ERNIE实现对话情绪识别

1 概述

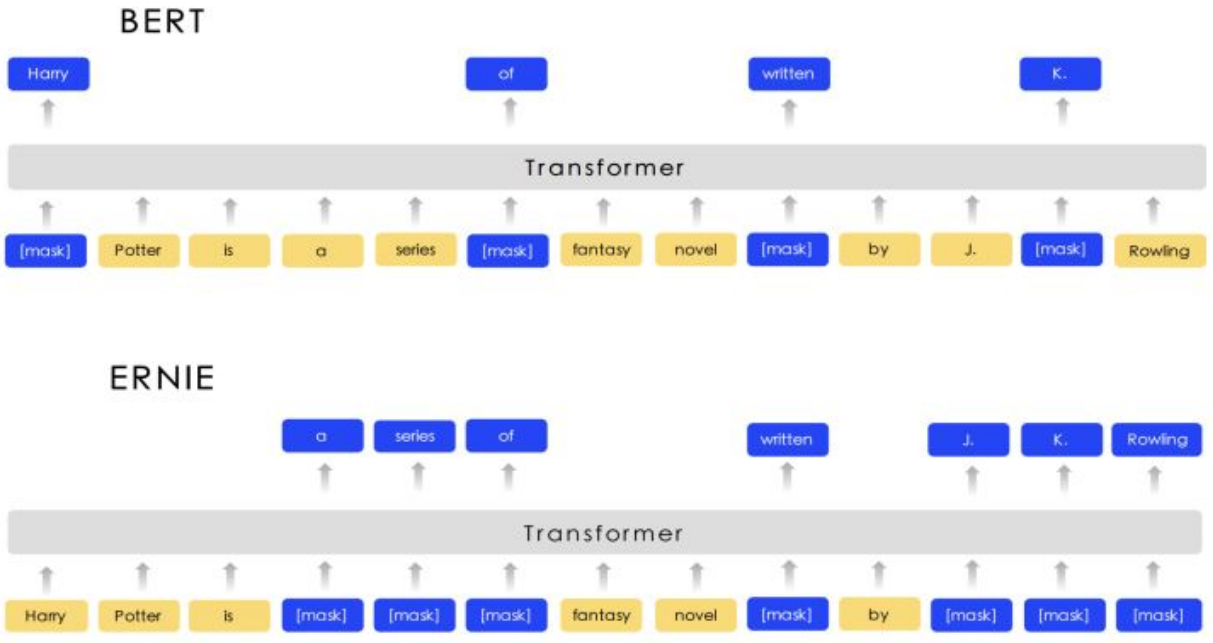
1.1 对话情绪识别

对话情绪识别（Emotion Detection，简称EmoTect），专注于识别智能对话场景中用户的情绪，针对智能对话场景中的用户文本，自动判断该文本的情绪类别并给出相应的置信度，情绪类型分为积极、消极、中性。对话情绪识别适用于聊天、客服等多个场景，能够帮助企业更好地把握对话质量、改善产品的用户交互体验，也能分析客服服务质量、降低人工质检成本。主要实现以下效果：

```
输入：今天天气真好
正确标签：积极
预测标签：积极

输入：今天是晴天
正确标签：中性
预测标签：中性

输入：今天天气也太差了
正确标签：消极
预测标签：消极
```

1.2 ERNIE

ERNIE: Enhanced Representation through Knowledge Integration 是百度在2019年3月的时候，基于BERT模型做的进一步优化，在中文的NLP任务上得到了state-of-the-art的结果。ERNIE 是一种基于知识增强的持续学习语义理解框架，该框架将大数据预训练与多源丰富知识相结合，通过持续学习技术，不断吸收海量文本数据中词汇、结构、语义等方面的知识，实现模型效果不断进化。ERNIE 在情感分析、文本匹配、自然语言推理、词法分析、阅读理解、智能问答等公开数据集上取得了优异的成绩。ERNIE 在工业界也得到了大规模应用，如搜索引擎、新闻推荐、广告系统、语音交互、智能客服等。ERNIE目前仍在不断优化中，本文基于ERNIE1.0版本开发完成。



1.3 环境要求

本文在MindSpore框架下实现了ERNIE模型，通过加载百度开放的标注机器人聊天数据集完成了模型训练、测试及预测。需要的环境如下：

- 硬件
  - 10G以上GPU显存或对应配置的Ascend服务器

- 软件
  - Python 3.5以上
  - MindSpore 1.8 (详情查看MindSpore教程)

1.4 包引入

我们这里提前将后续构建模型所需要的包引入。

```
import io
import os
import math
import copy
import json
import shutil
import argparse
import collections

import numpy as np

import paddle.fluid.dygraph as D
from paddle import fluid


import mindspore.nn as nn
from mindspore import context

from mindspore.mindrecord import FileWriter

from mindspore.common.initializer import TruncatedNormal, initializer
from mindspore.common.tensor import Tensor
from mindspore.common.parameter import Parameter
import mindspore.common.dtype as mstype

from mindspore.nn.wrap.loss_scale import DynamicLossScaleUpdateCell
from mindspore.nn.optim import Adam, AdamWeightDecay, Adagrad

from mindspore.train.model import Model
from mindspore.train.callback import CheckpointConfig, ModelCheckpoint, TimeMonitor


from mindspore.ops import operations as P
from mindspore.ops import functional as F
from mindspore.ops import composite as C
```

2 数据集预处理

本节使用百度提供的一份已标注的、经过分词预处理的机器人对话情绪识别数据集，其目录结构如下：

```
.
├── train.tsv       # 训练集
├── dev.tsv         # 验证集
├── test.tsv        # 测试集
├── infer.tsv       # 待预测数据
├── vocab.txt       # 词典
```

数据由两列组成，以制表符（'\t'）分隔，第一列是情绪分类的类别（0表示消极；1表示中性；2表示积极），第二列是以空格分词的中文文本，如下示例，文件为 utf8 编码。

label	text_a
0	谁 骂人 了？ 我 从来 不 骂人， 我 骂 的 都 不是 人， 你 是 人 吗？
1	我 有事 等会儿 就 回来 和 你 聊
2	我 见到 你 很高兴 谢谢 你 帮 我

2.1 数据下载

为了方便数据集和预训练词向量的下载，首先设计数据下载模块，实现下载流程，并保存至指定路径。我们通过`wget`工具来发起http请求并下载数据集，下载好的数据集为tar.gz文件，利用`tar`工具解压下载下载的数据集到指定位置，并将所有数据和标签分别进行存放。

> 说明：需要提前配置`wget`和`tar`软件，Ubuntu系列安装命令为:`apt-get install wget tar`

```
!DATA_URL=https://baidu-nlp.bj.bcebos.com/emotion_detection-dataset-1.0.0.tar.gz '配置下载链接
!wget --no-check-certificate ${DATA_URL} '下载上述链接的文件

!tar xvf emotion_detection-dataset-1.0.0.tar.gz '解压文件
!/bin/rm emotion_detection-dataset-1.0.0.tar.gz '删除原有压缩文件
```

2.2 MindSpore数据转换

下载数据后，现有数据是tsv文件格式，要加载数据到MindSpore，并按照特定格式获取批处理数据，需要转换成MindSpore独有的MindRecord数据格式。
运行数据格式转换脚本, 将数据集转为MindRecord格式。

首先我们定义一个tsv格式的`reader`，按照`tuple`格式读取tsv文件中的每一行数据

```
def csv_reader(fd, delimiter='\t'):
    """
    csv 文件读取
    """
    def gen():
        for i in fd:
            slots = i.rstrip('\n').split(delimiter)
            if len(slots) == 1:
                yield (slots,)
            else:
                yield slots
    return gen()
```

在实际处理过程中，数据中还可能会出现一些`bytes`格式的数据，为了处理这些数据中的乱码问题，我们定义一些数据格式转换方法。

```
import unicodedata

def convert_to_unicode(text):
    """Converts `text` to Unicode (if it's not already), assuming utf-8 input."""
    if isinstance(text, str):
        text = text
    elif isinstance(text, bytes):
        text = text.decode("utf-8", "ignore")
    else:
        raise ValueError("Unsupported string type: %s" % (type(text)))
    return text

def load_vocab(vocab_file):
    """Loads a vocabulary file into a dictionary."""
    vocab = collections.OrderedDict()
    fin = io.open(vocab_file, encoding="utf8")
    for num, line in enumerate(fin):
```

```
        items = convert_to_unicode(line.strip()).split("\t")
        if len(items) > 2:
            break
        token = items[0]
        index = items[1] if len(items) == 2 else num
        token = token.strip()
        vocab[token] = int(index)
    return vocab


def convert_by_vocab(vocab, items):
    """Converts a sequence of [tokens|ids] using the vocab."""
    output = []
    for item in items:
        output.append(vocab[item])
    return output


def whitespace_tokenize(text):
    """Runs basic whitespace cleaning and splitting on a piece of text."""
    text = text.strip()
    if not text:
        return []
    tokens = text.split()
    return tokens


def _is_whitespace(char):
    """Checks whether `chars` is a whitespace character."""
    # \t, \n, and \r are technically control characters but we treat them
    # as whitespace since they are generally considered as such.
    if char in (" ", "\t", "\n", "\r"):
        return True
    cat = unicodedata.category(char)
    if cat == "Zs":
        return True
    return False


def _is_control(char):
    """Checks whether `chars` is a control character."""
    # These are technically control characters but we count them as whitespace
    # characters.
    if char in ("\t", "\n", "\r"):
        return False
    cat = unicodedata.category(char)
    if cat.startswith("C"):
        return True
    return False


def _is_punctuation(char):
    """Checks whether `chars` is a punctuation character."""
    cp = ord(char)
    # We treat all non-letter/number ASCII as punctuation.
    # Characters such as "^", "$", and "`" are not in the Unicode
    # Punctuation class but we treat them as punctuation anyways, for
    # consistency.
    if ((33 <= cp <= 47) or
            (58 <= cp <= 64) or
            (91 <= cp <= 96) or
            (123 <= cp <= 126)):
        return True
    cat = unicodedata.category(char)
    if cat.startswith("P"):
        return True
    return False
```

