stdv = 1 / math.sqrt(hidden\_size) # 设置标准差

# 输入层权重初始化（权重从一个均匀分布[low,high)中随机采样）

weight\_i = Parameter(Tensor(

np.random.uniform(-stdv, stdv, (input\_size, 3\*hidden\_size)).astype(np.float32)), name='weight\_i')

# 隐藏层权重初始化

weight\_h = Parameter(Tensor(

np.random.uniform(-stdv, stdv, (hidden\_size, 3\*hidden\_size)).astype(np.float32)), name='weight\_h')

# 输入层偏置初始化

bias\_i = Parameter(Tensor(

np.random.uniform(-stdv, stdv, (3\*hidden\_size)).astype(np.float32)), name='bias\_i')

# 隐藏层偏置初始化

bias\_h = Parameter(Tensor(

np.random.uniform(-stdv, stdv, (3\*hidden\_size)).astype(np.float32)), name='bias\_h')

return weight\_i, weight\_h, bias\_i, bias\_h

# 定义GRU网络

class GRU(nn.Cell):

def \_\_init\_\_(self, config, is\_training=True):

super(GRU, self).\_\_init\_\_()

if is\_training: # 确认是否训练，若是训练，则采用训练集

self.batch\_size = config.batch\_size

else:

self.batch\_size = config.eval\_batch\_size

self.hidden\_size = config.hidden\_size # 调用参数config

self.weight\_i, self.weight\_h, self.bias\_i, self.bias\_h = \

gru\_default\_state(self.batch\_size, self.hidden\_size, self.hidden\_size) # 调用GRU网络中的权重和偏置

self.rnn = P.DynamicGRUV2() # 调用AI框架Mindspore的GRU网络

self.cast = P.Cast() # 转换成特定的数据类型

def construct(self, x, hidden):

x = self.cast(x, mstype.float16) # 转换成特定的数据类型

y1, h1, \_, \_, \_, \_ = self.rnn(x, self.weight\_i, self.weight\_h, self.bias\_i, self.bias\_h, None, hidden)

return y1, h1

# 定义编码器，将一组序列编码成一个向量，选用GRU在最后一个时间点的输出hidden来作为來context vector。

class Encoder(nn.Cell):

def \_\_init\_\_(self, config, is\_training=True):

super(Encoder, self).\_\_init\_\_()

self.vocab\_size = config.en\_vocab\_size # 英文词典大小

self.hidden\_size = config.hidden\_size # 隐藏层单元数量

if is\_training: # 确认训练过程

self.batch\_size = config.batch\_size

else:

self.batch\_size = config.eval\_batch\_size

self.trans = P.Transpose() # 矩阵转置

self.perm = (1, 0, 2) # 维度的重新排列

self.embedding = nn.Embedding(self.vocab\_size, self.hidden\_size) # 设置嵌入层于特定维度

self.gru = GRU(config, is\_training=is\_training).to\_float(mstype.float16) # 转换成特定类型

self.h = Tensor(np.zeros((self.batch\_size, self.hidden\_size)).astype(np.float16)) # 隐藏层张量初始输入设为0

def construct(self, encoder\_input):

embeddings = self.embedding(encoder\_input) # 输入嵌入层

embeddings = self.trans(embeddings, self.perm) # 设置输入层：转置+维度排列

output, hidden = self.gru(embeddings, self.h) # 经过隐藏层输出

return output, hidden

# 定义解码器，额外加上一个线性输出层out，用来预测当时时间点的输出字母：

class Decoder(nn.Cell):

def \_\_init\_\_(self, config, is\_training=True):

super(Decoder, self).\_\_init\_\_()

self.vocab\_size = config.ch\_vocab\_size # 中文词表大小

self.hidden\_size = config.hidden\_size # 隐藏层单元数量

self.trans = P.Transpose() # 矩阵转置

self.perm = (1, 0, 2) # 维度的重新排列

self.embedding = nn.Embedding(self.vocab\_size, self.hidden\_size) # 设置嵌入层于特定维度

self.gru = GRU(config, is\_training=is\_training).to\_float(mstype.float16) # 定义GRU网络，确认数据类型

