# 快速入门：GNMT v2的MindSpore实现

## 1. 项目概述

### 1.1 机器翻译

### 1.2 GNMT v2模型

GNMT v2 模型类似于Google 的 Neural Machine Translation System: Bridging the Gap between Human and Machine Translation中描述的模型，主要用于语料库翻译

GNMTv2 模型主要由编码器、解码器和注意力机制组成，其中编码器和解码器使用共享的词嵌入向量。编码器：由四个长短期记忆（LSTM）层组成。第一个 LSTM 层是双向的，而其他三层是单向的。解码器：由四个单向 LSTM 层和一个全连接分类器组成。LSTM 的输出嵌入维度为 1024。注意机制：使用标准化的 Bahdanau 注意机制。首先，解码器的第一层输出作为注意力机制的输入。然后，注意力机制的计算结果连接到解码器LSTM的输入，作为后续LSTM层的输入
这个模型由一个有 8 个编码器和 8 个解码器层的深度 LSTM 网络组成，使用残差连接以及从解码器网络到编码器的注意力模块连接，主要特点有：

- 为了提高并行度，从而减少训练时间，该模型使用注意机制将解码器的底层连接到编码器的顶层；
- 为了提高最终的翻译速度，在推理计算中采用了低精度的算法（限制数值范围）；
- 为了改进对罕见单词的处理，作者将单词分为有限的公共子单词单元（称之为“单词块”），用于输入和输出，该方法在“字符”分隔模型的灵活性和“单词”分隔模型的效率之间提供了良好的平衡，自然地处理了罕见单词的翻译，最终提高了系统的整体准确性；
- 通过一种波束搜索技术（beam search technique）使用了长度标准化过程和覆盖惩罚，这鼓励生成输出语句，最有可能覆盖源语句中的所有单词；
- 为了直接优化翻译任务的 BLEU 分数，作者还使用了强化学习来细化模型；

在 WMT14 的英法和英德基准测试中，GNMT 达到了最先进水平的竞争性结果，与谷歌的基于短语的生产系统相比，通过对一组孤立的简单句子进行人工并排评估，它平均减少了60%的翻译错误；

### 1.3 环境要求

- 硬件
    1. 带GPU显卡或华为Ascend处理器的服务器
- 软件
    1. python3.5+
    2. `mindspore`
    3. `numpy`
    4. `sacrebleu==1.4.14`
    5. `sacremoses==0.0.35`
    6. `subword_nmt==0.3.7`

## 2. 数据准备

## 2.1 数据下载

本项目使用WMT英语-德语数据集。使用一下脚本下载并转换数据到指定格式。

```
!wget
https://github.com/NVIDIA/DeepLearningExamples/blob/master/TensorFlow/Translatio
n/GNMT/scripts/wmt16_en_de.sh
!bash ./scripts/wmt16_en_de.sh
```

## 2.2 MindRecord数据转换

### 2.2.1 Tokenizer

定义`Tokenizer`分词器，这里`Tokenizer`分词器首先构建词表(Vocab list)，得到两个token和id，id和token相互对应的字典，然后利用`sacremoses`包分别实现`tokenize()`方法和`detokenize()`方法，也就是将英语和德语相互转换。

对于`tokenize()`方法，有个很重要的点就是BPE(Byte Pair Encoding)算法。

BPE首次在论文Neural Machine Translation of Rare Words with Subword Units(Sennrich et al., 2015)中被提出。BPE首先需要依赖一个可以预先将训练数据切分成单词的tokenizer，它们可以一些简单的基于空格的tokenizer，如GPT-2，Roberta等；也可以是一些更加复杂的、增加了一些规则的tokenizer，如XLM、FlauBERT。

在使用了这些tokenizer后，我们可以得到一个在训练数据中出现过的单词的集合以及它们对应的频数。下一步，BPE使用这个集合中的所有符号（将单词拆分为字母）创建一个基本词表，然后学习合并规则以将基本词表的两个符号形成一个新符号，从而实现对基本词表的更新。它将持续这一操作，直到词表的大小达到了预置的规模。值得注意的是，这个预置的词表大小是一个超参数，需要提前指定。

举个例子，假设经过预先切分后，单词及对应的频数如下：

`("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)`

因此，基本词表的内容为[“b”, “g”, “h”, “n”, “p”, “s”, “u”]。对应的，将所有的单词按照基本词表中的字母拆分，得到：

`("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)`

接下来，BPE计算任意两个字母（符号）拼接到一起时，出现在语料中的频数，然后选择频数最大的字母（符号）对。接上例，"hu"组合的频数为15（"hug"出现了10次，“hugs"中出现了5次）。在上面的例子中，频数最高的符号对是"ug”，一共有20次。因此，tokenizer学习到的第一个合并规则就是将所有的"ug"合并到一起。于是，基本词表变为：

`("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)`

应用相同的算法，下一个频数最高的组合是"un"，出现了16次，于是"un"被添加到词表中；接下来是"hug"，即"h"与第一步得到的"ug"组合的频数最高，共有15次，于是"hug"被添加到了词表中。

此时，词表的内容为[“b”, “g”, “h”, “n”, “p”, “s”, “u”, “ug”, “un”, “hug”]，原始的单词按照词表拆分后的内容如下：
`("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)`

假定BPE的训练到这一步就停止，接下来就是利用它学习到的这些规则来切分新的单词（只要新单词中没有超出基本词表之外的符号）。例如，单词"bug"将会被切分为[“b”, “ug”]，但是单词"mug"将会被切分为["", “ug”]——这是因为"m"不在词表中。