定义`BasicTokenizer`类，能够执行基本标记化（标点符号分割、小写等）。

```
class BasicTokenizer:
    """Runs basic tokenization (punctuation splitting, lower casing, etc.)."""

    def __init__(self, do_lower_case=True):
        """Constructs a BasicTokenizer.
        Args:
            do_lower_case: Whether to lower case the input.
        """
        self.do_lower_case = do_lower_case

    def tokenize(self, text):
        """Tokenizes a piece of text."""
        text = convert_to_unicode(text)
        text = self._clean_text(text)

        # This was added on November 1st, 2018 for the multilingual and Chinese
        # models. This is also applied to the English models now, but it doesn't
        # matter since the English models were not trained on any Chinese data
        # and generally don't have any Chinese data in them (there are Chinese
        # characters in the vocabulary because Wikipedia does have some Chinese
        # words in the English Wikipedia.).
        text = self._tokenize_chinese_chars(text)

        orig_tokens = whitespace_tokenize(text)
        split_tokens = []
        for token in orig_tokens:
            if self.do_lower_case:
                token = token.lower()
                token = self._run_strip_accents(token)
            split_tokens.extend(self._run_split_on_punc(token))

        output_tokens = whitespace_tokenize(" ".join(split_tokens))
        return output_tokens

    def _run_strip_accents(self, text):
        """Strips accents from a piece of text."""
        text = unicodedata.normalize("NFD", text)
        output = []
        for char in text:
            cat = unicodedata.category(char)
            if cat == "Mn":
                continue
            output.append(char)
        return "".join(output)

    def _run_split_on_punc(self, text):
        """Splits punctuation on a piece of text."""
        chars = list(text)
        i = 0
        start_new_word = True
        output = []
        while i < len(chars):
            char = chars[i]
            if _is_punctuation(char):
                output.append([char])
                start_new_word = True
            else:
                if start_new_word:
                    output.append([])
                start_new_word = False
                output[-1].append(char)
            i += 1

        return ["".join(x) for x in output]
```

```
    def _tokenize_chinese_chars(self, text):
        """Adds whitespace around any CJK character."""
        output = []
        for char in text:
            cp = ord(char)
            if self._is_chinese_char(cp):
                output.append(" ")
                output.append(char)
                output.append(" ")
            else:
                output.append(char)
        return "".join(output)

    def _is_chinese_char(self, cp):
        """Checks whether CP is the codepoint of a CJK character."""
        # This defines a "chinese character" as anything in the CJK Unicode block:
        #     https://en.wikipedia.org/wiki/CJK_Unified_Ideographs_(Unicode_block)
        #
        # Note that the CJK Unicode block is NOT all Japanese and Korean characters,
        # despite its name. The modern Korean Hangul alphabet is a different block,
        # as is Japanese Hiragana and Katakana. Those alphabets are used to write
        # space-separated words, so they are not treated specially and handled
        # like the all of the other languages.
        if ((0x4E00 <= cp <= 0x9FFF) or
                (0x3400 <= cp <= 0x4DBF) or
                (0x20000 <= cp <= 0x2A6DF) or
                (0x2A700 <= cp <= 0x2B73F) or
                (0x2B740 <= cp <= 0x2B81F) or
                (0x2B820 <= cp <= 0x2CEAF) or
                (0xF900 <= cp <= 0xFAFF) or
                (0x2F800 <= cp <= 0x2FA1F)):
            return True

        return False

    def _clean_text(self, text):
        """Performs invalid character removal and whitespace cleanup on text."""
        output = []
        for char in text:
            cp = ord(char)
            if cp == 0 or cp == 0xfffd or _is_control(char):
                continue
            if _is_whitespace(char):
                output.append(" ")
            else:
                output.append(char)
        return "".join(output)
```

定义`WordpieceTokenizer`类，该类能够将一段文本标记为其词条。使用贪婪的最长匹配优先算法来执行标记化使用给定的词汇。

```
class WordpieceTokenizer:
    """Runs WordPiece tokenziation."""

    def __init__(self, vocab, unk_token="[UNK]", max_input_chars_per_word=100):
        self.vocab = vocab
        self.unk_token = unk_token
        self.max_input_chars_per_word = max_input_chars_per_word

    def tokenize(self, text):
        """Tokenizes a piece of text into its word pieces.
        This uses a greedy longest-match-first algorithm to perform tokenization
        using the given vocabulary.
        For example:
            input = "unaffable"
            output = ["un", "##aff", "##able"]
        Args:
```

```
            text: A single token or whitespace separated tokens. This should have
                already been passed through `BasicTokenizer.
        Returns:
            A list of wordpiece tokens.
        """

        text = convert_to_unicode(text)

        output_tokens = []
        for token in whitespace_tokenize(text):
            chars = list(token)
            if len(chars) > self.max_input_chars_per_word:
                output_tokens.append(self.unk_token)
                continue

            is_bad = False
            start = 0
            sub_tokens = []
            while start < len(chars):
                end = len(chars)
                cur_substr = None
                while start < end:
                    substr = "".join(chars[start:end])
                    if start > 0:
                        substr = "##" + substr
                    if substr in self.vocab:
                        cur_substr = substr
                        break
                    end -= 1
                if cur_substr is None:
                    is_bad = True
                    break
                sub_tokens.append(cur_substr)
                start = end

            if is_bad:
                output_tokens.append(self.unk_token)
            else:
                output_tokens.extend(sub_tokens)
        return output_tokens
```

我们定义一个`FullTokenizer`类将上述的`BasickTokenizer`类和`WordPieceTokenizer`类结合起来。定义`tokenize()`方法将输入的文本分别进行`BasicTokenizer`和`WordPieceTokenizer`的`tokenize()`处理，得到处理分割后的文本词组；定义`convert_tokens_to_ids()`方法通过vocab字典转换成对应id。

```
class FullTokenizer:
    """Runs end-to-end tokenziation."""

    def __init__(self, vocab_file, do_lower_case=True):
        self.vocab = load_vocab(vocab_file)
        self.inv_vocab = {v: k for k, v in self.vocab.items()}
        self.basic_tokenizer = BasicTokenizer(do_lower_case=do_lower_case)
        self.wordpiece_tokenizer = WordpieceTokenizer(vocab=self.vocab)

    def tokenize(self, text):
        split_tokens = []
        for token in self.basic_tokenizer.tokenize(text):
            for sub_token in self.wordpiece_tokenizer.tokenize(token):
                split_tokens.append(sub_token)

        return split_tokens

    def convert_tokens_to_ids(self, tokens):
        return convert_by_vocab(self.vocab, tokens)
```

定义相关转换后，我们定义一个基本的文件转换类型对象`BaseReader`，该对象将tsv文件中的数据读取出来，为了更好处理句子为向量，统一句子长度为`max_seq_len`，多的截断，少的补0，为了衡量句子的真实长度，对有内容的部分赋1，从而获得句子真实长度。由于本文是基于文本对话情绪识别，文本数据中存在有多条句子存在的情况。为了衡量对话顺序，使用`[CLS]`作为句子的开头，使用`[SEP]`作为对话间句子分割。为了表明句子的对话发言情况，使用0代表对话中的第一句话，使用1代表该句的回复。

有如下定义：

- tokens：英文做wordpiece拆分原型和变形
- segment_ids(token_type_id): 表示第一句话还是第二句话
- input_ids: look up vocab找的每个字的索引，因为max_seq_len的存在，需要padding长度不够的位置补0
- input_mask: mask在有字的地方全是1，没有字的地方是0（0的位置后续不参与attention）

```
class BaseReader:
    """BaseReader for classify and sequence labeling task"""

    def __init__(self,
                 vocab_path,
                 label_map_config=None,
                 max_seq_len=512,
                 do_lower_case=True,
                 in_tokens=False,
                 random_seed=None):
        self.max_seq_len = max_seq_len
        self.tokenizer = FullTokenizer(
            vocab_file=vocab_path, do_lower_case=do_lower_case)
        self.vocab = self.tokenizer.vocab
        self.pad_id = self.vocab["[PAD]"]
        self.cls_id = self.vocab["[CLS]"]
        self.sep_id = self.vocab["[SEP]"]
        self.in_tokens = in_tokens

        np.random.seed(random_seed)

        self.current_example = 0
        self.current_epoch = 0
        self.num_examples = 0

        if label_map_config:
            with open(label_map_config) as f:
                self.label_map = json.load(f)
        else:
            self.label_map = None

    def _read_tsv(self, input_file, quotechar=None):
        """Reads a tab separated value file."""
        with io.open(input_file, "r", encoding="utf8") as f:
            reader = csv_reader(f, delimiter="\t")
            headers = next(reader)
            Example = collections.namedtuple('Example', headers)

            examples = []
            for line in reader:
                example = Example(*line)
                examples.append(example)
            return examples

    def _truncate_seq_pair(self, tokens_a, tokens_b, max_length):
        """Truncates a sequence pair in place to the maximum length."""

        # This is a simple heuristic which will always truncate the longer sequence
        # one token at a time. This makes more sense than truncating an equal percent
        # of tokens from each, since if one sequence is very short then each token
        # that's truncated likely contains more information than a longer sequence.
        while True:
            total_length = len(tokens_a) + len(tokens_b)
            if total_length <= max_length:
```

```
                break
            if len(tokens_a) > len(tokens_b):
                tokens_a.pop()
            else:
                tokens_b.pop()

    def _convert_example_to_record(self, example, max_seq_length, tokenizer):
        """Converts a single `Example` into a single `Record`."""

        text_a = convert_to_unicode(example.text_a)
        tokens_a = tokenizer.tokenize(text_a)
        tokens_b = None
        if "text_b" in example._fields:
            text_b = convert_to_unicode(example.text_b)
            tokens_b = tokenizer.tokenize(text_b)

        if tokens_b:
            # Modifies `tokens_a` and `tokens_b` in place so that the total
            # length is less than the specified length.
            # Account for [CLS], [SEP], [SEP] with "- 3"
            self._truncate_seq_pair(tokens_a, tokens_b, max_seq_length - 3)
        else:
            # Account for [CLS] and [SEP] with "- 2"
            if len(tokens_a) > max_seq_length - 2:
                tokens_a = tokens_a[0:(max_seq_length - 2)]

        # The convention in BERT/ERNIE is:
        # (a) For sequence pairs:
        #  tokens:   [CLS] is this jack ##son ##ville ? [SEP] no it is not . [SEP]
        #  type_ids: 0     0  0    0    0     0       0 0     1  1  1  1   1 1
        # (b) For single sequences:
        #  tokens:   [CLS] the dog is hairy . [SEP]
        #  type_ids: 0     0   0   0  0     0 0
        #
        # Where "type_ids" are used to indicate whether this is the first
        # sequence or the second sequence. The embedding vectors for `type=0` and
        # `type=1` were learned during pre-training and are added to the wordpiece
        # embedding vector (and position vector). This is not *strictly* necessary
        # since the [SEP] token unambiguously separates the sequences, but it makes
        # it easier for the model to learn the concept of sequences.
        #
        # For classification tasks, the first vector (corresponding to [CLS]) is
        # used as as the "sentence vector". Note that this only makes sense because
        # the entire model is fine-tuned.
        tokens = []
        segment_ids = []
        tokens.append("[CLS]")
        segment_ids.append(0)
        for token in tokens_a:
            tokens.append(token)
            segment_ids.append(0)
        tokens.append("[SEP]")
        segment_ids.append(0)

        if tokens_b:
            for token in tokens_b:
                tokens.append(token)
                segment_ids.append(1)
            tokens.append("[SEP]")
            segment_ids.append(1)

        input_ids = tokenizer.convert_tokens_to_ids(tokens)

        input_mask = [1] * len(input_ids)

        while len(input_ids) < max_seq_length:
            input_ids.append(0)
            input_mask.append(0)
```

```
            segment_ids.append(0)

        if self.label_map:
            label_id = self.label_map[example.label]
        else:
            label_id = example.label

        Record = collections.namedtuple(
            'Record',
            ['input_ids', 'input_mask', 'segment_ids', 'label_id'])

        record = Record(
            input_ids=input_ids,
            input_mask=input_mask,
            segment_ids=segment_ids,
            label_id=label_id)
        return record

    def get_num_examples(self, input_file):
        """return total number of examples"""
        examples = self._read_tsv(input_file)
        return len(examples)

    def get_examples(self, input_file):
        examples = self._read_tsv(input_file)
        return examples

    def file_based_convert_examples_to_features(self, input_file, output_file):
        """Convert a set of `InputExample`s to a MindDataset file."""
        examples = self._read_tsv(input_file)

        writer = FileWriter(file_name=output_file, shard_num=1)
        nlp_schema = {
            "input_ids": {"type": "int64", "shape": [-1]},
            "input_mask": {"type": "int64", "shape": [-1]},
            "segment_ids": {"type": "int64", "shape": [-1]},
            "label_ids": {"type": "int64", "shape": [-1]},
        }
        writer.add_schema(nlp_schema, "proprocessed classification dataset")
        data = []
        for index, example in enumerate(examples):
            if index % 10000 == 0:
                print("Writing example %d of %d" % (index, len(examples)))
            record = self._convert_example_to_record(example, self.max_seq_len, self.tokenizer)
            sample = {
                "input_ids": np.array(record.input_ids, dtype=np.int64),
                "input_mask": np.array(record.input_mask, dtype=np.int64),
                "segment_ids": np.array(record.segment_ids, dtype=np.int64),
                "label_ids": np.array([record.label_id], dtype=np.int64),
            }
            data.append(sample)
        writer.write_raw_data(data)
        writer.commit()
```