self.dense = nn.Dense(self.hidden\_size, self.vocab\_size) # 定义全连接层

self.softmax = nn.LogSoftmax(axis=2) # 定义LogSoftmax激活函数，数值稳定性优于Softmax

self.cast = P.Cast() # 转换数据类型

def construct(self, decoder\_input, hidden):

embeddings = self.embedding(decoder\_input) # 解码器嵌入层输入

embeddings = self.trans(embeddings, self.perm) # 输入嵌入层转置+维度排列

output, hidden = self.gru(embeddings, hidden) # 通过隐藏层输出

output = self.cast(output, mstype.float32) # 将输出转换数据格式

output = self.dense(output) # 通过全连接层输出

output = self.softmax(output) # 通过激活函数得结果

return output, hidden

# 构建Seq2Seq模型

class Seq2Seq(nn.Cell):

def \_\_init\_\_(self, config, is\_train=True):

super(Seq2Seq, self).\_\_init\_\_()

self.max\_len = config.max\_seq\_length # Token序列的最大长度

self.is\_train = is\_train # 确认是否训练

self.encoder = Encoder(config, is\_train) # 确认训练编码器

self.decoder = Decoder(config, is\_train) # 确认训练解码器

self.expanddims = P.ExpandDims() # 扩展维度

self.squeeze = P.Squeeze(axis=0) # 移除维度

self.argmax = P.ArgMaxWithValue(axis=int(2), keep\_dims=True) # 输出最大索引值

self.concat = P.Concat(axis=1) # 横向合并

self.concat2 = P.Concat(axis=0) # 纵向合并

self.select = P.Select()

def construct(self, src, dst):

encoder\_output, hidden = self.encoder(src) # 将编码器输入到隐藏层训练

decoder\_hidden = self.squeeze(encoder\_output[self.max\_len-2:self.max\_len-1:1, ::, ::]) # 将编码器的输出到解码器的隐藏层

if self.is\_train:

outputs, \_ = self.decoder(dst, decoder\_hidden)

else:

decoder\_input = dst[::,0:1:1]

decoder\_outputs = ()

for i in range(0, self.max\_len):

decoder\_output, decoder\_hidden = self.decoder(decoder\_input, decoder\_hidden)

# 从[seq\_length，batch\_size, hidden\_size]squeeze把第一维移除变成[batch\_size, hidden\_size]

decoder\_hidden = self.squeeze(decoder\_hidden)

decoder\_output, \_ = self.argmax(decoder\_output) # 获取最大结果

decoder\_output = self.squeeze(decoder\_output) # 移除第一维度（seq\_length）

decoder\_outputs += (decoder\_output,) # 更新结果

decoder\_input = decoder\_output

outputs = self.concat(decoder\_outputs) # 横向合并解码器结果

return outputs

class WithLossCell(nn.Cell):

def \_\_init\_\_(self, backbone, config):

super(WithLossCell, self).\_\_init\_\_(auto\_prefix=False)

self.\_backbone = backbone

self.batch\_size = config.batch\_size

self.onehot = nn.OneHot(depth=config.ch\_vocab\_size) # 独热编码中文

self.\_loss\_fn = NLLLoss() # 调用损失函数

self.max\_len = config.max\_seq\_length # 最长序列参数

self.squeeze = P.Squeeze() # 移除维度

self.cast = P.Cast() # 转换数据类型

self.argmax = P.ArgMaxWithValue(axis=1, keep\_dims=True)

self.print = P.Print()

def construct(self, src, dst, label):

out = self.\_backbone(src, dst)

loss\_total = 0 # 定义初始损失值

for i in range(self.batch\_size):

loss = self.\_loss\_fn(self.squeeze(out[::,i:i+1:1,::]), self.squeeze(label[i:i+1:1, ::]))

loss\_total += loss

loss = loss\_total / self.batch\_size # 单个批尺寸数据集的损失值

return loss

class InferCell(nn.Cell):

def \_\_init\_\_(self, network, config):

super(InferCell, self).\_\_init\_\_(auto\_prefix=False)

self.expanddims = P.ExpandDims() # 扩展维度

self.network = network

def construct(self, src, dst):

out = self.network(src, dst)

return out

### loss.py文件

"""NLLLoss cell"""

import mindspore.ops.operations as P

from mindspore.nn.loss.loss import \_Loss

from mindspore.ops import functional as F

# 定义损失函数

class NLLLoss(\_Loss):

