自然语言处理的应用——基于mindspore的情感分类实验

概述

情感分类是自然语言处理中文本分类问题的子集，属于自然语言处理最基础的应用。它是对带有感情色彩的主观性文本进行分析和推理的过程，即分析说话人的态度，是倾向正面还是反面。

> 通常情况下，我们会把情感类别分为正面、反面和中性三类。虽然“面无表情”的评论也有不少；不过，大部分时候会只采用正面和反面的案例进行训练，下面这个数据集就是很好的例子。

传统的文本主题分类问题的典型参考数据集为20 Newsgroups，该数据集由20组新闻数据组成，包含约20000个新闻文档。 其主题列表中有些类别的数据比较相似，例如comp.sys.ibm.pc.hardware和comp.sys.mac.hardware都是和电脑系统硬件相关的题目，相似度比较高。而有些主题类别的数据相对来说就毫无关联，例如misc.forsale和soc.religion.christian。

就网络本身而言，文本主题分类的网络结构和情感分类的网络结构大致相似。在掌握了情感分类网络如何构造之后，很容易可以构造一个类似的网络，稍作调参即可用于文本主题分类任务。

但在业务上下文侧，文本主题分类是分析文本讨论的客观内容，而情感分类是要从文本中得到它是否支持某种观点的信息。比如，“《阿甘正传》真是好看极了，影片主题明确，节奏流畅。”这句话，在文本主题分类是要将其归为类别为“电影”主题，而情感分类则要挖掘出这一影评的态度是正面还是负面。

相对于传统的文本主题分类，情感分类较为简单，实用性也较强。常见的购物网站、电影网站都可以采集到相对高质量的数据集，也很容易给业务领域带来收益。例如，可以结合领域上下文，自动分析特定类型客户对当前产品的意见，可以分主题分用户类型对情感进行分析，以作针对性的处理，甚至基于此进一步推荐产品，提高转化率，带来更高的商业收益。

特殊领域中，某些非极性词也充分表达了用户的情感倾向，比如下载使用APP时，“卡死了”、“下载太慢了”就表达了用户的负面情感倾向；股票领域中，“看涨”、“牛市”表达的就是用户的正面情感倾向。所以，本质上，我们希望模型能够在垂直领域中，挖掘出一些特殊的表达，作为极性词给情感分类系统使用：

垂直极性词=通用极性词+领域特有极性词垂直极性词=通用极性词+领域特有极性词

按照处理文本的粒度不同，情感分析可分为词语级、短语级、句子级、段落级以及篇章级等几个研究层次。这里以“段落级”为例，输入为一个段落，输出为影评是正面还是负面的信息。

本次实验，以IMDB影评情感分类体验MindSpore在自然语言处理上的应用。

整体流程

1. 准备环节。
2. 加载数据集，进行数据处理。
3. 定义网络。
4. 定义优化器和损失函数。
5. 使用网络训练数据，生成模型。
6. 得到模型之后，使用验证数据集，查看模型精度情况。

准备环节

数据集

本次实验采用IMDB影评数据集作为实验数据。

1. 下载IMDB影评数据集。

   以下是负面影评（Negative）和正面影评（Positive）的案例。

Review	Label
"Quitting" may be as much about exiting a pre-ordained identity as about drug withdrawal. As a rural guy coming to Beijing, class and success must have struck this young artist face on as an appeal to separate from his roots and far surpass his peasant parents' acting success. Troubles arise, however, when the new man is too new, when it demands too big a departure from family, history, nature, and personal identity. The ensuing splits, and confusion between the imaginary and the real and the dissonance between the ordinary and the heroic are the stuff of a gut check on the one hand or a complete escape from self on the other.	Negative
This movie is amazing because the fact that the real people portray themselves and their real life experience and do such a good job it's like they're almost living the past over again. Jia Hongsheng plays himself an actor who quit everything except music and drugs struggling with depression and searching for the meaning of life while being angry at everyone especially the people who care for him most.	Positive

将下载好的数据集解压并放在当前工作目录下的`datasets`目录下，每解压1000个文件将在底部追加打印一个黑点。

下载GloVe文件 下载并解压GloVe文件到当前工作目录下的`datasets`目录下，并在所有Glove文件开头处添加如下所示新的一行，意思是总共读取400000个单词，每个单词用300纬度的词向量表示。

```
400000 300
```