为了适配我们这里使用的tsv文件格式，我们继承上述`BaseReader`并定义新的`_read_tsv`方法来读取tsv文件数据格式，首先获取tsv文件的列名编号，在利用到csv文件的`reader`将所有行中的数据中的空格去掉，从而形成一句完整的句子。

```
class ClassifyReader(BaseReader):
    """ClassifyReader"""

    def _read_tsv(self, input_file, quotechar=None):
        """Reads a tab separated value file."""
        with io.open(input_file, "r", encoding="utf8") as f:
            reader = csv_reader(f, delimiter="\t")
            headers = next(reader)
            text_indices = [
```

```
                index for index, h in enumerate(headers) if h != "label"
            ]
            Example = collections.namedtuple('Example', headers)

            examples = []
            for line in reader:
                for index, text in enumerate(line):
                    if index in text_indices:
                        line[index] = text.replace(' ', '')
                example = Example(*line)
                examples.append(example)
            return examples
```

最后我们定义一个数据转换方法，对`train.tsv`、`test.tsv`、`dev.tsv`三个文件分别定义一个数据转换格式的`ClassifyReader`对象加载数据内容，然后调用`file_based_convert_examples_to_features()`方法，从而将每个tsv文件都转换为对应的MindRecord。

```
def convert_tsv_2_mindrecord(config):
    for file in ['train.tsv','test.tsv','dev.tsv']:
        reader = ClassifyReader(
            vocab_path=args_opt.vocab_path,
            label_map_config=args_opt.label_map_config,
            max_seq_len=args_opt.max_seq_len,
            do_lower_case=args_opt.do_lower_case,
            random_seed=args_opt.random_seed
        )
        reader.file_based_convert_examples_to_features(input_file=args_opt.input_file,
output_file=args_opt.output_file)
```

2.3 加载对话数据集

在将数据转换为对应的MindRecord之后，我们还需要将数据加载到`MindDataset`对象中，该对象能够将`MindRecord`中的数据按照给定的`Schema`数据类型读取到内存中，需将其加入到数据集处理流水线中，使用`map`接口对指定的column添加操作。最后指定数据集的batch大小，通过`batch`接口指定，并设置是否丢弃无法被batch size整除的剩余数据。

```
import mindspore.dataset as ds
import mindspore.dataset.transforms as T

def create_classification_dataset(batch_size=1,
                                  repeat_count=1,
                                  data_file_path=None,
                                  schema_file_path=None,
                                  do_shuffle=True,
                                  drop_remainder=True):
    """create finetune or evaluation dataset"""
    type_cast_op = T.TypeCast(mstype.int32)
    data_set = ds.MindDataset([data_file_path],
                              columns_list=["input_ids", "input_mask", "segment_ids",
"label_ids"],
                              shuffle=do_shuffle)
    data_set = data_set.map(operations=type_cast_op, input_columns="label_ids")
    data_set = data_set.map(operations=type_cast_op, input_columns="segment_ids")
    data_set = data_set.map(operations=type_cast_op, input_columns="input_mask")
    data_set = data_set.map(operations=type_cast_op, input_columns="input_ids")
    data_set = data_set.repeat(repeat_count)
    # apply batch operations
    data_set = data_set.batch(batch_size, drop_remainder=drop_remainder)
    return data_set
```

3 ERNIE模型构建

完成数据集的处理后，我们设计用于对话情绪识别的模型结构。首先需要将输入文本(即序列化后的index id列表)通过查表转为向量化表示，此时需要使用`nn.Embedding`层加载之前构建的词向量`input_ids`；然后使用RNN循环神经网络做特征提取；最后将RNN连接至一个全连接层，即nn.Dense，将特征转化为与分类数量相同的size，用于后续进行模型优化训练。整体模型结构如下：

下面对模型进行详解：

3.1 Embedding

3.1.1 EmbeddingLookup

其作用是使用index id对权重矩阵对应id的向量进行查找，当输入为一个由index id组成的序列时，则查找并返回一个相同长度的矩阵。这里我们只需要利用一个线性函数即可完成。

```
ernie_embedding_lookup = nn.Embedding(
            vocab_size=config.vocab_size,
            embedding_size=self.embedding_size,
            use_one_hot=use_one_hot_embeddings)
```

3.1.2 Embedding_postprocess

这一层包含有两部分，一是句子间序列信息token_type_embedding层，二是句子内部序列信息full_position_embedding层。

1. token_type_embedding层

token_type表的维度是[2,768]只有`0，1`两种可能性)，对`token_type_id`做`one_hot`再和`token_type`表相乘得到`token_type_embedding`,其`shape`是[8,128,768]。最后把token_type_embedding和之前的ernie_embedding_lookup相加，把位置信息融入input。

2. full_position_embedding层

对于batch里的所有样本，位置嵌入的信息是随机初始化的，值都是从1到128的位置，postion_embedding的维度是[128,768]，batch里的每一个样本都需要加上position_embedding

3. layer_normalization 和dropout

当我们使用梯度下降法做优化时，随着网络深度的增加，输入数据的特征分布会不断发生变化，为了保证数据特征分布的稳定性，这里加入了Layer Normalization。Normalization的主要作用就是把每层特征输入到激活函数之前，对它们进行normalization，使其转换为均值为1，方差为0的数据，从而可以避免数据落在激活函数的饱和区，以减少梯度消失的问题。

Dropout随机遮蔽一些神经元，防止模型过拟合。

```
class EmbeddingPostprocessor(nn.Cell):
    """
    Postprocessors apply positional and token type embeddings to word embeddings.
    Args:
        embedding_size (int): The size of each embedding vector.
        embedding_shape (list): [batch_size, seq_length, embedding_size], the shape of
                        each embedding vector.
        use_token_type (bool): Specifies whether to use token type embeddings. Default: False.
        token_type_vocab_size (int): Size of token type vocab. Default: 16.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        max_position_embeddings (int): Maximum length of sequences used in this
                                model. Default: 512.
        dropout_prob (float): The dropout probability. Default: 0.1.
    """
    def __init__(self,
                 embedding_size,
                 embedding_shape,
                 use_relative_positions=False,
                 use_token_type=False,
                 token_type_vocab_size=16,
                 use_one_hot_embeddings=False,
                 initializer_range=0.02,
                 max_position_embeddings=512,
                 dropout_prob=0.1):
        super(EmbeddingPostprocessor, self).__init__()
        self.use_token_type = use_token_type
        self.token_type_vocab_size = token_type_vocab_size
        self.use_one_hot_embeddings = use_one_hot_embeddings
```

```
        self.max_position_embeddings = max_position_embeddings
        self.token_type_embedding = nn.Embedding(
            vocab_size=token_type_vocab_size,
            embedding_size=embedding_size,
            use_one_hot=use_one_hot_embeddings)
        self.shape_flat = (-1,)
        self.one_hot = P.OneHot()
        self.on_value = Tensor(1.0, mstype.float32)
        self.off_value = Tensor(0.1, mstype.float32)
        self.array_mul = P.MatMul()
        self.reshape = P.Reshape()
        self.shape = tuple(embedding_shape)
        self.dropout = nn.Dropout(1 - dropout_prob)
        self.gather = P.Gather()
        self.use_relative_positions = use_relative_positions
        self.slice = P.StridedSlice()
        _, seq, _ = self.shape
        self.full_position_embedding = nn.Embedding(
            vocab_size=max_position_embeddings,
            embedding_size=embedding_size,
            use_one_hot=False)
        self.layernorm = nn.LayerNorm((embedding_size,))
        self.position_ids = Tensor(np.arange(seq).reshape(-1, seq).astype(np.int32))
        self.add = P.Add()

    def construct(self, token_type_ids, word_embeddings):
        """Postprocessors apply positional and token type embeddings to word embeddings."""
        output = word_embeddings
        if self.use_token_type:
            token_type_embeddings = self.token_type_embedding(token_type_ids)
            output = self.add(output, token_type_embeddings)
        if not self.use_relative_positions:
            position_embeddings = self.full_position_embedding(self.position_ids)
            output = self.add(output, position_embeddings)
        output = self.layernorm(output)
        output = self.dropout(output)
        return output
```

3.2 Encoder

3.2.1 attention mask

之前我们讲到mask在有字的地方全是1，没有字的地方是0，这里做的事情是：对于2维的[8, 128]的句子矩阵，每一个元素是词的id，对于每一个元素的位置，我们替换成一个向量，这个向量大小128，就是当前这个字所在的句子的一个mask信息,大概长这样[1,1,1,1...1,0,...0]。这样2维的句子矩阵就变成3维了[8, 128, 128]，这个向量是为了，当前这个词在attention的时候能看到哪几个词（0的位置不参与attention）。

```
class CreateAttentionMaskFromInputMask(nn.Cell):
    """
    Create attention mask according to input mask.
    Args:
        config (Class): Configuration for ErnieModel.
    """
    def __init__(self, config):
        super(CreateAttentionMaskFromInputMask, self).__init__()
        self.input_mask = None

        self.cast = P.Cast()
        self.reshape = P.Reshape()
        self.shape = (-1, 1, config.seq_length)

    def construct(self, input_mask):
        attention_mask = self.cast(self.reshape(input_mask, self.shape), mstype.float32)
        return attention_mask
```

3.2.2 ERNIE Self-Attention

这一块可以说是模型的核心区域，也是唯一涉及到公式的地方，所以将贴出大量代码。

首先介绍三个工具类，`RelaPosMatrixGenerator`能够生成输入之间的相对位置矩阵；`RelaPosEmbeddingsGenerator`类能够生成大小为[length, length, depth]的张量。`SaturateCast`类主要用来执行安全饱和投射。此操作在铸造前应用适当的收缩范围，以防止出现值溢出或下溢的错误。

1. RelaPosMatrixGenerator

```
class RelaPosMatrixGenerator(nn.Cell):
    """
    Generates matrix of relative positions between inputs.
    Args:
        length (int): Length of one dim for the matrix to be generated.
        max_relative_position (int): Max value of relative position.
    """
    def __init__(self, length, max_relative_position):
        super(RelaPosMatrixGenerator, self).__init__()
        self._length = length
        self._max_relative_position = max_relative_position
        self._min_relative_position = -max_relative_position
        self.range_length = -length + 1

        self.tile = P.Tile()
        self.range_mat = P.Reshape()
        self.sub = P.Sub()
        self.expanddims = P.ExpandDims()
        self.cast = P.Cast()

    def construct(self):
        """Generates matrix of relative positions between inputs."""
        range_vec_row_out = self.cast(F.tuple_to_array(F.make_range(self._length)), mstype.int32)
        range_vec_col_out = self.range_mat(range_vec_row_out, (self._length, -1))
        tile_row_out = self.tile(range_vec_row_out, (self._length,))
        tile_col_out = self.tile(range_vec_col_out, (1, self._length))
        range_mat_out = self.range_mat(tile_row_out, (self._length, self._length))
        transpose_out = self.range_mat(tile_col_out, (self._length, self._length))
        distance_mat = self.sub(range_mat_out, transpose_out)

        distance_mat_clipped = C.clip_by_value(distance_mat,
                                               self._min_relative_position,
                                               self._max_relative_position)

        # Shift values to be >=0. Each integer still uniquely identifies a
        # relative position difference.
        final_mat = distance_mat_clipped + self._max_relative_position
        return final_mat
```