'''

NLLLoss function输入是一个对数概率向量和一个目标标签。NLLLoss() ，即负对数似然损失函数（Negative Log Likelihood）

'''

def \_\_init\_\_(self, reduction='mean'):

super(NLLLoss, self).\_\_init\_\_(reduction)

self.one\_hot = P.OneHot() # 调用MindSpore中独热编码模块

self.reduce\_sum = P.ReduceSum() # # 调用MindSpore中求和模块，计算张量tensor沿着某一维度的和，可以在求和后降维。

def construct(self, logits, label):

label\_one\_hot = self.one\_hot(label, F.shape(logits)[-1], F.scalar\_to\_array(1.0), F.scalar\_to\_array(0.0)) # 将标签进行独热编码

# print('NLLLoss label\_one\_hot:',label\_one\_hot, label\_one\_hot.shape)

# print('NLLLoss logits:',logits, logits.shape)

# print('xxx:', logits \* label\_one\_hot)

loss = self.reduce\_sum(-1.0 \* logits \* label\_one\_hot, (1,)) # 为计算损失值，最小化损失函数值，函数取负号，若实际标签张量在模型输出结果的对应位置的值越接近0，则具有越小的损失值

return self.get\_loss(loss)

### train.py文件

import argparse

import os

import numpy as np

from src.dataset import create\_dataset

from src.seq2seq import Seq2Seq, WithLossCell

from src.config import cfg

from mindspore import Tensor, nn, Model, context

from mindspore.train.callback import LossMonitor, CheckpointConfig, ModelCheckpoint, TimeMonitor

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser(description='MindSpore LSTM Example')

parser.add\_argument('--dataset\_path', type=str, default='./preprocess', help='dataset path.')

parser.add\_argument('--ckpt\_save\_path', type=str, default='./', help='checkpoint save path.')

args = parser.parse\_args()

# 在Ascend芯片设备中训练

context.set\_context(

mode=context.GRAPH\_MODE,

save\_graphs=False,

device\_target='Ascend')

ds\_train = create\_dataset(args.dataset\_path, cfg.batch\_size) # 获取数据集，分批训练

network = Seq2Seq(cfg) # 根据预设参数构建模型

network = WithLossCell(network, cfg) # 记录单个批尺寸数据集的损失值

optimizer = nn.Adam(network.trainable\_params(), learning\_rate=cfg.learning\_rate, beta1=0.9, beta2=0.98) # 使用Adam优化器

model = Model(network, optimizer=optimizer) # 构建模型

loss\_cb = LossMonitor() # 监测损失值

# 保存检查点

config\_ck = CheckpointConfig(save\_checkpoint\_steps=cfg.save\_checkpoint\_steps, keep\_checkpoint\_max=cfg.keep\_checkpoint\_max)

ckpoint\_cb = ModelCheckpoint(prefix="gru", directory=args.ckpt\_save\_path, config=config\_ck)

time\_cb = TimeMonitor(data\_size=ds\_train.get\_dataset\_size()) # 监测时间

callbacks = [time\_cb, ckpoint\_cb, loss\_cb] # 使用回调函数

model.train(cfg.num\_epochs, ds\_train, callbacks=callbacks, dataset\_sink\_mode=False) # 训练模型

### eval.py文件

import argparse

import os

import numpy as np

from src.dataset import create\_dataset

from src.seq2seq import Seq2Seq, InferCell

from src.config import cfg

from mindspore import Tensor, nn, Model, context, DatasetHelper

from mindspore.train.serialization import load\_param\_into\_net, load\_checkpoint

from mindspore.communication.management import init, get\_rank

from mindspore.context import ParallelMode

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser(description='MindSpore GRU Example')

parser.add\_argument('--dataset\_path', type=str, default='./preprocess', help='dataset path.')

parser.add\_argument('--checkpoint\_path', type=str, default='', help='checkpoint path.')

args = parser.parse\_args()