将数据集解压到当前工作目录下，建立结构如下所示。

```
├── ckpt
├── datasets
│   ├── aclImdb
│   │   ├── imdbEr.txt
│   │   ├── imdb.vocab
│   │   ├── README
│   │   ├── test
│   │   └── train
│   └── glove
│       ├── glove.6B.100d.txt
│       ├── glove.6B.200d.txt
```

```
|       ├── glove.6B.300d.txt
|       └── glove.6B.50d.txt
├── nlp_application.ipynb
└── preprocess


7 directories, 10 files
```

确定评价标准

作为典型的分类问题，情感分类的评价标准可以比照普通的分类问题处理。常见的精度（Accuracy）、精准度（Precision）、召回率（Recall）和F_beta分数都可以作为参考。

精度（$Accuracy$）=分类正确的样本数目/总样本数目精度（Accuracy）=分类正确的样本数目/总样本数目

精准度（$Precision$）=真阳性样本数目/所有预测类别为阳性的样本数目精准度（Precision）=真阳性样本数目/所有预测类别为阳性的样本数目

召回率（$Recall$）=真阳性样本数目/所有真实类别为阳性的样本数目召回率（Recall）=真阳性样本数目/所有真实类别为阳性的样本数目

$F1$分数=(2∗$Precision$∗$Recall$)/($Precision$+$Recall$)F1分数=(2∗Precision∗Recall)/(Precision+Recall)

在IMDB这个数据集中，正负样本数差别不大，可以简单地用精度（accuracy）作为分类器的衡量标准。

确定网络

我们使用基于LSTM构建的SentimentNet网络进行自然语言处理。

> LSTM（Long short-term memory，长短期记忆）网络是一种时间循环神经网络，适合于处理和预测时间序列中间隔和延迟非常长的重要事件。 本次实验面向GPU或CPU硬件平台。

配置运行信息和SentimentNet网络参数

1. 使用`parser`模块传入运行必要的信息。
    - `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GPU或CPU环境。
2. 进行训练前，需要配置必要的信息，包括环境信息、执行的模式、后端信息及硬件信息。

安装`easydict`依赖包。

```
pip install easydict
```

安装`gensim`依赖包。

```
pip install gensim
```

运行以下一段代码中配置训练所需相关参数（详细的接口配置信息，请参见MindSpore官网`context.set_context`API接口说明）。

```
import argparse
from mindspore import context
from easydict import EasyDict as edict


# LSTM CONFIG
lstm_cfg = edict({
    'num_classes': 2,
    'learning_rate': 0.1,
    'momentum': 0.9,
    'num_epochs': 10,
    'batch_size': 64,
    'embed_size': 300,
    'num_hiddens': 100,
    'num_layers': 2,
    'bidirectional': True,
    'save_checkpoint_steps': 390,
    'keep_checkpoint_max': 10
})

cfg = lstm_cfg

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="./datasets/aclImdb",
                    help='path where the dataset is stored.')
parser.add_argument('--glove_path', type=str, default="./datasets/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="./models/ckpt/nlp_application",
                    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="GPU", choices=['GPU',
'CPU'],
                    help='the target device to run, support "GPU", "CPU".
Default: "GPU".')
args = parser.parse_args(['--device_target', 'CPU', '--preprocess', 'true'])

context.set_context(
        mode=context.GRAPH_MODE,
        save_graphs=False,
        device_target=args.device_target)

print("Current context loaded:\n    mode: {}\n    device_target:
{}".format(context.get_context("mode"), context.get_context("device_target")))
```