正如之前提到的，词表的规模——也就是基本词表的大小加上合并后的单词的数量——是一个超参数。例如，对于GPT而言，其词表的大小为40,478，其中，基本字符一共有478个，合并后的词有40,000个。.

```
import os
from collections import defaultdict
from functools import partial
import subword_nmt.apply_bpe
import sacremoses

class Tokenizer:
    """
    Tokenizer class.
    """

    def __init__(self, vocab_address=None, bpe_code_address=None,
                 src_en='en', tgt_de='de', vocab_pad=8, isolator='@@'):
        """
        Constructor for the Tokenizer class.

        Args:
            vocab_address: vocabulary address.
            bpe_code_address: path to the file with bpe codes.
            vocab_pad: pads vocabulary to a multiple of 'vocab_pad' tokens.
            isolator: tokenization isolator.
        """
        self.padding_index = 0
        self.unk_index = 1
        self.bos_index = 2
        self.eos_index = 3
        self.pad_word = '<pad>'
        self.unk_word = '<unk>'
        self.bos_word = '<s>'
        self.eos_word = r'<\s>'
        self.isolator = isolator
        self.init_bpe(bpe_code_address)
        self.vocab_establist(vocab_address, vocab_pad)
        self.sacremoses_tokenizer = sacremoses.MosesTokenizer(src_en)
        self.sacremoses_detokenizer = sacremoses.MosesDetokenizer(tgt_de)

    def init_bpe(self, bpe_code_address):
        """Init bpe."""
        if (bpe_code_address is not None) and os.path.exists(bpe_code_address):
            with open(bpe_code_address, 'r') as f1:
                self.bpe = subword_nmt.apply_bpe.BPE(f1)

    def vocab_establist(self, vocab_address, vocab_pad):
        """Establish vocabulary."""
        if (vocab_address is None) or (not os.path.exists(vocab_address)):
            return
        vocab_words = [self.pad_word, self.unk_word, self.bos_word,
self.eos_word]
        with open(vocab_address) as f1:
            for sentence in f1:
```

```
                vocab_words.append(sentence.strip())
        vocab_size = len(vocab_words)
        padded_vocab_size = (vocab_size + vocab_pad - 1) // vocab_pad *
vocab_pad
        for idx in range(0, padded_vocab_size - vocab_size):
            fil_token = f'filled{idx:04d}'
            vocab_words.append(fil_token)
        self.vocab_size = len(vocab_words)
        self.tok2idx = defaultdict(partial(int, self.unk_index))
        for idx, token in enumerate(vocab_words):
            self.tok2idx[token] = idx
        self.idx2tok = {}
        self.idx2tok = defaultdict(partial(str, ","))
        for token, idx in self.tok2idx.items():
            self.idx2tok[idx] = token

    def tokenize(self, sentence):
        """Tokenize sentence."""
        tokenized = self.sacremoses_tokenizer.tokenize(sentence,
return_str=True)
        bpe = self.bpe.process_line(tokenized)
        sentence = bpe.strip().split()
        inputs = [self.tok2idx[i] for i in sentence]
        inputs = [self.bos_index] + inputs + [self.eos_index]
        return inputs

    def detokenize(self, indexes, gap=' '):
        """Detokenizes single sentence and removes token isolator characters."""
        reconstruction_bpe = gap.join([self.idx2tok[idx] for idx in indexes])
        reconstruction_bpe = reconstruction_bpe.replace(self.isolator + ' ', '')
        reconstruction_bpe = reconstruction_bpe.replace(self.isolator, '')
        reconstruction_bpe = reconstruction_bpe.replace(self.bos_word, '')
        reconstruction_bpe = reconstruction_bpe.replace(self.eos_word, '')
        reconstruction_bpe = reconstruction_bpe.replace(self.unk_word, '')
        reconstruction_bpe = reconstruction_bpe.replace(self.pad_word, '')
        reconstruction_bpe = reconstruction_bpe.strip()
        reconstruction_words =
self.sacremoses_detokenizer.detokenize(reconstruction_bpe.split())
        return reconstruction_words
```

### 2.2.2 DataLoader

定义`DataLoader`类，该类有`padding()`方法，当句子长度不够时，可以为句子添加`<pad>`特殊填充符号。实现了`write_to_mindrecord()`方法，该方法可以将给定Schema类型的数据写入到`MindRecord`格式的数据集中。

```
SCHEMA = {
    "src": {"type": "int64", "shape": [-1]},
    "src_padding": {"type": "int64", "shape": [-1]},
    "prev_opt": {"type": "int64", "shape": [-1]},
    "target": {"type": "int64", "shape": [-1]},
    "tgt_padding": {"type": "int64", "shape": [-1]},
}

TEST_SCHEMA = {
    "src": {"type": "int64", "shape": [-1]},
    "src_padding": {"type": "int64", "shape": [-1]},
```

```
}

class DataLoader:
    """Data loader for dataset."""
    _SCHEMA = SCHEMA
    _TEST_SCHEMA = TEST_SCHEMA

    def __init__(self):
        self._examples = []

    def _load(self):
        raise NotImplementedError

    def padding(self, sen, padding_idx, need_sentence_len=None, dtype=np.int64):
        """Padding <pad> to sentence."""
        if need_sentence_len is None:
            return None
        if sen.shape[0] > need_sentence_len:
            return None
        new_sen = np.array([padding_idx] * need_sentence_len, dtype=dtype)
        new_sen[:sen.shape[0]] = sen[:]
        return new_sen

    def write_to_mindrecord(self, path, train_mode, shard_num=1, desc="gnmt"):
        """
        Write mindrecord file.

        Args:
            path (str): File path.
            shard_num (int): Shard num.
            desc (str): Description.
        """
        if not os.path.isabs(path):
            path = os.path.abspath(path)

        writer = FileWriter(file_name=path, shard_num=shard_num)
        if train_mode:
            writer.add_schema(self._SCHEMA, desc)
        else:
            writer.add_schema(self._TEST_SCHEMA, desc)
        if not self._examples:
            self._load()

        writer.write_raw_data(self._examples)
        writer.commit()
        print(f"| Wrote to {path}.")

    def _add_example(self, example):
        self._examples.append(example)
```