# 在Ascend芯片设备中训练

context.set\_context(

mode=context.GRAPH\_MODE,# MindSpore图模式

save\_graphs=False,

device\_target='Ascend')

rank = 0

device\_num = 1 # 设备数

ds\_eval= create\_dataset(args.dataset\_path, cfg.eval\_batch\_size, is\_training=False) # 获取数据集

network = Seq2Seq(cfg,is\_train=False) # 建立Seq2Seq网络

network = InferCell(network, cfg) # 将设定的参数带入网络

network.set\_train(False) # 验证阶段

parameter\_dict = load\_checkpoint(args.checkpoint\_path) # 加载检查点

load\_param\_into\_net(network, parameter\_dict)

model = Model(network) # 建立模型

with open(os.path.join(args.dataset\_path,"en\_vocab.txt"), 'r', encoding='utf-8') as f:

data = f.read() # 读取英文词表

en\_vocab = list(data.split('\n')) # 换行分割

with open(os.path.join(args.dataset\_path,"ch\_vocab.txt"), 'r', encoding='utf-8') as f:

data = f.read() # 读取中文词表

ch\_vocab = list(data.split('\n'))

# 创建中英文对照输出

for data in ds\_eval.create\_dict\_iterator():

en\_data=''

ch\_data=''

for x in data['encoder\_data'][0]: # 编码器输出

if x == 0:

break

en\_data += en\_vocab[x] # 将英文数据逐步更新

en\_data += ' ' # 空格间隔

for x in data['decoder\_data'][0]: # 解码器输出

if x == 0:

break

if x == 1:

continue

ch\_data += ch\_vocab[x] # 将中文数据逐步更新

output = network(data['encoder\_data'],data['decoder\_data']) # 输出结果

print('English:', en\_data) # 打印英文结果

print('expect Chinese:', ch\_data) # 打印对应的中文翻译

out ='' # 中文结果初始空白

for x in output[0]:

if x == 0:

break

out += ch\_vocab[x]

print('predict Chinese:', out) # 答应翻译中文结果

print(' ')

### main.ipynb文件

查看当前文件夹内容：

!ls

输出：

eval.py main.ipynb src train.py

导入项目环境：

import os

import numpy as np

from mindspore import Tensor, nn, Model, context

from mindspore.train.callback import LossMonitor, CheckpointConfig, ModelCheckpoint, TimeMonitor

from mindspore.communication.management import init, get\_rank

from mindspore.context import ParallelMode

from mindspore.train.serialization import load\_param\_into\_net, load\_checkpoint

from src.preprocess import convert\_to\_mindrecord

from src.dataset import create\_dataset

from src.seq2seq import Seq2Seq, WithLossCell, InferCell

from src.config import cfg

context.set\_context(mode=context.GRAPH\_MODE, save\_graphs=False, device\_target='Ascend') # 选用Ascend芯片执行运算

创建preprocess文件夹：

! mkdir preprocess # Linux命令用于创建目录。

!ls

输出：

eval.py main.ipynb preprocess src train.py

执行preprocess.py文件：

! python src/preprocess.py # 运行文件

输出：

en\_vocab\_size: 1154

ch\_vocab\_size: 1116

模型训练：

ds\_train = create\_dataset(cfg.dataset\_path, cfg.batch\_size) # 获取数据集

network = Seq2Seq(cfg) # 根据设定参数构建Seq2Seq

network = WithLossCell(network, cfg) # 记录损失值

optimizer = nn.Adam(network.trainable\_params(), learning\_rate=cfg.learning\_rate, beta1=0.9, beta2=0.98) # Adam优化器

model = Model(network, optimizer=optimizer) # 加Adam优化器构建网路

loss\_cb = LossMonitor() # 检测显示损失值

config\_ck = CheckpointConfig(save\_checkpoint\_steps=cfg.save\_checkpoint\_steps, keep\_checkpoint\_max=cfg.keep\_checkpoint\_max) # 保存检查点

ckpoint\_cb = ModelCheckpoint(prefix="gru", directory=cfg.ckpt\_save\_path, config=config\_ck) # 保存模型

time\_cb = TimeMonitor(data\_size=ds\_train.get\_dataset\_size()) # 检测时间

callbacks = [time\_cb, ckpoint\_cb, loss\_cb] # 设置回调函数

model.train(cfg.num\_epochs, ds\_train, callbacks=callbacks, dataset\_sink\_mode=True) # 训练模型