> Current context loaded:
>
> mode: 0

> device_target: CPU

数据处理

预处理数据集

执行数据集预处理：

- 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。
- 定义`convert_to_mindrecord`函数将数据集格式转换为MindRecord格式，便于MindSpore读取。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。
- 调用`convert_to_mindrecord`函数执行数据集预处理。

```
import os
from itertools import chain
import numpy as np
import gensim
from mindspore.mindrecord import FileWriter


class ImdbParser():
    """
    parse aclImdb data to features and labels.
    sentence->tokenized->encoded->padding->features
    """

    def __init__(self, imdb_path, glove_path, embed_size=300):
        self.__segs = ['train', 'test']
        self.__label_dic = {'pos': 1, 'neg': 0}
        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')

        # properties
        self.__imdb_datas = {}
        self.__features = {}
        self.__labels = {}
        self.__vacab = {}
        self.__word2idx = {}
        self.__weight_np = {}
        self.__wvmodel = None

    def parse(self):
        """
        parse imdb data to memory
        """
        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):
        """
        load data from txt
        """
        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):
                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):
        """
        parse features and labels
        """
        features = []
        labels = []
        for sentence, label in self.__imdb_datas[seg]:
            features.append(sentence)
            labels.append(label)

        self.__features[seg] = features
        self.__labels[seg] = labels

        # update feature to tokenized
        self.__updata_features_to_tokenized(seg)
        # parse vacab
        self.__parse_vacab(seg)
        # encode feature
        self.__encode_features(seg)
        # padding feature
        self.__padding_features(seg)

    def __updata_features_to_tokenized(self, seg):
        tokenized_features = []
        for sentence in self.__features[seg]:
            tokenized_sentence = [word.lower() for word in sentence.split(" ")]
            tokenized_features.append(tokenized_sentence)
        self.__features[seg] = tokenized_features

    def __parse_vacab(self, seg):
        # vocab
        tokenized_features = self.__features[seg]
        vocab = set(chain(*tokenized_features))
        self.__vacab[seg] = vocab

        # word_to_idx: {'hello': 1, 'world':111, ... '<unk>': 0}
        word_to_idx = {word: i + 1 for i, word in enumerate(vocab)}
        word_to_idx['<unk>'] = 0
        self.__word2idx[seg] = word_to_idx

    def __encode_features(self, seg):
        """ encode word to index """
```

```
        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=500, pad=0):
        """ pad all features to the same length """
        padded_features = []
        for feature in self.__features[seg]:
            if len(feature) >= maxlen:
                padded_feature = feature[:maxlen]
            else:
                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):
        """
        generate weight by gensim
        """
        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):
        """
        return features, labels, and weight
        """
        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



def _convert_to_mindrecord(data_home, features, labels, weight_np=None,
training=True):
    """
    convert imdb dataset to mindrecoed dataset
    """
    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):
        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):
    """
    convert imdb dataset to mindrecoed dataset
    """
    parser = ImdbParser(aclimdb_path, glove_path, embed_size)
    parser.parse()

    if not os.path.exists(preprocess_path):
        print(f"preprocess path {preprocess_path} is not exist")
        os.makedirs(preprocess_path)

    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)

if args.preprocess == "true":
    os.system("rm -f ./preprocess/aclImdb* weight*")
    print("============== Starting Data Pre-processing ==============")
    convert_to_mindrecord(cfg.embed_size, args.aclimdb_path,
args.preprocess_path, args.glove_path)
    print("======================== Successful ======================")
```

```
============== Starting Data Pre-processing ==============

======================== 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，此时查看`preprocess`文件目录结构。

此时文件结构如下：

```
preprocess
├── aclImdb_test.mindrecord0
├── aclImdb_test.mindrecord0.db
├── aclImdb_test.mindrecord1
├── aclImdb_test.mindrecord1.db
├── aclImdb_test.mindrecord2
├── aclImdb_test.mindrecord2.db
├── aclImdb_test.mindrecord3
├── aclImdb_test.mindrecord3.db
├── aclImdb_train.mindrecord0
├── aclImdb_train.mindrecord0.db
├── aclImdb_train.mindrecord1
├── aclImdb_train.mindrecord1.db
├── aclImdb_train.mindrecord2
├── aclImdb_train.mindrecord2.db
├── aclImdb_train.mindrecord3
├── aclImdb_train.mindrecord3.db
└── weight.txt


0 directories, 17 files
```

此时`preprocess`目录下的文件为：

- 名称包含`aclImdb_train.mindrecord`的为转换后的MindRecord格式的训练数据集。
- 名称包含`aclImdb_test.mindrecord`的为转换后的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`。
- 通过`create_dict_iterator`方法创建字典迭代器，读取已创建的数据集`ds_train`中的数据。

运行以下一段代码，创建数据集并读取第1个`batch`中的`label`数据列表，和第1个`batch`中第1个元素的`feature`数据。

```
import os
import mindspore.dataset as ds


def lstm_create_dataset(data_home, batch_size, repeat_num=1, training=True):
    """Data operations."""
    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)