### 2.2.3 TextDataLoader

定义`TextDataLoader`类来加载测试预料，这里需要对数据语料简单处理，首先将测试语料通过`Tokenizer`分词器将器转换成向量，同时为每一句向量添加开始符(bos_index)和结束符(eod_index)，同时去掉长度太长的句子得到输入和输出双语向量，完成数据加载，最后在根据各列类型制作类型schema的Json文件。

```
class TextDataLoader(DataLoader):
    """Loader for text data."""

    def __init__(self,
                 src_filepath: str,
                 tokenizer: Tokenizer,
                 min_sen_len=0,
                 source_max_sen_len=None,
                 schema_address=None):
        super(TextDataLoader, self).__init__()
        self._src_filepath = src_filepath
        self.tokenizer = tokenizer
        self.min_sen_len = min_sen_len
        self.source_max_sen_len = source_max_sen_len
        self.schema_address = schema_address

    def _load(self):
        count = 0
        if self.source_max_sen_len is None:
            with open(self._src_filepath, "r") as _src_file:
                print(f" | count the max_sen_len of corpus
{self._src_filepath}.")
                max_src = 0
                for _, _pair in enumerate(_src_file):
                    src_tokens = self.tokenizer.tokenize(_pair)
                    src_len = len(src_tokens)
                    if src_len > max_src:
                        max_src = src_len
                self.source_max_sen_len = max_src

        with open(self._src_filepath, "r") as _src_file:
            print(f" | Processing corpus {self._src_filepath}.")
            for _, _pair in enumerate(_src_file):
                src_tokens = self.tokenizer.tokenize(_pair)
                src_len = len(src_tokens)
                src_tokens = np.array(src_tokens)
                # encoder inputs
                encoder_input = self.padding(src_tokens,
self.tokenizer.padding_index, self.source_max_sen_len)
                src_padding = np.zeros(shape=self.source_max_sen_len,
dtype=np.int64)
                for i in range(src_len):
                    src_padding[i] = 1

                example = {
                    "src": encoder_input,
                    "src_padding": src_padding
                }
                self._add_example(example)
                count += 1

            print(f" | source padding_len = {self.source_max_sen_len}.")
            print(f" | Total  activate  sen = {count}.")
            print(f" | Total  sen = {count}.")

            if self.schema_address is not None:
```

```
                provlist = [count, self.source_max_sen_len,
self.source_max_sen_len]
                columns = ["src", "src_padding"]
                with open(self.schema_address, "w", encoding="utf-8") as  f:
                    f.write("{\n")
                    f.write('  "datasetType":"MS",\n')
                    f.write('  "numRows":%s,\n' % provlist[0])
                    f.write('  "columns":{\n')
                    t = 1
                    for name in columns:
                        f.write('    "%s":{\n' % name)
                        f.write('      "type":"int64",\n')
                        f.write('      "rank":1,\n')
                        f.write('      "shape":[%s]\n' % provlist[t])
                        f.write('    }')
                        if t < len(columns):
                            f.write(',')
                        f.write('\n')
                        t += 1
                    f.write('  }\n}\n')
                    print(" | Write to " + self.schema_address)
```