输出：

epoch: 1 step: 125, loss is 2.5660143

epoch time: 78101.794 ms, per step time: 624.814 ms

epoch: 2 step: 125, loss is 2.62084

epoch time: 11434.319 ms, per step time: 91.475 ms

……

epoch: 13 step: 125, loss is 0.09685211

epoch time: 11445.864 ms, per step time: 91.567 ms

epoch: 14 step: 125, loss is 0.12204757

epoch time: 11428.892 ms, per step time: 91.431 ms

epoch: 15 step: 125, loss is 0.078245044

epoch time: 11440.658 ms, per step time: 91.525 ms

模型验证：

rank = 0

device\_num = 1 # 设备数

ds\_eval= create\_dataset(cfg.dataset\_path, cfg.eval\_batch\_size, is\_training=False) # 验证阶段

network = Seq2Seq(cfg,is\_train=False) # 构建Seq2Seq

network = InferCell(network, cfg) # 根据设定参数

network.set\_train(False) # 验证阶段

parameter\_dict = load\_checkpoint(cfg.checkpoint\_path) # 保存检查点

load\_param\_into\_net(network, parameter\_dict) # 加载参数

model = Model(network) # 构建模型

with open(os.path.join(cfg.dataset\_path,"en\_vocab.txt"), 'r', encoding='utf-8') as f:

data = f.read() # 读取英文词表

en\_vocab = list(data.split('\n'))

with open(os.path.join(cfg.dataset\_path,"ch\_vocab.txt"), 'r', encoding='utf-8') as f:

data = f.read() # 读取中文词表

ch\_vocab = list(data.split('\n'))

for data in ds\_eval.create\_dict\_iterator():

en\_data='' # 初始英文词句

ch\_data='' # 初始中文词句

for x in data['encoder\_data'][0].asnumpy():

if x == 0:

break

en\_data += en\_vocab[x] # 从此表中更新英文词句

en\_data += ' '

for x in data['decoder\_data'][0].asnumpy():

if x == 0:

break # 如果没有词句就中断

if x == 1:

continue

ch\_data += ch\_vocab[x] # # 从此表中更新中文词句

output = network(data['encoder\_data'],data['decoder\_data']) # 输出encoder和decoder的内容

print('English:', en\_data) # 输出英文词句

print('expect Chinese:', ch\_data) # 输出对应中文词句

out ='' # 初始结果

for x in output[0].asnumpy():

if x == 0:

break

out += ch\_vocab[x]

print('predict Chinese:', out) # 输出预测中文词句

print(' ')

输出：

English: he came in person .

expect Chinese: 他亲自来了。

predict Chinese: 他亲自来了。

English: i trust you .

expect Chinese: 我信任你。

predict Chinese: 我信任你。

English: facebook is boring .

expect Chinese: Facebook很

predict Chinese: Facebook很

English: where is my book ?

expect Chinese: 我的书在哪？

predict Chinese: 我的书在哪？

English: do you live here ?

expect Chinese: 你住这里吗?

predict Chinese: 你住这里吗?



## 实验总结

本章主要基于MindSpore构建神经网络模型用于中英文翻译的处理，导入数据，定义模型，设置好相关参数，测试样本数据，通过本实验希望学员能够进行自然语言处理中机器翻译的处理以及具备深度学习模型构建的基础编程能力

## 思考题-汇总

结合Seq2Seq框架及内部使用单元思考，Seq2Seq框架在编码-解码过程中是否存在信息丢失？具体表现在哪些方面？

## 开放题（可选）

2014年，Bahdanau等人提出注意力机制，其核心思想是在编码阶段提高了对序列前面信息的关注，请详细阅读论文并了解算法实现过程，尤其是Attention环节的实现，写一篇500字的总结。

论文链接：

DZMITRY B, CHO K, YOSHUA B. Neural machine translation by joinly learning to align and translate.[J]. arXiv preprint arXiv:1409.0473, 2014