2. RelaPosEmbeddingsGenerator

```
class RelaPosEmbeddingsGenerator(nn.Cell):
    """
    Generates tensor of size [length, length, depth].
    Args:
        length (int): Length of one dim for the matrix to be generated.
        depth (int): Size of each attention head.
        max_relative_position (int): Maxmum value of relative position.
        initializer_range (float): Initialization value of TruncatedNormal.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
    """
    def __init__(self,
                 length,
                 depth,
```

```
                max_relative_position,
                initializer_range,
                use_one_hot_embeddings=False):
        super(RelaPosEmbeddingsGenerator, self).__init__()
        self.depth = depth
        self.vocab_size = max_relative_position * 2 + 1
        self.use_one_hot_embeddings = use_one_hot_embeddings

        self.embeddings_table = Parameter(
            initializer(TruncatedNormal(initializer_range),
                        [self.vocab_size, self.depth]))

        self.relative_positions_matrix = RelaPosMatrixGenerator(length=length,

 max_relative_position=max_relative_position)
        self.reshape = P.Reshape()
        self.one_hot = nn.OneHot(depth=self.vocab_size)
        self.shape = P.Shape()
        self.gather = P.Gather()  # index_select
        self.matmul = P.BatchMatMul()

    def construct(self):
        """Generate embedding for each relative position of dimension depth."""
        relative_positions_matrix_out = self.relative_positions_matrix()

        if self.use_one_hot_embeddings:
            flat_relative_positions_matrix = self.reshape(relative_positions_matrix_out, (-1,))
            one_hot_relative_positions_matrix = self.one_hot(
                flat_relative_positions_matrix)
            embeddings = self.matmul(one_hot_relative_positions_matrix, self.embeddings_table)
            my_shape = self.shape(relative_positions_matrix_out) + (self.depth,)
            embeddings = self.reshape(embeddings, my_shape)
        else:
            embeddings = self.gather(self.embeddings_table,
                                     relative_positions_matrix_out, 0)
        return embeddings
```

3. SaturateCast

```
class SaturateCast(nn.Cell):
    """
    Performs a safe saturating cast. This operation applies proper clamping before casting to
prevent
    the danger that the value will overflow or underflow.
    Args:
        src_type (:class:`mindspore.dtype`): The type of the elements of the input tensor.
Default: mstype.float32.
        dst_type (:class:`mindspore.dtype`): The type of the elements of the output tensor.
Default: mstype.float32.
    """
    def __init__(self, src_type=mstype.float32, dst_type=mstype.float32):
        super(SaturateCast, self).__init__()
        np_type = mstype.dtype_to_nptype(dst_type)

        self.tensor_min_type = float(np.finfo(np_type).min)
        self.tensor_max_type = float(np.finfo(np_type).max)

        self.min_op = P.Minimum()
        self.max_op = P.Maximum()
        self.cast = P.Cast()
        self.dst_type = dst_type

    def construct(self, x):
        out = self.max_op(x, self.tensor_min_type)
        out = self.min_op(out, self.tensor_max_type)
        return self.cast(out, self.dst_type)
```

3.2.2.1 Attention

其核心内容是为输入向量的每个单词学习一个权重。通过给定一个任务相关的查询向量Query向量，计算Query和各个Key的相似性或者相关性得到注意力分布，即得到每个Key对应Value的权重系数，然后对Value进行加权求和得到最终的Attention数值。

```
class ErnieAttention(nn.Cell):
    """
    Apply multi-headed attention from "from_tensor" to "to_tensor".
    Args:
        from_tensor_width (int): Size of last dim of from_tensor.
        to_tensor_width (int): Size of last dim of to_tensor.
        from_seq_length (int): Length of from_tensor sequence.
        to_seq_length (int): Length of to_tensor sequence.
        num_attention_heads (int): Number of attention heads. Default: 1.
        size_per_head (int): Size of each attention head. Default: 512.
        query_act (str): Activation function for the query transform. Default: None.
        key_act (str): Activation function for the key transform. Default: None.
        value_act (str): Activation function for the value transform. Default: None.
        has_attention_mask (bool): Specifies whether to use attention mask. Default: False.
        attention_probs_dropout_prob (float): The dropout probability for
                                      ErnieAttention. Default: 0.0.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        do_return_2d_tensor (bool): True for return 2d tensor. False for return 3d
                            tensor. Default: False.
        use_relative_positions (bool): Specifies whether to use relative positions. Default:
False.
        compute_type (:class:`mindspore.dtype`): Compute type in ErnieAttention. Default:
mstype.float32.
    """
    def __init__(self,
                 from_tensor_width,
                 to_tensor_width,
                 from_seq_length,
                 to_seq_length,
                 num_attention_heads=1,
                 size_per_head=512,
                 query_act=None,
                 key_act=None,
                 value_act=None,
                 has_attention_mask=False,
                 attention_probs_dropout_prob=0.0,
                 use_one_hot_embeddings=False,
                 initializer_range=0.02,
                 do_return_2d_tensor=False,
                 use_relative_positions=False,
                 compute_type=mstype.float32):

        super(ErnieAttention, self).__init__()
        self.from_seq_length = from_seq_length
        self.to_seq_length = to_seq_length
        self.num_attention_heads = num_attention_heads
        self.size_per_head = size_per_head
        self.has_attention_mask = has_attention_mask
        self.use_relative_positions = use_relative_positions

        self.scores_mul = 1.0 / math.sqrt(float(self.size_per_head))
        self.reshape = P.Reshape()
        self.shape_from_2d = (-1, from_tensor_width)
        self.shape_to_2d = (-1, to_tensor_width)
        weight = TruncatedNormal(initializer_range)
        units = num_attention_heads * size_per_head
        self.query_layer = nn.Dense(from_tensor_width,
                                    units,
                                    activation=query_act,
```

```
                                  weight_init=weight).to_float(compute_type)
        self.key_layer = nn.Dense(to_tensor_width,
                                  units,
                                  activation=key_act,
                                  weight_init=weight).to_float(compute_type)
        self.value_layer = nn.Dense(to_tensor_width,
                                    units,
                                    activation=value_act,
                                    weight_init=weight).to_float(compute_type)

        self.shape_from = (-1, from_seq_length, num_attention_heads, size_per_head)
        self.shape_to = (-1, to_seq_length, num_attention_heads, size_per_head)

        self.matmul_trans_b = P.BatchMatMul(transpose_b=True)
        self.multiply = P.Mul()
        self.transpose = P.Transpose()
        self.trans_shape = (0, 2, 1, 3)
        self.trans_shape_relative = (2, 0, 1, 3)
        self.trans_shape_position = (1, 2, 0, 3)
        self.multiply_data = -10000.0
        self.matmul = P.BatchMatMul()

        self.softmax = nn.Softmax()
        self.dropout = nn.Dropout(1 - attention_probs_dropout_prob)

        if self.has_attention_mask:
            self.expand_dims = P.ExpandDims()
            self.sub = P.Sub()
            self.add = P.Add()
            self.cast = P.Cast()
            self.get_dtype = P.DType()
        if do_return_2d_tensor:
            self.shape_return = (-1, num_attention_heads * size_per_head)
        else:
            self.shape_return = (-1, from_seq_length, num_attention_heads * size_per_head)

        self.cast_compute_type = SaturateCast(dst_type=compute_type)
        if self.use_relative_positions:
            self._generate_relative_positions_embeddings = \
                RelaPosEmbeddingsGenerator(length=to_seq_length,
                                           depth=size_per_head,
                                           max_relative_position=16,
                                           initializer_range=initializer_range,
                                           use_one_hot_embeddings=use_one_hot_embeddings)

    def construct(self, from_tensor, to_tensor, attention_mask):
        """reshape 2d/3d input tensors to 2d"""
        from_tensor_2d = self.reshape(from_tensor, self.shape_from_2d)
        to_tensor_2d = self.reshape(to_tensor, self.shape_to_2d)
        query_out = self.query_layer(from_tensor_2d)
        key_out = self.key_layer(to_tensor_2d)
        value_out = self.value_layer(to_tensor_2d)

        query_layer = self.reshape(query_out, self.shape_from)
        query_layer = self.transpose(query_layer, self.trans_shape)
        key_layer = self.reshape(key_out, self.shape_to)
        key_layer = self.transpose(key_layer, self.trans_shape)

        attention_scores = self.matmul_trans_b(query_layer, key_layer)

        # use_relative_position, supplementary logic
        if self.use_relative_positions:
            # relations_keys is [F|T, F|T, H]
            relations_keys = self._generate_relative_positions_embeddings()
            relations_keys = self.cast_compute_type(relations_keys)
            # query_layer_t is [F, B, N, H]
            query_layer_t = self.transpose(query_layer, self.trans_shape_relative)
            # query_layer_r is [F, B * N, H]
```

```
            query_layer_r = self.reshape(query_layer_t,
                                         (self.from_seq_length,
                                          -1,
                                          self.size_per_head))
            # key_position_scores is [F, B * N, F|T]
            key_position_scores = self.matmul_trans_b(query_layer_r,
                                                      relations_keys)
            # key_position_scores_r is [F, B, N, F|T]
            key_position_scores_r = self.reshape(key_position_scores,
                                                 (self.from_seq_length,
                                                  -1,
                                                  self.num_attention_heads,
                                                  self.from_seq_length))
            # key_position_scores_r_t is [B, N, F, F|T]
            key_position_scores_r_t = self.transpose(key_position_scores_r,
                                                     self.trans_shape_position)
            attention_scores = attention_scores + key_position_scores_r_t

        attention_scores = self.multiply(self.scores_mul, attention_scores)

        if self.has_attention_mask:
            attention_mask = self.expand_dims(attention_mask, 1)
            multiply_out = self.sub(self.cast(F.tuple_to_array((1.0,)),
self.get_dtype(attention_scores)),
                                    self.cast(attention_mask, self.get_dtype(attention_scores)))

            adder = self.multiply(multiply_out, self.multiply_data)
            attention_scores = self.add(adder, attention_scores)

        attention_probs = self.softmax(attention_scores)
        attention_probs = self.dropout(attention_probs)

        value_layer = self.reshape(value_out, self.shape_to)
        value_layer = self.transpose(value_layer, self.trans_shape)
        context_layer = self.matmul(attention_probs, value_layer)

        # use_relative_position, supplementary logic
        if self.use_relative_positions:
            # relations_values is [F|T, F|T, H]
            relations_values = self._generate_relative_positions_embeddings()
            relations_values = self.cast_compute_type(relations_values)
            # attention_probs_t is [F, B, N, T]
            attention_probs_t = self.transpose(attention_probs, self.trans_shape_relative)
            # attention_probs_r is [F, B * N, T]
            attention_probs_r = self.reshape(
                attention_probs_t,
                (self.from_seq_length,
                 -1,
                 self.to_seq_length))
            # value_position_scores is [F, B * N, H]
            value_position_scores = self.matmul(attention_probs_r,
                                                relations_values)
            # value_position_scores_r is [F, B, N, H]
            value_position_scores_r = self.reshape(value_position_scores,
                                                   (self.from_seq_length,
                                                    -1,
                                                    self.num_attention_heads,
                                                    self.size_per_head))
            # value_position_scores_r_t is [B, N, F, H]
            value_position_scores_r_t = self.transpose(value_position_scores_r,
                                                       self.trans_shape_position)
            context_layer = context_layer + value_position_scores_r_t

        context_layer = self.transpose(context_layer, self.trans_shape)
        context_layer = self.reshape(context_layer, self.shape_return)

        return context_layer
```