### 2.2.4 BiLingualDataLoader

定义`BiLingualDataLoader`双语数据加载器，这里需要对数据预料简单处理，首先将双语语料通过`Tokenizer`分词器将器转换成向量，同时为每一句向量添加开始符(bos_index)和结束符(eod_index)，同时去掉长度太长的句子得到输入和输出双语向量，完成数据加载，最后在根据各列类型制作类型schema的Json文件。

```
class BiLingualDataLoader(DataLoader):
    """Loader for bilingual data."""

    def __init__(self,
                 src_filepath: str,
                 tgt_filepath: str,
                 tokenizer: Tokenizer,
                 min_sen_len=0,
                 source_max_sen_len=None,
                 target_max_sen_len=80,
                 schema_address=None):
        super(BiLingualDataLoader, self).__init__()
        self._src_filepath = src_filepath
        self._tgt_filepath = tgt_filepath
        self.tokenizer = tokenizer
        self.min_sen_len = min_sen_len
        self.source_max_sen_len = source_max_sen_len
        self.target_max_sen_len = target_max_sen_len
        self.schema_address = schema_address

    def _load(self):
        count = 0
        if self.source_max_sen_len is None:
            with open(self._src_filepath, "r") as _src_file:
                print(f" | count the max_sen_len of corpus
{self._src_filepath}.")
                max_src = 0
```

```
                for _, _pair in enumerate(_src_file):
                    src_tokens = [
                        int(self.tokenizer.tok2idx[t])
                        for t in _pair.strip().split(" ") if t
                    ]
                    src_len = len(src_tokens)
                    if src_len > max_src:
                        max_src = src_len
                self.source_max_sen_len = max_src + 2

        if self.target_max_sen_len is None:
            with open(self._src_filepath, "r") as _tgt_file:
                print(f" | count the max_sen_len of corpus
{self._src_filepath}.")
                max_tgt = 0
                for _, _pair in enumerate(_tgt_file):
                    src_tokens = [
                        int(self.tokenizer.tok2idx[t])
                        for t in _pair.strip().split(" ") if t
                    ]
                    tgt_len = len(src_tokens)
                    if tgt_len > max_tgt:
                        max_tgt = tgt_len
                self.target_max_sen_len = max_tgt + 1

        with open(self._src_filepath, "r") as _src_file:
            print(f" | Processing corpus {self._src_filepath}.")
            print(f" | Processing corpus {self._tgt_filepath}.")
            with open(self._tgt_filepath, "r") as _tgt_file:
                for _, _pair in enumerate(zip(_src_file, _tgt_file)):

                    src_tokens = [
                        int(self.tokenizer.tok2idx[t])
                        for t in _pair[0].strip().split(" ") if t
                    ]
                    tgt_tokens = [
                        int(self.tokenizer.tok2idx[t])
                        for t in _pair[1].strip().split(" ") if t
                    ]
                    src_tokens.insert(0, self.tokenizer.bos_index)
                    src_tokens.append(self.tokenizer.eos_index)
                    tgt_tokens.insert(0, self.tokenizer.bos_index)
                    tgt_tokens.append(self.tokenizer.eos_index)
                    src_tokens = np.array(src_tokens)
                    tgt_tokens = np.array(tgt_tokens)
                    src_len = src_tokens.shape[0]
                    tgt_len = tgt_tokens.shape[0]

                    if (src_len > self.source_max_sen_len) or (src_len <
self.min_sen_len) or (
                            tgt_len > (self.target_max_sen_len + 1)) or (tgt_len
< self.min_sen_len):
                        print(f"+++++ delete! src_len={src_len}, tgt_len=
{tgt_len - 1}, "
                              f"source_max_sen_len={self.source_max_sen_len},"
                              f"target_max_sen_len={self.target_max_sen_len}")
                        continue
                    # encoder inputs
```

```
                    encoder_input = self.padding(src_tokens,
self.tokenizer.padding_index, self.source_max_sen_len)
                    src_padding = np.zeros(shape=self.source_max_sen_len,
dtype=np.int64)
                    for i in range(src_len):
                        src_padding[i] = 1
                    # decoder inputs
                    decoder_input = self.padding(tgt_tokens[:-1],
self.tokenizer.padding_index, self.target_max_sen_len)
                    # decoder outputs
                    decoder_output = self.padding(tgt_tokens[1:],
self.tokenizer.padding_index, self.target_max_sen_len)
                    tgt_padding = np.zeros(shape=self.target_max_sen_len + 1,
dtype=np.int64)
                    for j in range(tgt_len):
                        tgt_padding[j] = 1
                    tgt_padding = tgt_padding[1:]
                    decoder_input = np.array(decoder_input, dtype=np.int64)
                    decoder_output = np.array(decoder_output, dtype=np.int64)
                    tgt_padding = np.array(tgt_padding, dtype=np.int64)

                    example = {
                        "src": encoder_input,
                        "src_padding": src_padding,
                        "prev_opt": decoder_input,
                        "target": decoder_output,
                        "tgt_padding": tgt_padding
                    }
                    self._add_example(example)
                    count += 1

                print(f" | source padding_len = {self.source_max_sen_len}.")
                print(f" | target padding_len = {self.target_max_sen_len}.")
                print(f" | Total  activate  sen = {count}.")
                print(f" | Total  sen = {count}.")

                if self.schema_address is not None:
                    provlist = [count, self.source_max_sen_len,
self.source_max_sen_len,
                                self.target_max_sen_len,
self.target_max_sen_len, self.target_max_sen_len]
                    columns = ["src", "src_padding", "prev_opt", "target",
"tgt_padding"]
                    with open(self.schema_address, "w", encoding="utf-8") as  f:
                        f.write("{\n")
                        f.write('  "datasetType":"MS",\n')
                        f.write('  "numRows":%s,\n' % provlist[0])
                        f.write('  "columns":{\n')
                        t = 1
                        for name in columns:
                            f.write('    "%s":{\n' % name)
                            f.write('      "type":"int64",\n')
                            f.write('      "rank":1,\n')
                            f.write('      "shape":[%s]\n' % provlist[t])
                            f.write('    }')
                            if t < len(columns):
                                f.write(',')
                            f.write('\n')
```

```
                t += 1
            f.write('  }\n}\n')
            print(" | Write to " + self.schema_address)
```