# 附录：ModelArts开发环境搭建

在华为云ModelArts平台上创建AI框架为Mindspore-1.7，硬件环境为Ascend 910+ARM的开发环境。

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。

图形用户界面, 文本, 应用程序

描述已自动生成

华为云ModelArts主页

创建Notebook训练作业

控制台区域选择“华北-北京四”，在左侧菜单栏中选择“开发环境”的“Notebook”，点击进入Notebook创建页面。



ModelArts控制台

点击“创建”按钮，创建一个新的Notebook，其配置如下：

名称：自定义。

自动停止：建议1小时。

镜像：Ascend+ARM算法开发和训练基础镜像。

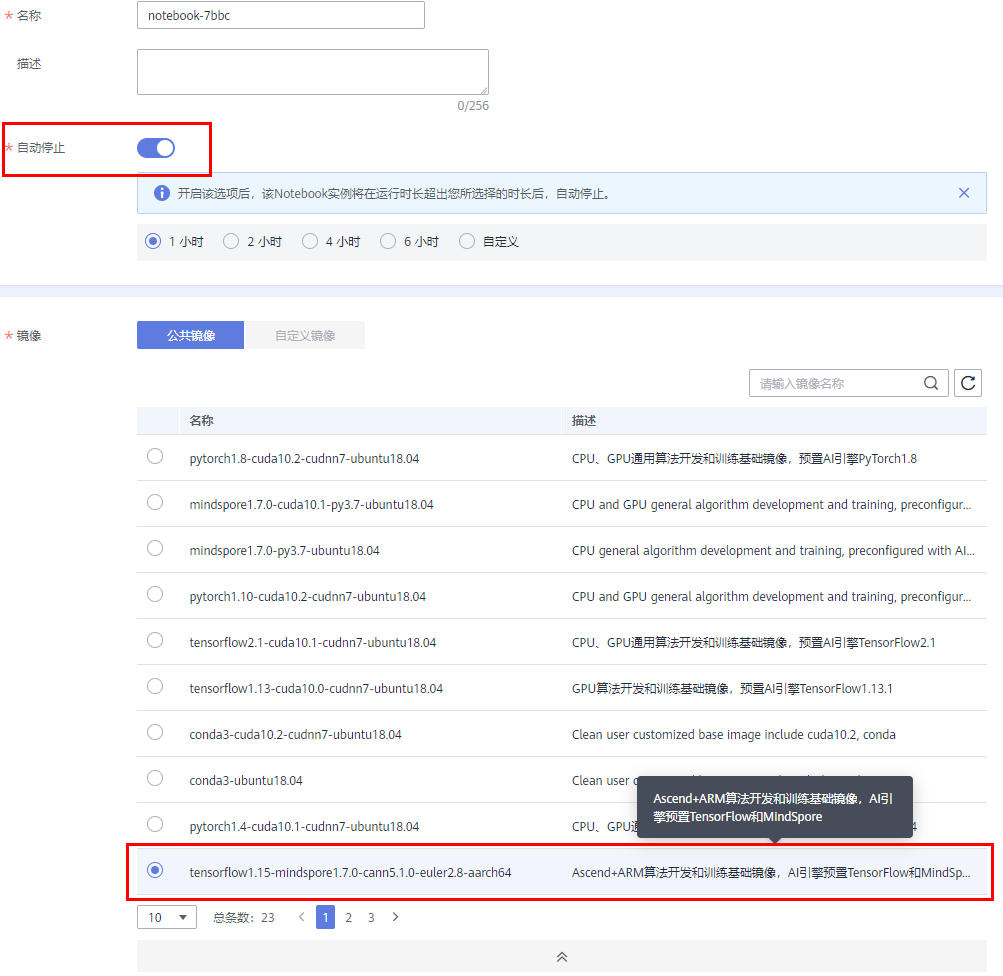
资源池：公共资源池。

类型：ASCEND。

规格：Ascend: 1\*Ascend910|CPU: 24核 96GB。

存储配置：默认存储（50GB），亦可选择EVS，支持自定义存储规格且为专属资源。