3.2.2.2 Output

为了便于计算，且方便后续引用，我们将Self-Attention模块中最后一部分的线性计算和残差连接封装为一个单独的类。该模型是一个全连接(Dense)+Dropout+LayerNorm结构。在全连接层我们使用了`TruncatedNormal`来生成生成一个服从截断正态（高斯）分布的随机数组，然后对其进行dropout操作提高泛化性，接着对输入(input)进行残差连接计算(residual connect),最后放入`LayerNorm`正则化输出。

```
class ErnieOutput(nn.Cell):
    """
    Apply a linear computation to hidden status and a residual computation to input.
    Args:
        in_channels (int): Input channels.
        out_channels (int): Output channels.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        dropout_prob (float): The dropout probability. Default: 0.1.
        compute_type (:class:`mindspore.dtype`): Compute type in ErnieTransformer. Default:
mstype.float32.
    """
    def __init__(self,
                 in_channels,
                 out_channels,
                 initializer_range=0.02,
                 dropout_prob=0.1,
                 compute_type=mstype.float32):
        super(ErnieOutput, self).__init__()
        self.dense = nn.Dense(in_channels, out_channels,

 weight_init=TruncatedNormal(initializer_range)).to_float(compute_type)
        self.dropout = nn.Dropout(1 - dropout_prob)
        self.dropout_prob = dropout_prob
        self.add = P.Add()
        self.layernorm = nn.LayerNorm((out_channels,)).to_float(compute_type)
        self.cast = P.Cast()

    def construct(self, hidden_status, input_tensor):
        output = self.dense(hidden_status)
        output = self.dropout(output)
        output = self.add(input_tensor, output)
        output = self.layernorm(output)
        return output
```

3.2.2.3 Self-Attention

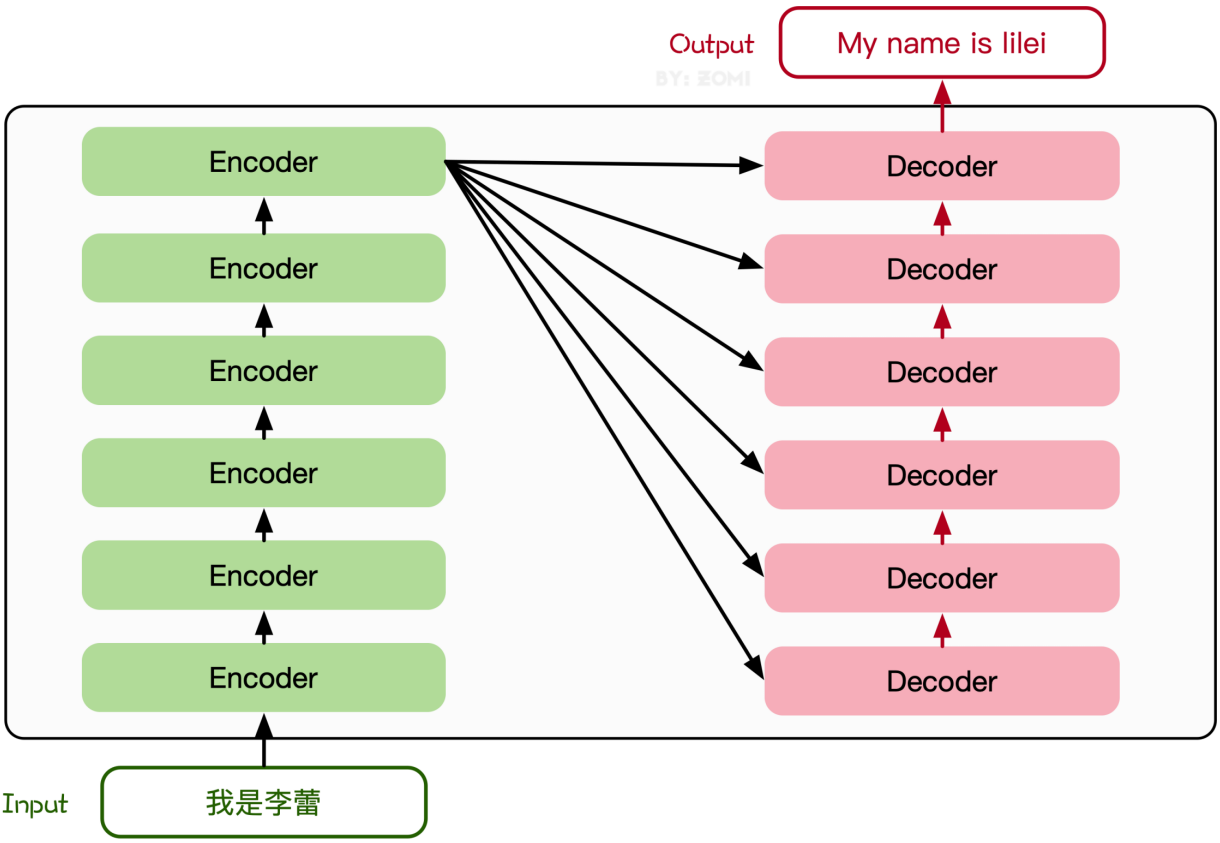
将上述`Attention`类与`Output`模块拼装后，即可完成`Self-Attention`模块的编写

```
class ErnieSelfAttention(nn.Cell):
    """
    Apply self-attention.
    Args:
        seq_length (int): Length of input sequence.
        hidden_size (int): Size of the Ernie encoder layers.
        num_attention_heads (int): Number of attention heads. Default: 12.
        attention_probs_dropout_prob (float): The dropout probability for
                                       ErnieAttention. Default: 0.1.
        use_one_hot_embeddings (bool): Specifies whether to use one_hot encoding form. Default:
False.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        hidden_dropout_prob (float): The dropout probability for ErnieOutput. Default: 0.1.
        use_relative_positions (bool): Specifies whether to use relative positions. Default:
False.
        compute_type (:class:`mindspore.dtype`): Compute type in ErnieSelfAttention. Default:
mstype.float32.
    """
    def __init__(self,
                 seq_length,
```

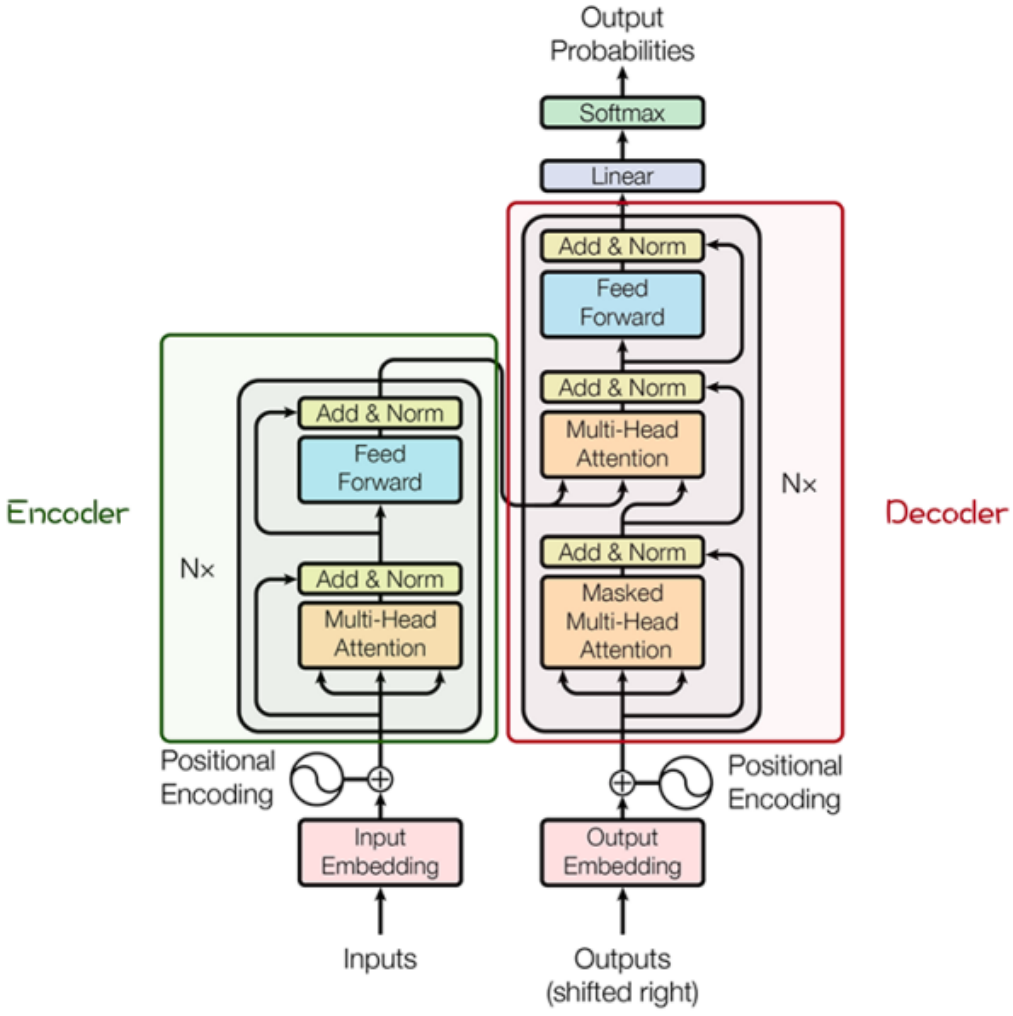
```
                hidden_size,
                num_attention_heads=12,
                attention_probs_dropout_prob=0.1,
                use_one_hot_embeddings=False,
                initializer_range=0.02,
                hidden_dropout_prob=0.1,
                use_relative_positions=False,
                compute_type=mstype.float32):
        super(ErnieSelfAttention, self).__init__()
        if hidden_size % num_attention_heads != 0:
            raise ValueError("The hidden size (%d) is not a multiple of the number "
                             "of attention heads (%d)" % (hidden_size, num_attention_heads))

        self.size_per_head = int(hidden_size / num_attention_heads)

        self.attention = ErnieAttention(
            from_tensor_width=hidden_size,
            to_tensor_width=hidden_size,
            from_seq_length=seq_length,
            to_seq_length=seq_length,
            num_attention_heads=num_attention_heads,
            size_per_head=self.size_per_head,
            attention_probs_dropout_prob=attention_probs_dropout_prob,
            use_one_hot_embeddings=use_one_hot_embeddings,
            initializer_range=initializer_range,
            use_relative_positions=use_relative_positions,
            has_attention_mask=True,
            do_return_2d_tensor=True,
            compute_type=compute_type)

        self.output = ErnieOutput(in_channels=hidden_size,
                                  out_channels=hidden_size,
                                  initializer_range=initializer_range,
                                  dropout_prob=hidden_dropout_prob,
                                  compute_type=compute_type)
        self.reshape = P.Reshape()
        self.shape = (-1, hidden_size)

    def construct(self, input_tensor, attention_mask):
        input_tensor = self.reshape(input_tensor, self.shape)
        attention_output = self.attention(input_tensor, input_tensor, attention_mask)
        output = self.output(attention_output, input_tensor)
        return output
```

3.2.3 ERNIETransformer

ERNIE成功的关键在于运用了Transformer模型，Transformer模型源于2017年的一篇文章[2]。在这篇文章中提出的基于Attention机制的编码器-解码器型结构在自然语言处理领域获得了巨大的成功。模型结构如下图所示：