    # apply map operations on images
    data_set = data_set.shuffle(buffer_size=data_set.get_dataset_size())
    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)

iterator = next(ds_train.create_dict_iterator())
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
```

定义网络

1. 导入初始化网络所需模块。
2. 定义需要单层LSTM小算子堆叠的设备类型。
3. 定义 `lstm_default_state` 函数来初始化网络参数及网络状态。
4. 定义 `stack_lstm_default_state` 函数来初始化小算子堆叠需要的初始化网络参数及网络状态。
5. 针对CPU场景，自定义单层LSTM小算子堆叠，来实现多层LSTM大算子功能。
6. 使用 `Cell` 方法，定义网络结构（`SentimentNet` 网络）。
7. 实例化 `SentimentNet`，创建网络，最后输出网络中加载的参数。

```
import math
import numpy as np
from mindspore import Tensor, nn, context, Parameter, ParameterTuple
from mindspore.common.initializer import initializer
import mindspore.ops as ops

STACK_LSTM_DEVICE = ["CPU"]

# Initialize short-term memory (h) and long-term memory (c) to 0
def lstm_default_state(batch_size, hidden_size, num_layers, bidirectional):
    """init default input."""
    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)


class SentimentNet(nn.Cell):
    """Sentiment network structure."""

    def __init__(self,
                 vocab_size,
                 embed_size,
                 num_hiddens,
                 num_layers,
                 bidirectional,
                 num_classes,
                 weight,
                 batch_size):
        super(SentimentNet, self).__init__()
        # Mapp words to vectors
        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)

        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)
        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[499]))
        outputs = self.decoder(encoding)
        return outputs

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)

print(network.parameters_dict(recurse=True))
```

训练并保存模型

运行以下一段代码，创建优化器和损失函数模型，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。

```
from mindspore import Model
from mindspore.train.callback import CheckpointConfig, ModelCheckpoint,
TimeMonitor, LossMonitor
from mindspore.nn import Accuracy
from mindspore import nn

os.system("rm -f {0}/*.ckpt {0}/*.meta".format(args.ckpt_path))
loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
opt = nn.Momentum(network.trainable_params(), cfg.learning_rate, cfg.momentum)
model = Model(network, loss, opt, {'acc': Accuracy()})
```

```
loss_cb = LossMonitor(per_print_times=78)
print("============== Starting Training ==============")
config_ck = CheckpointConfig(save_checkpoint_steps=cfg.save_checkpoint_steps,
                             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":
    model.train(cfg.num_epochs, ds_train, callbacks=[time_cb, ckpoint_cb,
loss_cb], dataset_sink_mode=False)
else:
    model.train(cfg.num_epochs, ds_train, callbacks=[time_cb, ckpoint_cb,
loss_cb])
print("============== Training Success ==============")
```

```
============== Starting Training ==============

epoch: 1 step: 78, loss is 0.2971678

epoch: 1 step: 156, loss is 0.30519545

... ...

epoch: 10 step: 312, loss is 0.050257515

epoch: 10 step: 390, loss is 0.025655827

Epoch time: 27546.935, per step time: 70.633

============== Training Success ==============
```

模型验证

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。

```
from mindspore import load_checkpoint, load_param_into_net
args.ckpt_path_saved = f'{args.ckpt_path}/lstm-{cfg.num_epochs}_390.ckpt'
print("============== Starting Testing ==============")
ds_eval = lstm_create_dataset(args.preprocess_path, cfg.batch_size,
training=False)
param_dict = load_checkpoint(args.ckpt_path_saved)
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))
```

训练结果评价

根据以上一段代码的输出可以看到，在经历了10轮epoch之后，使用验证的数据集，对文本的情感分析正确率在85%左右，达到一个基本满意的结果。

总结

以上便完成了MindSpore自然语言处理应用的体验，我们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。

实验要求：

1. 按实验手册完成实验
2. 编写实验报告（内容包括但不限于实验内容、实验思路、实现过程、结果截图展示、实验总结等）

   实验报告命名方式：**学号-姓名-实验序号**，例如：111702xxxxx -王xx-实验三
3. **5月29日中午12:00之前**将程序代码（ipynb文件）、实验报告三并打包上传至邮箱。

压缩包命名同实验报告

邮箱：NLPSpring2022@163.com

邮件主题：**姓名+实验序号** 例：王xx实验三