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”。

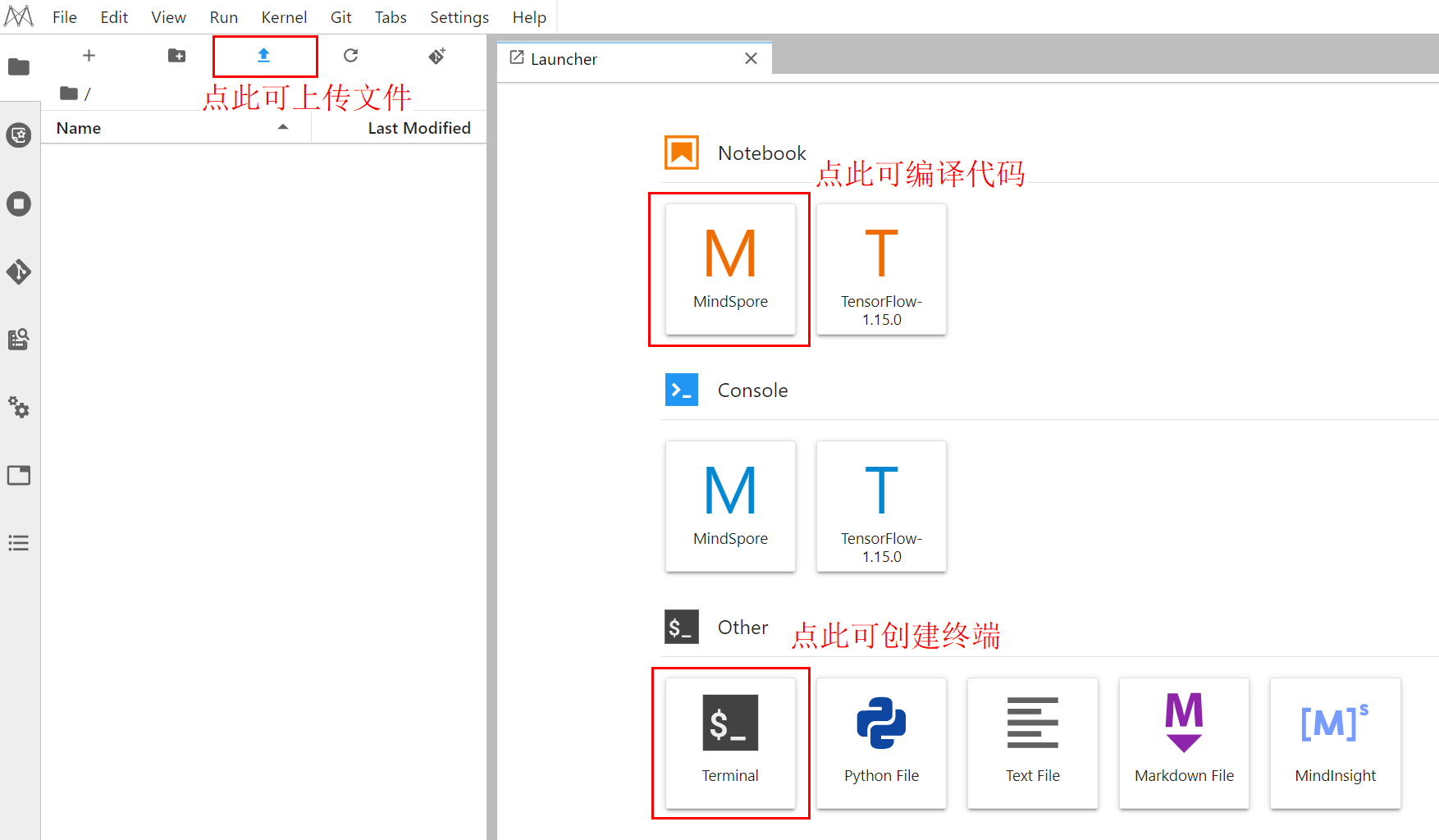
启动Notebook进入开发环境

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



打开Notebook环境

可以在此页面创建或编辑MindSpore的项目，如图所示：



Notebook开发页面

\*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。

停止Notebook训练作业

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

登录[华为云ModelArts控制台](https://console.huaweicloud.com/modelarts/?region=cn-north-4" \l "/dev-container)，在“操作”栏选择“停止”操作。

如下图所示：



及时停止Notebook

至此训练用的线上Notebook环境搭建完成。