### 2.2.5 数据集转换

定义`create_dataset()`方法将原有文本转换成`MindRecord`数据集格式。首先实例化上述定义的`Tokenizer`分词器，

```
def create_dataset():
    dicts = []
    train_src_file = "train.tok.clean.bpe.32000.en"
    train_tgt_file = "train.tok.clean.bpe.32000.de"
    test_src_file = "newstest2014.en"
    test_tgt_file = "newstest2014.de"

    vocab = args.src_folder + "/vocab.bpe.32000"
    bpe_codes = args.src_folder + "/bpe.32000"
    pad_vocab = 8
    tokenizer = Tokenizer(vocab, bpe_codes, src_en='en', tgt_de='de',
vocab_pad=pad_vocab)

    test = TextDataLoader(
        src_filepath=os.path.join(args.src_folder, test_src_file),
        tokenizer=tokenizer,
        source_max_sen_len=None,
        schema_address=args.output_folder + "/" + test_src_file + ".json"
    )
    print(f" | It's writing, please wait a moment.")
    test.write_to_mindrecord(
        path=os.path.join(
            args.output_folder,
            os.path.basename(test_src_file) + ".mindrecord"
        ),
        train_mode=False
    )

    train = BiLingualDataLoader(
        src_filepath=os.path.join(args.src_folder, train_src_file),
        tgt_filepath=os.path.join(args.src_folder, train_tgt_file),
        tokenizer=tokenizer,
        source_max_sen_len=51,
        target_max_sen_len=50,
        schema_address=args.output_folder + "/" + train_src_file + ".json"
    )
    print(f" | It's writing, please wait a moment.")
    train.write_to_mindrecord(
        path=os.path.join(
            args.output_folder,
            os.path.basename(train_src_file) + ".mindrecord"
        ),
        train_mode=True
    )

    print(f" | Vocabulary size: {tokenizer.vocab_size}.")
```

## 2.3 数据加载

定义数据加载方法，使用`mindspore.dataset`中的`MindDataset`读取之前转换的`MindRecord`文件，根据列名读取文件后转换为划分`batch`的`dataset`数据集类实例

```
import mindspore.common.dtype as mstype
import mindspore.dataset as ds
import mindspore.dataset.transforms as deC

def _load_dataset(input_files, batch_size, sink_mode=False,
                  rank_size=1, rank_id=0, shuffle=True, drop_remainder=True,
                  is_translate=False):
    """
    Load dataset according to passed in params.

    Args:
        input_files (list): Data files.
        batch_size (int): Batch size.
        sink_mode (bool): Whether enable sink mode.
        rank_size (int): Rank size.
        rank_id (int): Rank id.
        shuffle (bool): Whether shuffle dataset.
        drop_remainder (bool): Whether drop the last possibly incomplete batch.
        is_translate (bool): Whether translate the text.

    Returns:
        Dataset, dataset instance.
    """
    if not input_files:
        raise FileNotFoundError("Require at least one dataset.")

    if not isinstance(sink_mode, bool):
        raise ValueError("`sink` must be type of bool.")

    for datafile in input_files:
        print(f" | Loading {datafile}.")

    if not is_translate:
        data_set = ds.MindDataset(
            input_files, columns_list=[
                "src", "src_padding",
                "prev_opt",
                "target", "tgt_padding"
            ], shuffle=False, num_shards=rank_size, shard_id=rank_id,
            num_parallel_workers=8
        )

        ori_dataset_size = data_set.get_dataset_size()
        print(f" | Dataset size: {ori_dataset_size}.")
        if shuffle:
            data_set = data_set.shuffle(buffer_size=ori_dataset_size // 20)
        type_cast_op = deC.TypeCast(mstype.int32)
        data_set = data_set.map(input_columns="src", operations=type_cast_op,
num_parallel_workers=8)
        data_set = data_set.map(input_columns="src_padding",
operations=type_cast_op, num_parallel_workers=8)
```

```
        data_set = data_set.map(input_columns="prev_opt",
operations=type_cast_op, num_parallel_workers=8)
        data_set = data_set.map(input_columns="target", operations=type_cast_op,
num_parallel_workers=8)
        data_set = data_set.map(input_columns="tgt_padding",
operations=type_cast_op, num_parallel_workers=8)

        data_set = data_set.rename(
            input_columns=["src",
                           "src_padding",
                           "prev_opt",
                           "target",
                           "tgt_padding"],
            output_columns=["source_eos_ids",
                            "source_eos_mask",
                            "target_sos_ids",
                            "target_eos_ids",
                            "target_eos_mask"]
        )
        data_set = data_set.batch(batch_size, drop_remainder=drop_remainder)
    else:
        data_set = ds.MindDataset(
            input_files, columns_list=[
                "src", "src_padding"
            ],
            shuffle=False, num_shards=rank_size, shard_id=rank_id,
            num_parallel_workers=8
        )

        ori_dataset_size = data_set.get_dataset_size()
        print(f" | Dataset size: {ori_dataset_size}.")
        if shuffle:
            data_set = data_set.shuffle(buffer_size=ori_dataset_size // 20)
        type_cast_op = deC.TypeCast(mstype.int32)
        data_set = data_set.map(input_columns="src", operations=type_cast_op,
num_parallel_workers=8)
        data_set = data_set.map(input_columns="src_padding",
operations=type_cast_op, num_parallel_workers=8)

        data_set = data_set.rename(
            input_columns=["src",
                           "src_padding"],
            output_columns=["source_eos_ids",
                            "source_eos_mask"]
        )
        data_set = data_set.batch(batch_size, drop_remainder=drop_remainder)

    return data_set


def load_dataset(data_files: list, batch_size: int, sink_mode: bool,
                 rank_size: int = 1, rank_id: int = 0, shuffle=True,
drop_remainder=True, is_translate=False):
    """
    Load dataset.

    Args:
        data_files (list): Data files.
```

```
        batch_size (int): Batch size.
        sink_mode (bool): Whether enable sink mode.
        rank_size (int): Rank size.
        rank_id (int): Rank id.
        shuffle (bool): Whether shuffle dataset.

    Returns:
        Dataset, dataset instance.
    """
    return _load_dataset(data_files, batch_size, sink_mode, rank_size, rank_id,
shuffle=shuffle,
                         drop_remainder=drop_remainder,
is_translate=is_translate)
```