其主要结构为多个Encoder和Decoder模块所组成，其中Encoder和Decoder的详细结构如下图[2]所示：



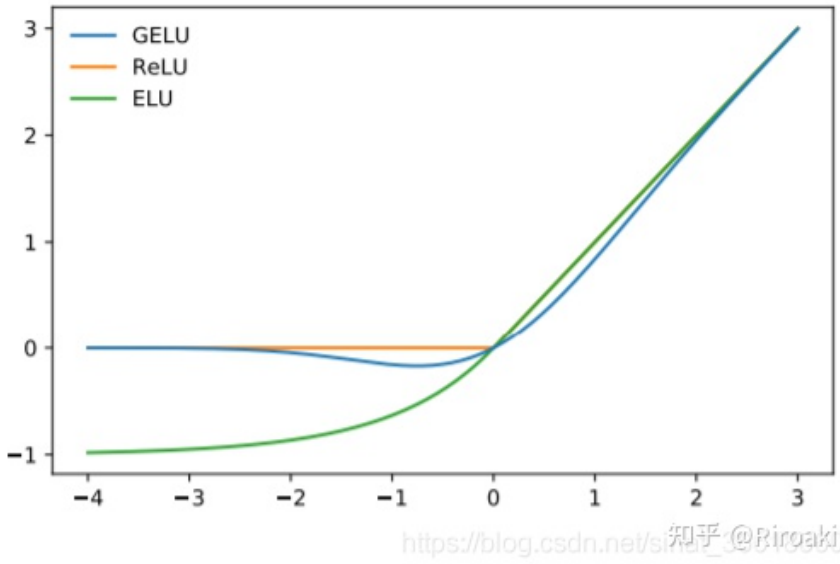
Encoder与Decoder由许多结构组成，如：多头注意力（Multi-Head Attention）层，Feed Forward层，Normaliztion层，甚至残差连接（Residual Connection，图中的“Add”）。不过，其中最重要的结构是多头注意力（Multi-Head Attention）结构，该结构基于自注意力（Self-Attention）机制，是多个Self-Attention的并行组成。

3.2.3.1 ErnieEncoderCell

这一层包装了Self-Attention、Intermediate和ErnieOutput（即Attention后的FFN部分），以及这里直接忽略的cross-attention部分（将BERT作为Decoder时涉及的部分）。

理论上，这里顺序调用三个子模块就可以，没有什么值得说明的地方。

在Attention后面还有一个全连接+激活的操作，这里的全连接做了一个扩展，将维度扩展为3072，是原始维度768的4倍之多。这里的激活函数默认实现为gelu（Gaussian Error Linerar Units(GELUS），它是无法直接计算的，可以用一个包含tanh的表达式进行近似。
作为参考（图源网络）：



至于为什么在transformer中要用这个激活函数，应该是GeLU比ReLU这些表现都好，以至于后续的语言模型都沿用了这一激活函数。

```
class ErnieEncoderCell(nn.Cell):
    """
    Encoder cells used in ErnieTransformer.
    Args:
        hidden_size (int): Size of the Ernie encoder layers. Default: 768.
        seq_length (int): Length of input sequence. Default: 512.
        num_attention_heads (int): Number of attention heads. Default: 12.
        intermediate_size (int): Size of intermediate layer. Default: 3072.
        attention_probs_dropout_prob (float): The dropout probability for
                                      ErnieAttention. Default: 0.02.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        hidden_dropout_prob (float): The dropout probability for ErnieOutput. Default: 0.1.
        use_relative_positions (bool): Specifies whether to use relative positions. Default:
False.
        hidden_act (str): Activation function. Default: "gelu".
        compute_type (:class:`mindspore.dtype`): Compute type in attention. Default:
mstype.float32.
    """
    def __init__(self,
                 hidden_size=768,
                 seq_length=512,
                 num_attention_heads=12,
                 intermediate_size=3072,
                 attention_probs_dropout_prob=0.02,
                 use_one_hot_embeddings=False,
                 initializer_range=0.02,
                 hidden_dropout_prob=0.1,
```

```
                 use_relative_positions=False,
                 hidden_act="gelu",
                 compute_type=mstype.float32):
        super(ErnieEncoderCell, self).__init__()
        self.attention = ErnieSelfAttention(
            hidden_size=hidden_size,
            seq_length=seq_length,
            num_attention_heads=num_attention_heads,
            attention_probs_dropout_prob=attention_probs_dropout_prob,
            use_one_hot_embeddings=use_one_hot_embeddings,
            initializer_range=initializer_range,
            hidden_dropout_prob=hidden_dropout_prob,
            use_relative_positions=use_relative_positions,
            compute_type=compute_type)
        self.intermediate = nn.Dense(in_channels=hidden_size,
                                     out_channels=intermediate_size,
                                     activation=hidden_act,

weight_init=TruncatedNormal(initializer_range)).to_float(compute_type)
        self.output = ErnieOutput(in_channels=intermediate_size,
                                  out_channels=hidden_size,
                                  initializer_range=initializer_range,
                                  dropout_prob=hidden_dropout_prob,
                                  compute_type=compute_type)

    def construct(self, hidden_states, attention_mask):
        # self-attention
        attention_output = self.attention(hidden_states, attention_mask)
        # feed construct
        intermediate_output = self.intermediate(attention_output)
        # add and normalize
        output = self.output(intermediate_output, attention_output)
        return output
```

3.2.3.2 Transformer

这里，我们构造ERNIE的Transformer模型如下，添加了`num_hidden_layers`（默认是12）个ERNIE Encode组件。

```
class ErnieTransformer(nn.Cell):
    """
    Multi-layer Ernie transformer.
    Args:
        hidden_size (int): Size of the encoder layers.
        seq_length (int): Length of input sequence.
        num_hidden_layers (int): Number of hidden layers in encoder cells.
        num_attention_heads (int): Number of attention heads in encoder cells. Default: 12.
        intermediate_size (int): Size of intermediate layer in encoder cells. Default: 3072.
        attention_probs_dropout_prob (float): The dropout probability for
                                      ErnieAttention. Default: 0.1.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
        initializer_range (float): Initialization value of TruncatedNormal. Default: 0.02.
        hidden_dropout_prob (float): The dropout probability for ErnieOutput. Default: 0.1.
        use_relative_positions (bool): Specifies whether to use relative positions. Default:
False.
        hidden_act (str): Activation function used in the encoder cells. Default: "gelu".
        compute_type (:class:`mindspore.dtype`): Compute type in ErnieTransformer. Default:
mstype.float32.
        return_all_encoders (bool): Specifies whether to return all encoders. Default: False.
    """
    def __init__(self,
                 hidden_size,
                 seq_length,
                 num_hidden_layers,
                 num_attention_heads=12,
                 intermediate_size=3072,
                 attention_probs_dropout_prob=0.1,
```

```
                 use_one_hot_embeddings=False,
                 initializer_range=0.02,
                 hidden_dropout_prob=0.1,
                 use_relative_positions=False,
                 hidden_act="gelu",
                 compute_type=mstype.float32,
                 return_all_encoders=False):
        super(ErnieTransformer, self).__init__()
        self.return_all_encoders = return_all_encoders

        layers = []
        for _ in range(num_hidden_layers):
            layer = ErnieEncoderCell(hidden_size=hidden_size,
                                     seq_length=seq_length,
                                     num_attention_heads=num_attention_heads,
                                     intermediate_size=intermediate_size,
                                     attention_probs_dropout_prob=attention_probs_dropout_prob,
                                     use_one_hot_embeddings=use_one_hot_embeddings,
                                     initializer_range=initializer_range,
                                     hidden_dropout_prob=hidden_dropout_prob,
                                     use_relative_positions=use_relative_positions,
                                     hidden_act=hidden_act,
                                     compute_type=compute_type)
            layers.append(layer)

        self.layers = nn.CellList(layers)

        self.reshape = P.Reshape()
        self.shape = (-1, hidden_size)
        self.out_shape = (-1, seq_length, hidden_size)

    def construct(self, input_tensor, attention_mask):
        """Multi-layer Ernie transformer."""
        prev_output = self.reshape(input_tensor, self.shape)

        all_encoder_layers = ()
        for layer_module in self.layers:
            layer_output = layer_module(prev_output, attention_mask)
            prev_output = layer_output

            if self.return_all_encoders:
                layer_output = self.reshape(layer_output, self.out_shape)
                all_encoder_layers = all_encoder_layers + (layer_output,)

        if not self.return_all_encoders:
            prev_output = self.reshape(prev_output, self.out_shape)
            all_encoder_layers = all_encoder_layers + (prev_output,)
        return all_encoder_layers
```

3.2.4 整体构建ERNIE模型

综合上述提到的`EmbeddingPostprocessor`、`ErnieTransformer`等子类，我们可以构造ERNIE模型如下：

```
class ErnieModel(nn.Cell):
    """
    Bidirectional Encoder Representations from Transformers.
    Args:
        config (Class): Configuration for ErnieModel.
        is_training (bool): True for training mode. False for eval mode.
        use_one_hot_embeddings (bool): Specifies whether to use one hot encoding form. Default:
False.
    """
    def __init__(self,
                 config,
                 is_training,
                 use_one_hot_embeddings=False):
        super(ErnieModel, self).__init__()
```

```
        config = copy.deepcopy(config)
        if not is_training:
            config.hidden_dropout_prob = 0.0
            config.attention_probs_dropout_prob = 0.0

        self.seq_length = config.seq_length
        self.hidden_size = config.hidden_size
        self.num_hidden_layers = config.num_hidden_layers
        self.embedding_size = config.hidden_size
        self.token_type_ids = None

        self.last_idx = self.num_hidden_layers - 1
        output_embedding_shape = [-1, self.seq_length, self.embedding_size]

        self.ernie_embedding_lookup = nn.Embedding(
            vocab_size=config.vocab_size,
            embedding_size=self.embedding_size,
            use_one_hot=use_one_hot_embeddings)

        self.ernie_embedding_postprocessor = EmbeddingPostprocessor(
            embedding_size=self.embedding_size,
            embedding_shape=output_embedding_shape,
            use_relative_positions=config.use_relative_positions,
            use_token_type=True,
            token_type_vocab_size=config.type_vocab_size,
            use_one_hot_embeddings=use_one_hot_embeddings,
            initializer_range=0.02,
            max_position_embeddings=config.max_position_embeddings,
            dropout_prob=config.hidden_dropout_prob)

        self.ernie_encoder = ErnieTransformer(
            hidden_size=self.hidden_size,
            seq_length=self.seq_length,
            num_attention_heads=config.num_attention_heads,
            num_hidden_layers=self.num_hidden_layers,
            intermediate_size=config.intermediate_size,
            attention_probs_dropout_prob=config.attention_probs_dropout_prob,
            use_one_hot_embeddings=use_one_hot_embeddings,
            initializer_range=config.initializer_range,
            hidden_dropout_prob=config.hidden_dropout_prob,
            use_relative_positions=config.use_relative_positions,
            hidden_act=config.hidden_act,
            compute_type=config.compute_type,
            return_all_encoders=True)

        self.cast = P.Cast()
        self.dtype = config.dtype
        self.cast_compute_type = SaturateCast(dst_type=config.compute_type)
        self.slice = P.StridedSlice()

        self.squeeze_1 = P.Squeeze(axis=1)
        self.dense = nn.Dense(self.hidden_size, self.hidden_size,
                              activation="tanh",

weight_init=TruncatedNormal(config.initializer_range)).to_float(config.compute_type)
        self._create_attention_mask_from_input_mask = CreateAttentionMaskFromInputMask(config)

    def construct(self, input_ids, token_type_ids, input_mask):
        """Bidirectional Encoder Representations from Transformers."""
        # embedding
        word_embeddings = self.ernie_embedding_lookup(input_ids)
        embedding_output = self.ernie_embedding_postprocessor(token_type_ids,
                                                              word_embeddings)

        # attention mask [batch_size, seq_length, seq_length]
        attention_mask = self._create_attention_mask_from_input_mask(input_mask)

        # ernie encoder
```

```
        encoder_output = self.ernie_encoder(self.cast_compute_type(embedding_output),
                                            attention_mask)

        sequence_output = self.cast(encoder_output[self.last_idx], self.dtype)

        # pooler
        batch_size = P.Shape()(input_ids)[0]
        sequence_slice = self.slice(sequence_output,
                                    (0, 0, 0),
                                    (batch_size, 1, self.hidden_size),
                                    (1, 1, 1))
        first_token = self.squeeze_1(sequence_slice)
        pooled_output = self.dense(first_token)
        pooled_output = self.cast(pooled_output, self.dtype)

        return sequence_output, pooled_output
```

