**实验（实习）报告**

1. 实验目的

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。

1. 实验任务

1）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。

2）Seq2seq机器翻译：基于seq2seq模型进行中英文的翻译。

（见《深度学习》循环递归网络实验手册）

1. 实验步骤
2. 基于Mindspore的RNN-IMDB情感分析实验

1.1实验准备

步骤 1下载实验所需模块gensim

!pip install gensim



步骤 2导入实验所需模块

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

步骤 3删除文件夹

import shutil

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

    shutil.rmtree('./preprocess')

步骤 4传入必要信息

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')

步骤 5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

1.2 导入数据集并预处理

步骤 6数据集预处理

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

步骤 7数据集转换为MindRecord格式

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)

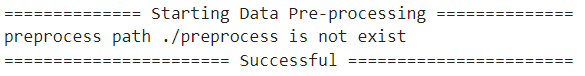
调用convert\_to\_mindrecord函数将数据集格式转换为MindRecord格式。

if args.preprocess == "true":

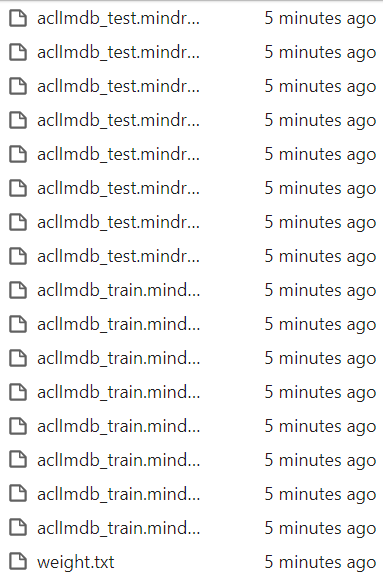
    print("============== Starting Data Pre-processing ==============")

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

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



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



步骤 8创建训练集

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)

步骤 9打印预处理后的数据

# 打印第一批的标签以及第一批第一个数据的特征

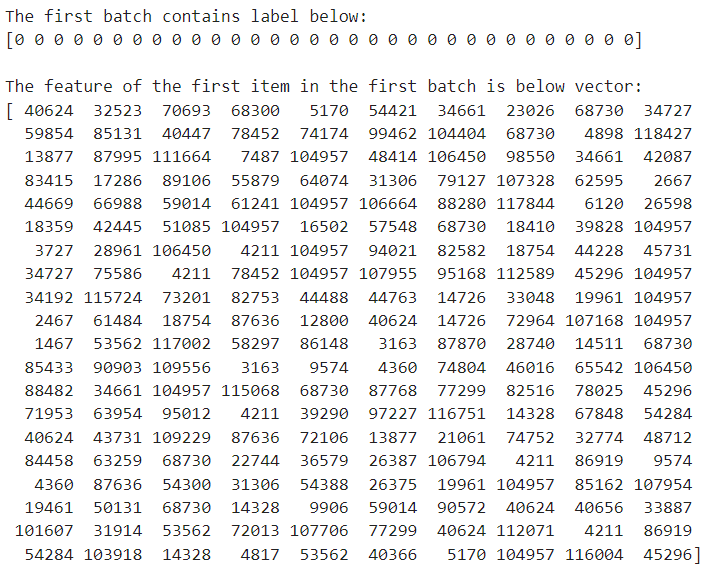
iterator = ds\_train.create\_dict\_iterator().\_get\_next()

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}")



1.3 LSTM模型建模

步骤 10初始化网络参数及网络状态

STACK\_LSTM\_DEVICE = ["CPU"]

# 短期内存（h）和长期内存（c）初始化为0

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

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)

步骤 11创建网络

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)

步骤 12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)

步骤 13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

步骤 14训练模型并保存

# 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)

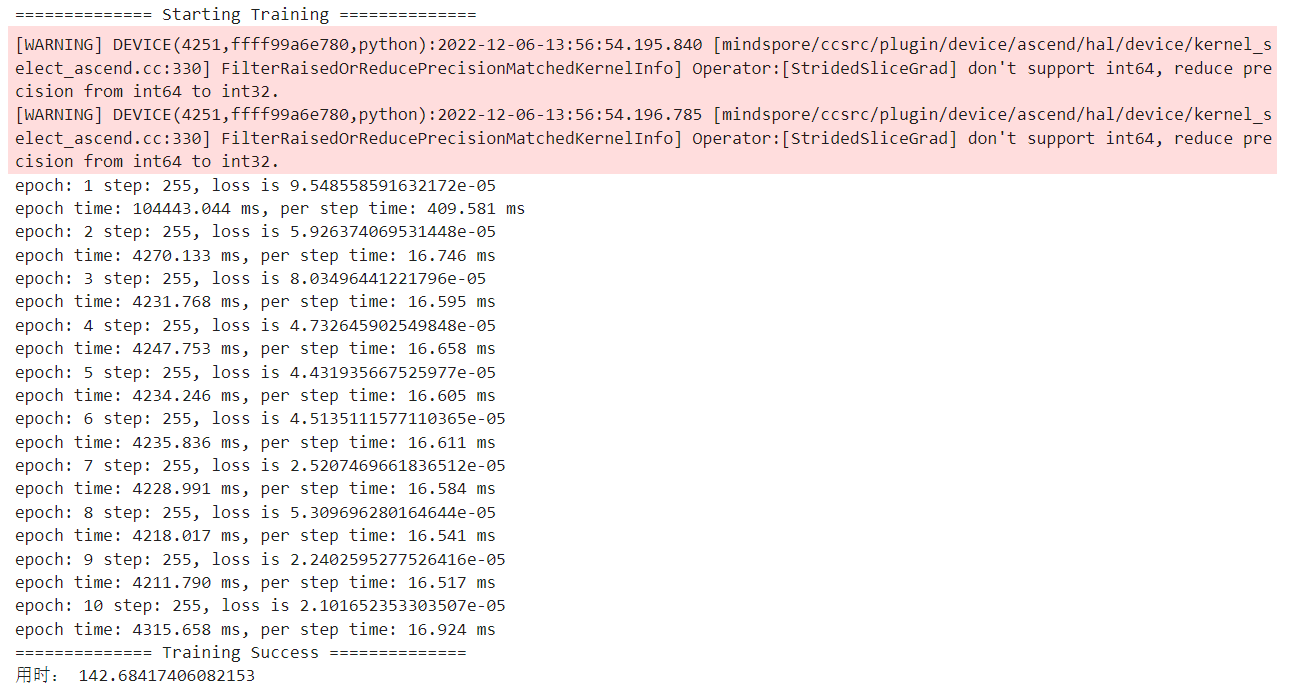
else:

    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)



1.4 模型评估

# 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))



1. Seq2seq机器翻译

2.1 代码详解

2.1.1 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

})

2.1.2 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

2.1.3 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)

2.1.5 loss.py文件

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)

2.1.6 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

2.1.7 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(' ')

2.2 main.ipynb

2.2.1查看当前文件夹内容：

!ls



2.2.2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

2.2.3创建preprocess文件夹：

! mkdir preprocess # Linux命令用于创建目录。

!ls



2.2.4执行preprocess.py文件：

! python src/preprocess.py # 运行文件

2.2.5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

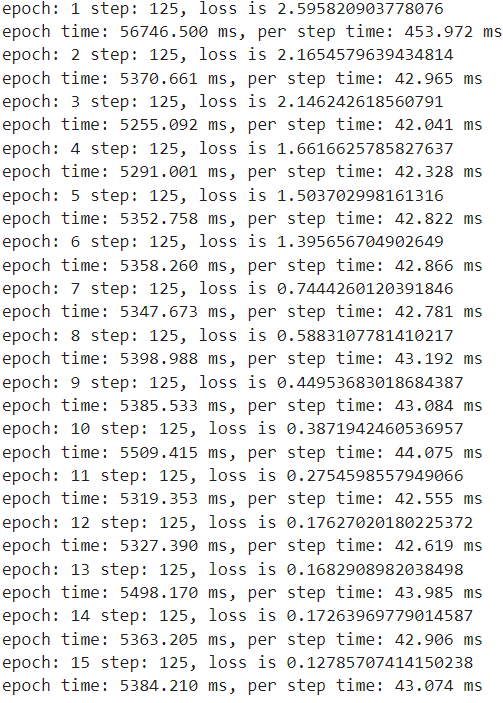
config\_ck = CheckpointConfig(save\_checkpoint\_steps=cfg.save\_checkpoint\_steps, keep\_checkpoint\_max=cfg.keep\_checkpoint\_max) # 保存检查点

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



2.2.6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(' ')

预测结果

English: speak clearly .

expect Chinese: 讲清楚。

predict Chinese: 讲清楚。

English: look out !

expect Chinese: 当心！

predict Chinese: 当心！

English: i don t like wine .

expect Chinese: 我不喜欢红酒。

predict Chinese: 我不喜欢喝茶。

English: let s not argue .

expect Chinese: 我们别吵了。

predict Chinese: 我们别吵了。

English: i haven t changed .

expect Chinese: 我没改变。

predict Chinese: 我没通过。

English: i m so happy .

expect Chinese: 我好高兴。

predict Chinese: 我好高兴。

English: please contact us .

expect Chinese: 请联系我们。

predict Chinese: 请联系我们。

English: he asked after you .

expect Chinese: 他问候了你。

predict Chinese: 他问候了你。

English: lemons are sour .

expect Chinese: 柠檬是酸的。

predict Chinese: 柠檬是酸的。

English: we need more food .

expect Chinese: 我们需要更多食物。

predict Chinese: 我们需要更多食物。

English: why not ?

expect Chinese: 为什么不？

predict Chinese: 为什么？

English: it s dangerous !

expect Chinese: 它是危险的!

predict Chinese: 它是危险的!

English: i saw five men .

expect Chinese: 我看到了五个男人。

predict Chinese: 我看到了五个男人。

English: the well ran dry .

expect Chinese: 这口井干涸了。

predict Chinese: 这口井干涸了。

English: it s very cold .

expect Chinese: 非常冷。

predict Chinese: 非常冷。

English: i said shut up !

expect Chinese: 我说过了，闭嘴！

predict Chinese: 我说过了，闭嘴！

English: i don t like eggs .

expect Chinese: 我不喜欢鸡蛋。

predict Chinese: 我不喜欢鸡蛋。

English: there s no rush .

expect Chinese: 不急。

predict Chinese: 不急。

English: i went on reading .

expect Chinese: 我继续阅读。

predict Chinese: 我继续阅读。

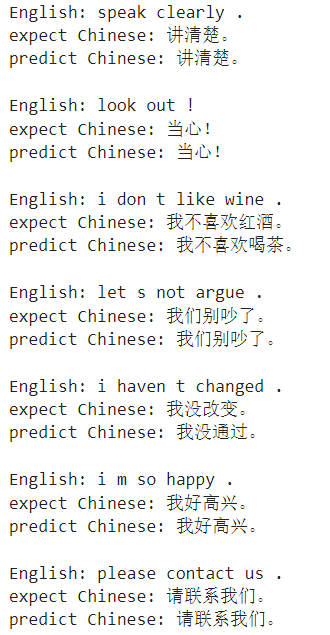
English: i love sports .

expect Chinese: 我喜欢运动。

predict Chinese: 我喜欢运动。

1. 实验结果





1. 实验总结

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。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。