## 2.4 config文件加载配置

配置训练参数。定义`get_config()`函数统一配置如训练平台、配置参数和优化器参数等参数设置。

```
def _is_dataset_file(file: str):
    return "tfrecord" in file.lower() or "mindrecord" in file.lower()

def _get_files_from_dir(folder: str):
    _files = []
    for file in os.listdir(folder):
        if _is_dataset_file(file):
            _files.append(os.path.join(folder, file))
    return _files

def get_source_list(folder: str) -> List:
    """
    Get file list from a folder.

    Returns:
        list, file list.
    """
    _list = []
    if not folder:
        return _list

    if os.path.isdir(folder):
        _list = _get_files_from_dir(folder)
    else:
        if _is_dataset_file(folder):
            _list.append(folder)
    return _list

def get_config(config):
    '''get config.'''
    config.pre_train_dataset = None if config.pre_train_dataset == "" else
config.pre_train_dataset
    config.fine_tune_dataset = None if config.fine_tune_dataset == "" else
config.fine_tune_dataset
    config.valid_dataset = None if config.valid_dataset == "" else
config.valid_dataset
    config.test_dataset = None if config.test_dataset == "" else
config.test_dataset
    if hasattr(config, 'test_tgt'):
```

```
        config.test_tgt = None if config.test_tgt == "" else config.test_tgt

    config.pre_train_dataset = get_source_list(config.pre_train_dataset)
    config.fine_tune_dataset = get_source_list(config.fine_tune_dataset)
    config.valid_dataset = get_source_list(config.valid_dataset)
    config.test_dataset = get_source_list(config.test_dataset)

    if not isinstance(config.epochs, int) and config.epochs < 0:
        raise ValueError("`epoch` must be type of int.")

    config.compute_type = mstype.float16
    config.dtype = mstype.float32
    return config
```

# 3.模型构建

## 3.1 Embedding层

定义`EmbeddingLookup`类，该类能够产生词嵌入表。

在这里有一个关键参数是`use_one_hot_embeddings`。当use_one_hot_embeddings参数设置为True时，则会在生成embedding时先生成一个对应的onehot张量，并用onehot 张量与embedding table相乘最终获得对应的embedding张量。而当use_one_hot_embeddings参数设置为False时，则会直接利用tf.gather()方法在embedding table中将对应的embedding提取出来，组合成为对应的embedding张量。

```
import numpy as np

import mindspore.common.dtype as mstype
from mindspore import nn
from mindspore.ops import operations as P
from mindspore.common.tensor import Tensor
from mindspore.common.parameter import Parameter

class EmbeddingLookup(nn.Cell):
    """
    Embeddings lookup table with a fixed dictionary and size.

    Args:
        is_training (bool): Whether to train.
        vocab_size (int): Size of the dictionary of embeddings.
        embed_dim (int): The size of word embedding.
        initializer_range (int): The initialize range of parameters.
        use_one_hot_embeddings (bool): Whether use one-hot embedding. Default:
False.
    """

    def __init__(self,
                 is_training,
                 vocab_size,
                 embed_dim,
                 initializer_range=0.1,
                 use_one_hot_embeddings=False):

        super(EmbeddingLookup, self).__init__()
```

```
        self.is_training = is_training
        self.embedding_dim = embed_dim
        self.vocab_size = vocab_size
        self.use_one_hot_embeddings = use_one_hot_embeddings

        init_weight = np.random.normal(-initializer_range, initializer_range,
size=[vocab_size, embed_dim])
        self.embedding_table = Parameter(Tensor(init_weight, mstype.float32))
        self.expand = P.ExpandDims()
        self.gather = P.Gather()
        self.one_hot = P.OneHot()
        self.on_value = Tensor(1.0, mstype.float32)
        self.off_value = Tensor(0.0, mstype.float32)
        self.array_mul = P.MatMul()
        self.reshape = P.Reshape()
        self.get_shape = P.Shape()
        self.cast = P.Cast()

    def construct(self, input_ids):
        """
        Construct network.

        Args:
            input_ids (Tensor): A batch of sentences with shape (N, T).

        Returns:
            Tensor, word embeddings with shape (N, T, D)
        """
        _shape = self.get_shape(input_ids)  # (N, T).
        _batch_size = _shape[0]
        _max_len = _shape[1]
        if self.is_training:
            embedding_table = self.cast(self.embedding_table, mstype.float16)
        else:
            embedding_table = self.embedding_table

        flat_ids = self.reshape(input_ids, (_batch_size * _max_len,))
        if self.use_one_hot_embeddings:
            one_hot_ids = self.one_hot(flat_ids, self.vocab_size, self.on_value,
self.off_value)
            if self.is_training:
                one_hot_ids = self.cast(one_hot_ids, mstype.float16)
            output_for_reshape = self.array_mul(one_hot_ids, embedding_table)
        else:
            output_for_reshape = self.gather(embedding_table, flat_ids, 0)

        output = self.reshape(output_for_reshape, (_batch_size, _max_len,
self.embedding_dim))
        if self.is_training:
            output = self.cast(output, mstype.float32)
            embedding_table = self.cast(embedding_table, mstype.float32)
        return output, embedding_table
```