3.2.5 构建分类评估模型

在构造好原始模型后，我们为其增加文本分类任务评估对象。该类负责分类任务评估，即XNLI（num_tlabels=3），LCQMC（num_labels=2），Chnsenti（num_tlabels=2）。返回的输出表示最终log _softmax的结果与softmax的值成正比。

```
class ErnieCLSModel(nn.Cell):
    """
    This class is responsible for classification task evaluation, i.e. XNLI(num_labels=3),
    LCQMC(num_labels=2), Chnsenti(num_labels=2). The returned output represents the final
    logits as the results of log_softmax is proportional to that of softmax.
    """
    def __init__(self, config, is_training, num_labels=2, dropout_prob=0.0,
use_one_hot_embeddings=False,
                 assessment_method=""):
        super(ErnieCLSModel, self).__init__()
        if not is_training:
            config.hidden_dropout_prob = 0.0
            config.hidden_probs_dropout_prob = 0.0
        self.ernie = ErnieModel(config, is_training, use_one_hot_embeddings)
        self.cast = P.Cast()
        self.weight_init = TruncatedNormal(config.initializer_range)
        self.log_softmax = P.LogSoftmax(axis=-1)
        self.dtype = config.dtype
        self.num_labels = num_labels
        self.dense_1 = nn.Dense(config.hidden_size, self.num_labels, weight_init=self.weight_init,
                                has_bias=True).to_float(config.compute_type)
        self.dropout = nn.Dropout(1 - dropout_prob)
        self.assessment_method = assessment_method

    def construct(self, input_ids, input_mask, token_type_id):
        _, pooled_output = \
            self.ernie(input_ids, token_type_id, input_mask)
        cls = self.cast(pooled_output, self.dtype)
        cls = self.dropout(cls)
        logits = self.dense_1(cls)
        logits = self.cast(logits, self.dtype)
        if self.assessment_method != "spearman_correlation":
            logits = self.log_softmax(logits)
        return logits
```

3.3 损失函数与优化器

3.3.1 CrossEntropyCalculation

对于ERNIE模型，其损失函数是由交叉熵损失函数构成的，为了配合后续计算，我们定义交叉熵损失计算对象`CrossEntropyCalculation`,该对象能够通过交叉熵计算模型损失。

```
class CrossEntropyCalculation(nn.Cell):
    """
    Cross Entropy loss
    """
    def __init__(self, is_training=True):
        super(CrossEntropyCalculation, self).__init__()
        self.onehot = P.OneHot()
        self.on_value = Tensor(1.0, mstype.float32)
        self.off_value = Tensor(0.0, mstype.float32)
        self.reduce_sum = P.ReduceSum()
        self.reduce_mean = P.ReduceMean()
        self.reshape = P.Reshape()
        self.last_idx = (-1,)
        self.neg = P.Neg()
        self.cast = P.Cast()
        self.is_training = is_training

    def construct(self, logits, label_ids, num_labels):
        if self.is_training:
            label_ids = self.reshape(label_ids, self.last_idx)
            one_hot_labels = self.onehot(label_ids, num_labels, self.on_value, self.off_value)
            per_example_loss = self.neg(self.reduce_sum(one_hot_labels * logits, self.last_idx))
            loss = self.reduce_mean(per_example_loss, self.last_idx)
            return_value = self.cast(loss, mstype.float32)
        else:
            return_value = logits * 1.0
        return return_value
```

3.3.2 ErnieLearningRate

这里定义学习率对象`ErnieLearningRate`来作为ERNIE模型的学习率参数。在`ErnieLearningRate`中，首先使用基于多项式衰减函数来计算学习率的`PolynomialDecayLR`,根据设定的`decay_steps`从`learing_rate`逐步变化`end_learning_rate`。
对于当前step，`PolynomialDecayLR`计算学习率的公式为：

$$decayed_learning_rate = (learning_rate - end_learning_rate) * (1 - tmp_step / tmp_decay_steps)power + end_learning_rate$$

其中，$tmp_step=min(current_step,decay_steps)$,
如果设置`update_decay_steps`为`true`，则每`decay_steps`更新`tmp_decay_step`的值。公式为：

$$tmp_decay_steps = decay_steps * ceil(current_step / decay_steps)$$

另外，由于刚开始训练时,模型的权重(weights)是随机初始化的，此时若选择一个较大的学习率,可能带来模型的不稳定(振荡)，选择Warmup预热学习率的方式，可以使得开始训练的几个epoches或者一些steps内学习率较小,在预热的小学习率下，模型可以慢慢趋于稳定,等模型相对稳定后再选择预先设置的学习率进行训练,使得模型收敛速度变得更快，模型效果更佳[引用]。我们这里使用MindSpore中的`WarmUpLR`函数来预热学习率。

```
from mindspore.nn.learning_rate_schedule import LearningRateSchedule, PolynomialDecayLR, WarmUpLR

class ErnieLearningRate(LearningRateSchedule):
    """
    Warmup-decay learning rate for Ernie network.
    """
    def __init__(self, learning_rate, end_learning_rate, warmup_steps, decay_steps, power):
        super(ErnieLearningRate, self).__init__()
        self.warmup_flag = False
        if warmup_steps > 0:
            self.warmup_flag = True
            self.warmup_lr = WarmUpLR(learning_rate, warmup_steps)
        self.decay_lr = PolynomialDecayLR(learning_rate, end_learning_rate, decay_steps, power)
        self.warmup_steps = Tensor(np.array([warmup_steps]).astype(np.float32))
```

```
        self.greater = P.Greater()
        self.one = Tensor(np.array([1.0]).astype(np.float32))
        self.cast = P.Cast()

    def construct(self, global_step):
        decay_lr = self.decay_lr(global_step)
        if self.warmup_flag:
            is_warmup = self.cast(self.greater(self.warmup_steps, global_step), mstype.float32)
            warmup_lr = self.warmup_lr(global_step)
            lr = (self.one - is_warmup) * decay_lr + is_warmup * warmup_lr
        else:
            lr = decay_lr
        return lr
```

3.3.3 训练接口

```
class ErnieCLS(nn.Cell):
    """
    Train interface for classification finetuning task.
    """
    def __init__(self, config, is_training, num_labels=2, dropout_prob=0.0,
use_one_hot_embeddings=False,
                 assessment_method=""):
        super(ErnieCLS, self).__init__()
        self.ernie = ErnieCLSModel(config, is_training, num_labels, dropout_prob,
use_one_hot_embeddings)
        self.loss = CrossEntropyCalculation(is_training)
        self.num_labels = num_labels
        self.is_training = is_training

    def construct(self, input_ids, input_mask, token_type_id, label_ids):
        logits = self.ernie(input_ids, input_mask, token_type_id)
        loss = self.loss(logits, label_ids, self.num_labels)
        return loss
```

3.4 评估指标

训练逻辑完成后，需要对模型进行评估。即使用模型的预测结果和测试集的正确标签进行对比，求出预测的准确率。对话情感分析为三分类问题，通过对获得的概率分布求得最大概率的分类标签(0、1或2)，然后判断是否与正确标签相等即可。

```
class Accuracy():
    '''
    calculate accuracy
    '''
    def __init__(self):
        self.acc_num = 0
        self.total_num = 0
    def update(self, logits, labels):
        labels = labels.asnumpy()
        labels = np.reshape(labels, -1)
        logits = logits.asnumpy()
        logit_id = np.argmax(logits, axis=-1)
        self.acc_num += np.sum(labels == logit_id)
        self.total_num += len(labels)
```

4 模型训练与保存

前序完成了模型构建和训练、评估逻辑的设计，下面进行模型训练。这里我们设置模型训练轮数为3轮。

4.1 模型保存

为了尽可能保存的模型参数，通常在训练模型的过程中，每隔一段时间就会将训练模型信息保存一次，这些模型信息包含模型的参数信息，还包含其他信息，如当前的迭代次数，优化器的参数等，以便用于快速加载模型，这个保存模型信息的时间点就叫做CheckPoint。

MindSpore提供了callback机制，可以在训练过程中执行自定义逻辑。这里使用框架提供的ModelCheckpoint、TimeMonitor和LossCallBack三个函数。 ModelCheckpoint可以保存网络模型和参数，以便进行后续的fine-tuning操作。 TimeMonitor是MindSpore官方提供的callback函数，可以用于监控训练过程中单步迭代时间。LossCallBack是我们自己定义的展示每一个step的loss的函数,代码如下。

```
from mindspore.train.callback import Callback

class LossCallBack(Callback):
    """
    Monitor the loss in training.
    If the loss in NAN or INF terminating training.
    Note:
        if per_print_times is 0 do not print loss.
    Args:
        per_print_times (int): Print loss every times. Default: 1.
    """
    def __init__(self, dataset_size=-1):
        super(LossCallBack, self).__init__()
        self._dataset_size = dataset_size
    def step_end(self, run_context):
        """
        Print loss after each step
        """
        cb_params = run_context.original_args()
        if self._dataset_size > 0:
            percent, epoch_num = math.modf(cb_params.cur_step_num / self._dataset_size)
            if percent == 0:
                percent = 1
                epoch_num -= 1
            print("epoch: {}, current epoch percent: {}, step: {}, outputs are {}"
                  .format(int(epoch_num), "%.3f" % percent, cb_params.cur_step_num,
str(cb_params.net_outputs)),
                  flush=True)
        else:
            print("epoch: {}, step: {}, outputs are {}".format(cb_params.cur_epoch_num,
cb_params.cur_step_num,
                                                               str(cb_params.net_outputs)),
flush=True)
```

接下来我们将所有训练过程中需要的callbacks封装成为一个集合，用户在模型训练的时候执行这些回调函数。

```
def get_callbacks(steps_per_epoch, save_checkpoint_path, dataset):
    ckpt_config = CheckpointConfig(save_checkpoint_steps=steps_per_epoch, keep_checkpoint_max=10)
    ckpoint_cb = ModelCheckpoint(prefix="classifier",
                                 directory=None if save_checkpoint_path == "" else
save_checkpoint_path,
                                 config=ckpt_config)
    callbacks = [TimeMonitor(dataset.get_dataset_size()),
LossCallBack(dataset.get_dataset_size()), ckpoint_cb]
    return callbacks
```

4.2 模型训练

MindSpore中将优化器应用在模型中的方法是使用`TrainOneStepWithLossScaleCell`类，该类能够使用混合精度功能的训练网络。

实现了包含损失缩放（loss scale）的单次训练。它使用网络、优化器和用于更新损失缩放系数（loss scale）的Cell(或一个Tensor)作为参数。可在host侧或device侧更新损失缩放系数。 如果需要在host侧更新，使用Tensor作为 scale_sense ，否则，使用可更新损失缩放系数的Cell实例作为 scale_sense 。