## 3.2 Encoder

### 3.2.1 DynamicRNNCell

定义`DynamicRNNCell`类，该类中首先定义了$w$权重参数和$b$偏置参数，随后根据当前运行平台的不同，初始化不同的循环神经网络，在华为的`ascend`平台，使用的是`mindspore.ops`中的`DynamicRNN`；在其他平台上，使用的是传统的`mindspore.nn`中的`LSTM`。同时需要对不同平台的数据格式进行转换，所以也使用了`ops`中的`Cast`类，能够将数据根据不同平台转换为不同计算类型。

```
from mindspore import context

class DynamicRNNCell(nn.Cell):
    """
    DynamicRNN Cell.

    Args:
        num_setp (int): Lengths of setences.
        batch_size (int): Batch size.
        word_embed_dim (int): Input size.
        hidden_size (int): Hidden size .
        initializer_range (float): Initial range. Default: 0.02
    """

    def __init__(self,
                 num_setp=50,
                 batch_size=128,
                 word_embed_dim=1024,
                 hidden_size=1024,
                 initializer_range=0.1):
        super(DynamicRNNCell, self).__init__()
        self.num_step = num_setp
        self.batch_size = batch_size
        self.input_size = word_embed_dim
        self.hidden_size = hidden_size
        # w
        dynamicRNN_w = np.random.uniform(-initializer_range, initializer_range,
                                         size=[self.input_size +
self.hidden_size, 4 * self.hidden_size])
        self.dynamicRNN_w = Parameter(Tensor(dynamicRNN_w, mstype.float32))
        # b
        dynamicRNN_b = np.random.uniform(-initializer_range, initializer_range,
size=[4 * self.hidden_size])
        self.dynamicRNN_b = Parameter(Tensor(dynamicRNN_b, mstype.float32))

        self.dynamicRNN_h = Tensor(np.zeros((1, self.batch_size,
self.hidden_size)), mstype.float32)
        self.dynamicRNN_c = Tensor(np.zeros((1, self.batch_size,
self.hidden_size)), mstype.float32)
        self.cast = P.Cast()
        self.is_ascend = context.get_context("device_target") == "Ascend"
        if self.is_ascend:
            self.compute_type = mstype.float16
            self.rnn = P.DynamicRNN()
        else:
            self.compute_type = mstype.float32
            self.lstm = nn.LSTM(self.input_size,
```

```
                                self.hidden_size,
                                num_layers=1,
                                has_bias=True,
                                batch_first=False,
                                dropout=0.0,
                                bidirectional=False)

    def construct(self, x, init_h=None, init_c=None):
        """DynamicRNNCell Network."""
        if init_h is None or init_c is None:
            init_h = self.cast(self.dynamicRNN_h, self.compute_type)
            init_c = self.cast(self.dynamicRNN_c, self.compute_type)
        if self.is_ascend:
            w = self.cast(self.dynamicRNN_w, self.compute_type)
            b = self.cast(self.dynamicRNN_b, self.compute_type)
            output, hn, cn, _, _, _, _, _ = self.rnn(x, w, b, None, init_h,
init_c)
        else:
            output, (hn, cn) = self.lstm(x, (init_h, init_c))
        return output, hn, cn
```

### 3.2.2 DynamicRNNNet

定义`DynamicRNNNet`类，该类封装了上述定义的`DynamicRNNCell`类，能够实现RNN功能。

```
class DynamicRNNNet(nn.Cell):
    """
    DynamicRNN Network.

    Args:
        seq_length (int): Lengths of setences.
        batchsize (int): Batch size.
        word_embed_dim (int): Input size.
        hidden_size (int): Hidden size.
    """

    def __init__(self,
                 seq_length=80,
                 batchsize=128,
                 word_embed_dim=1024,
                 hidden_size=1024):
        super(DynamicRNNNet, self).__init__()
        self.max_length = seq_length
        self.hidden_size = hidden_size
        self.cast = P.Cast()
        self.concat = P.Concat(axis=0)
        self.get_shape = P.Shape()
        self.net = DynamicRNNCell(num_setp=seq_length,
                                  batch_size=batchsize,
                                  word_embed_dim=word_embed_dim,
                                  hidden_size=hidden_size)
        self.is_ascend = context.get_context("device_target") == "Ascend"
        if self.is_ascend:
            self.compute_type = mstype.float16
        else:
            self.compute_type = mstype.float32
```

```
    def construct(self, inputs, init_state=None):
        """DynamicRNN Network."""
        inputs = self.cast(inputs, self.compute_type)
        if init_state is not None:
            init_h = self.cast(init_state[0:1, :, :], self.compute_type)
            init_c = self.cast(init_state[-1:, :, :], self.compute_type)
            out, state_h, state_c = self.net(inputs, init_h, init_c)
        else:
            out, state_h, state_c = self.net(inputs)
        out = self.cast(out, mstype.float32)
        state = self.concat((state_h[-1:, :, :], state_c[-1:, :, :]))
        state = self.cast(state, mstype.float32)
        # out:[T,b,D], state:[2,b,D]
        return out, state
```

### 3.2.3 GNMTEncoder

多层堆叠的LSTM网络通常会比层数少的网络有更好的性能，然而，简单的错层堆叠会造成训练的缓慢以及容易受到剃度爆炸或梯度消失的影响，在实验中，简单堆叠在4层工作良好，6层简单堆叠性能还好的网络很少见，8层的就更罕见了，为了解决这个问题，在模型中引入了残差连接。