这里我们单独为ERNIE模型定义微调模块。

```
GRADIENT_CLIP_TYPE = 1
GRADIENT_CLIP_VALUE = 1.0
grad_scale = C.MultitypeFuncGraph("grad_scale")
reciprocal = P.Reciprocal()

clip_grad = C.MultitypeFuncGraph("clip_grad")


@clip_grad.register("Number", "Number", "Tensor")
def _clip_grad(clip_type, clip_value, grad):
    """
    Clip gradients.
    Inputs:
        clip_type (int): The way to clip, 0 for 'value', 1 for 'norm'.
        clip_value (float): Specifies how much to clip.
        grad (tuple[Tensor]): Gradients.
    Outputs:
        tuple[Tensor], clipped gradients.
    """
    if clip_type not in (0, 1):
        return grad
    dt = F.dtype(grad)
    if clip_type == 0:
        new_grad = C.clip_by_value(grad, F.cast(F.tuple_to_array((-clip_value,)), dt),
                                   F.cast(F.tuple_to_array((clip_value,)), dt))
    else:
        new_grad = nn.ClipByNorm()(grad, F.cast(F.tuple_to_array((clip_value,)), dt))
    return new_grad

@grad_scale.register("Tensor", "Tensor")
def tensor_grad_scale(scale, grad):
    return grad * reciprocal(scale)

_grad_overflow = C.MultitypeFuncGraph("_grad_overflow")
grad_overflow = P.FloatStatus()
@_grad_overflow.register("Tensor")
def _tensor_grad_overflow(grad):
    return grad_overflow(grad)


class ErnieFinetuneCell(nn.TrainOneStepWithLossScaleCell):
    """
    Especially defined for finetuning where only four inputs tensor are needed.
    Append an optimizer to the training network after that the construct
    function can be called to create the backward graph.
    Different from the builtin loss_scale wrapper cell, we apply grad_clip before the
optimization.
    Args:
        network (Cell): The training network. Note that loss function should have been added.
        optimizer (Optimizer): Optimizer for updating the weights.
        scale_update_cell (Cell): Cell to do the loss scale. Default: None.
    """
    def __init__(self, network, optimizer, scale_update_cell=None):
        super(ErnieFinetuneCell, self).__init__(network, optimizer, scale_update_cell)
        self.cast = P.Cast()

    def construct(self,
                  input_ids,
                  input_mask,
                  token_type_id,
                  label_ids,
                  sens=None):
        """Ernie Finetune"""
```

```
        weights = self.weights
        loss = self.network(input_ids,
                            input_mask,
                            token_type_id,
                            label_ids)
        if sens is None:
            scaling_sens = self.scale_sense
        else:
            scaling_sens = sens

        status, scaling_sens = self.start_overflow_check(loss, scaling_sens)
        grads = self.grad(self.network, weights)(input_ids,
                                                 input_mask,
                                                 token_type_id,
                                                 label_ids,
                                                 self.cast(scaling_sens,
                                                           mstype.float32))
        grads = self.hyper_map(F.partial(grad_scale, scaling_sens), grads)
        grads = self.hyper_map(F.partial(clip_grad, GRADIENT_CLIP_TYPE, GRADIENT_CLIP_VALUE),
grads)
        if self.reducer_flag:
            grads = self.grad_reducer(grads)
        cond = self.get_overflow_status(status, grads)
        overflow = self.process_loss_scale(cond)
        if not overflow:
            self.optimizer(grads)
        return (loss, cond)
```

MindSpore在构建训练过程时有多种方法，我们这里使用MindSpore的`Model`对象来配置一个模型训练过程。`Model`是模型训练或推理的高阶接口。`Model`会根据用户传入的参数封装可训练或推理的实例。

`Model`对象的实例拥有`train`、`eval`等实现方法，当我们在GPU或者Ascend上调用模型训练接口`train`时，模型流程可以通过下沉模式执行。我们只需要将需要训练的轮次`epoch`、一个训练数据迭代器`train_dataset`和训练过程中需要执行的回调对象或者回调对象列表`callbacks`传入`train`接口，即可开始模型训练。

```
from mindspore.train.serialization import load_checkpoint, load_param_into_net

def do_train(dataset=None, network=None, load_checkpoint_path="", save_checkpoint_path="",
epoch_num=1):
    """ do train """
    if load_checkpoint_path == "":
        raise ValueError("Pretrain model missed, finetune task must load pretrain model!")
    steps_per_epoch = 500
    # optimizer
    if optimizer_cfg.optimizer == 'AdamWeightDecay':
        lr_schedule = ErnieLearningRate(learning_rate=optimizer_cfg.AdamWeightDecay.learning_rate,

 end_learning_rate=optimizer_cfg.AdamWeightDecay.end_learning_rate,
                                        warmup_steps=int(steps_per_epoch * epoch_num * 0.1),
                                        decay_steps=steps_per_epoch * epoch_num,
                                        power=optimizer_cfg.AdamWeightDecay.power)
        params = network.trainable_params()
        decay_params = list(filter(optimizer_cfg.AdamWeightDecay.decay_filter, params))
        other_params = list(filter(lambda x: not optimizer_cfg.AdamWeightDecay.decay_filter(x),
params))
        group_params = [{'params': decay_params, 'weight_decay':
optimizer_cfg.AdamWeightDecay.weight_decay},
                        {'params': other_params, 'weight_decay': 0.0}]

        optimizer = AdamWeightDecay(group_params, lr_schedule,
eps=optimizer_cfg.AdamWeightDecay.eps)
    elif optimizer_cfg.optimizer == 'Adam':
        optimizer = Adam(network.trainable_params(),
learning_rate=optimizer_cfg.Adam.learning_rate)
    elif optimizer_cfg.optimizer == 'Adagrad':
```

```
        optimizer = Adagrad(network.trainable_params(),
learning_rate=optimizer_cfg.Adagrad.learning_rate)
    # load checkpoint into network
    param_dict = load_checkpoint(load_checkpoint_path)
    unloaded_params = load_param_into_net(network, param_dict)
    if len(unloaded_params) > 2:
        print(unloaded_params)
        print('Loading ernie model failed, please check the checkpoint file.')

    update_cell = DynamicLossScaleUpdateCell(loss_scale_value=2**32, scale_factor=2,
scale_window=1000)
    netwithgrads = ErnieFinetuneCell(network, optimizer=optimizer, scale_update_cell=update_cell)
    model = Model(netwithgrads)
    callbacks = get_callbacks()
    model.train(epoch_num, dataset, callbacks=callbacks)
```

4.3 开始训练

```
num_class = 3
epoch_num = 3
train_batch_size = 21

train_data_file_path = '../data/train.mindrecord'
schema_file_path = '../ms_log/train_classifier_log.txt'
train_data_shuffle = True

load_pretrain_checkpoint_path = '../pretrain_models/converted/ernie.ckpt'
save_finetune_checkpoint_path = '../save_models'

ernie_net_cfg = ErnieConfig(
    seq_length=64,
    vocab_size=18000,
    hidden_size=768,
    num_hidden_layers=12,
    num_attention_heads=12,
    intermediate_size=3072,
    hidden_act="relu",
    hidden_dropout_prob=0.1,
    attention_probs_dropout_prob=0.1,
    max_position_embeddings=513,
    type_vocab_size=2,
    initializer_range=0.02,
    use_relative_positions=False,
    dtype=mstype.float32,
    compute_type=mstype.float16,
)


netwithloss = ErnieCLS(ernie_net_cfg, True, num_labels=num_class, dropout_prob=0.1)
ds = create_classification_dataset(batch_size=train_batch_size, repeat_count=1,
                                           data_file_path=train_data_file_path,
                                           schema_file_path=schema_file_path,
                                           do_shuffle=train_data_shuffle)

do_train(ds, netwithloss, load_pretrain_checkpoint_path, save_finetune_checkpoint_path, epoch_num)

load_finetune_checkpoint_dir = make_directory(save_finetune_checkpoint_path)
load_finetune_checkpoint_path = LoadNewestCkpt(load_finetune_checkpoint_dir,
                                               ds.get_dataset_size(), epoch_num, "classifier")
```

5 模型加载与评估

使用训练过程中保存的checkpoint文件进行推理，验证模型的泛化能力。首先通过load_checkpoint接口加载模型文件，然后调用Model的eval接口对输入图片类别作出预测，再与输入图片的真实类别做比较，得出最终的预测精度值。

5.1 模型加载

首先使用 `load_checkpoint` 方法将模型文件加载到内存，然后使用 `load_param_into_net` 将参数加载到网络中，返回网络中没有被加载的参数列表。

5.2 模型评估流程

1. 使用 `load_checkpoint` 接口加载模型文件。
2. 使用 `dataset.create_dict_iterator` 根据需要评估的 `epoch` 来划分数据集批次。
3. 使用定义的原生模型按批次读入测试数据集，进行推理。
4. 使用之前定义的 `Accuracy` 不断根据模型返回结果计算精度值。
5. 计算完成，返回模型的准确率和推理时间。

```
import time
def do_eval(dataset=None, network=None, num_class=2, load_checkpoint_path=""):
    if load_checkpoint_path == "":
        raise ValueError("Finetune model missed, evaluation task must load finetune model!")
    net_for_pretraining = network(ernie_net_cfg, False, num_class)
    net_for_pretraining.set_train(False)
    param_dict = load_checkpoint(load_checkpoint_path)
    load_param_into_net(net_for_pretraining, param_dict)

    callback = Accuracy()

    evaluate_times = []
    columns_list = ["input_ids", "input_mask", "segment_ids", "label_ids"]
    for data in dataset.create_dict_iterator(num_epochs=1):
        input_data = []
        for i in columns_list:
            input_data.append(data[i])
        input_ids, input_mask, token_type_id, label_ids = input_data
        time_begin = time.time()
        logits = net_for_pretraining(input_ids, input_mask, token_type_id, label_ids)
        time_end = time.time()
        evaluate_times.append(time_end - time_begin)
        callback.update(logits, label_ids)
    print("==============================================================")
    print("acc_num {} , total_num {}, accuracy {:.6f}".format(callback.acc_num,
callback.total_num,
                                                              callback.acc_num /
callback.total_num))
    print("(w/o first and last) elapsed time: {}, per step time : {}".format(
        sum(evaluate_times[1:-1]), sum(evaluate_times[1:-1])/(len(evaluate_times) - 2)))
    print("==============================================================")
```

5.3 开始评估

```
num_class = 3
eval_batch_size = 32
eval_data_file_path = '../data/test.mindrecord'
schema_file_path = '../ms_log/eval_classifier_log.txt'
eval_data_shuffle = 'false'

ds = create_classification_dataset(batch_size=eval_batch_size, repeat_count=1,
                                   data_file_path=eval_data_file_path,
                                   schema_file_path=schema_file_path,
                                   do_shuffle=(eval_data_shuffle.lower() == "true"),
                                   drop_remainder=False)
do_eval(ds, ErnieCLS, num_class, load_finetune_checkpoint_path)
```

6 模型测试

最后我们设计一个预测函数，实现开头描述的效果，输入一句对话，获得对话的情绪分类。具体包含以下步骤:

1. 将输入句子进行分词;

2. 使用词表获取对应的index id序列；
3. index id序列转为Tensor；
4. 送入模型获得预测结果；
5. 反馈输出预测结果。

```
```

7. 总结

本案例完成了ERNIE模型在百度公开的机器人聊天数据上进行训练，验证和推理的过程。其中，对关键的ERNIE模型结构和原理作了讲解。通过学习本案例，理解源码可以帮助用户掌握Multi-Head Attention，TransformerEncoder，pos_embedding等关键概念，如果要详细理解ERNIE的模型原理，建议基于源码更深层次的详细阅读，可以参考MindSpore实现的Model Zoo中的EmoTect项目:Gitee:EmoTect

8. 引用

[1] Sun Y, Wang S, Li Y, et al. Ernie: Enhanced representation through knowledge integration[J]. arXiv preprint arXiv:1904.09223, 2019.
[2] Vaswani A，Shazeer N，Parmar N , et al. Attention Is All You Need[C]// arXiv. arXiv, 2017.