通过上述定义的`DynamicRNNNet`实现在`Ascend`和`GPU`上都可以运行的`GNMTEncoder`。本文实现的GNMT v2的Encoder相对于GNMT有所改进，本文所需要构造的GNMT v2的Encoder结构如下：

- 总共4层LSTM，每层的隐藏向量大小1024，第一层为双向LSTM，其余为单向LSTM。
- 从第三层开始加上残差连接。
- 所有LSTM层的输入都应用dropout，dropout概率设置为0.2。
- LSTM层的隐藏状态初始化为零。
- LSTM层的`weight`和`bias`用均匀分布`(-0.1, 0.1)`初始化。

```
from mindspore import nn
from mindspore.ops import operations as P
from mindspore.common import dtype as mstype


class GNMTEncoder(nn.Cell):
    """
    Implements of GNMT encoder.

    Args:
        config: Configuration of GNMT network.
        is_training (bool): Whether to train.
        compute_type (mstype): Mindspore data type.

    Returns:
        Tensor, shape of (N, T, D).
    """

    def __init__(self,
                 config,
                 is_training: bool,
                 compute_type=mstype.float32):
        super(GNMTEncoder, self).__init__()
        self.input_mask_from_dataset = config.input_mask_from_dataset
        self.max_positions = config.seq_length
        self.attn_embed_dim = config.hidden_size
```

```
        self.num_layers = config.num_hidden_layers
        self.hidden_dropout_prob = config.hidden_dropout_prob
        self.vocab_size = config.vocab_size
        self.seq_length = config.seq_length
        self.batch_size = config.batch_size
        self.word_embed_dim = config.hidden_size

        self.transpose = P.Transpose()
        self.transpose_orders = (1, 0, 2)
        self.reshape = P.Reshape()
        self.concat = P.Concat(axis=-1)
        encoder_layers = []
        for i in range(0, self.num_layers + 1):
            if i == 2:
                # the bidirectional layer's output is [T,D,2N]
                scaler = 2
            else:
                # the rest layer's output is [T,D,N]
                scaler = 1
            layer = DynamicRNNNet(seq_length=self.seq_length,
                                  batchsize=self.batch_size,
                                  word_embed_dim=scaler * self.word_embed_dim,
                                  hidden_size=self.word_embed_dim)
            encoder_layers.append(layer)
        self.encoder_layers = nn.CellList(encoder_layers)
        self.reverse_v2 = P.ReverseV2(axis=[0])
        self.dropout = nn.Dropout(keep_prob=1.0 - config.hidden_dropout_prob)

    def construct(self, inputs):
        """Encoder."""
        inputs = self.dropout(inputs)
        # bidirectional layer, fwd_encoder_outputs: [T,N,D]
        fwd_encoder_outputs, _ = self.encoder_layers[0](inputs)

        # the input need reverse.
        inputs_r = self.reverse_v2(inputs)
        bak_encoder_outputs, _ = self.encoder_layers[1](inputs_r)
        # the result need reverse.
        bak_encoder_outputs = self.reverse_v2(bak_encoder_outputs)

        # bi_encoder_outputs: [T,N,2D]
        bi_encoder_outputs = self.concat((fwd_encoder_outputs,
bak_encoder_outputs))

        # 1st unidirectional layer. encoder_outputs: [T,N,D]
        bi_encoder_outputs = self.dropout(bi_encoder_outputs)
        encoder_outputs, _ = self.encoder_layers[2](bi_encoder_outputs)
        # Build all the rest unidi layers of encoder
        for i in range(3, self.num_layers + 1):
            residual = encoder_outputs
            encoder_outputs = self.dropout(encoder_outputs)
            # [T,N,D] -> [T,N,D]
            encoder_outputs_o, _ = self.encoder_layers[i](encoder_outputs)
            encoder_outputs = encoder_outputs_o + residual

        return encoder_outputs
```

# 3.3 Decoder

## 3.3.1 BahdanauAttention

Luong Attention 和 Bahdanau Attention 是最经典的两种注意力机制，我们这里实现Bahdanau Attention机制。

Bahdanau本质是一种 加性attention机制，将decoder的隐状态和encoder所有位置输出通过线性组合对齐，得到context向量，用于改善序列到序列的翻译模型。

其本质是两层全连接网络，隐藏层激活函数tanh，输出层维度为1。

Bahdanau的特点为：

- 编码器隐状态 ： 编码器对于每一个输入向量产生一个隐状态向量；
- 计算对齐分数：使用上一时刻的隐状态$\boldsymbol s_{t-1}$和编码器每个位置输出$\boldsymbol x_i$计算对齐分数（使用前馈神经网络计算），编码器最终时刻隐状态可作为解码器初始时刻隐状态；
- 概率化对齐分数：解码器上一时刻隐状态$\boldsymbol s_{t-1}$在编码器每个位置输出的对齐分数，通过softmax转化为概率分布向量；
- 计算上下文向量：根据概率分布化的对齐分数，加权编码器各位置输出，得上下文向量$\boldsymbol c_t$；
- 解码器输出：将上下文向量$\boldsymbol c_t$和上一时刻编码器输出$\hat y*{t-1} *$对应的*embedding*拼接，作为当前时刻编码器输入，经*RNN*网络产生新的输出和隐状态，训练过程中有真实目标序列*$\boldsymbol y=(y_1\cdots y_m)$*，多使用*$y*{t-1}$取代$\hat y_{t-1}$作为解码器$t$时刻输入；

时刻$t$，解码器的隐状态表示为

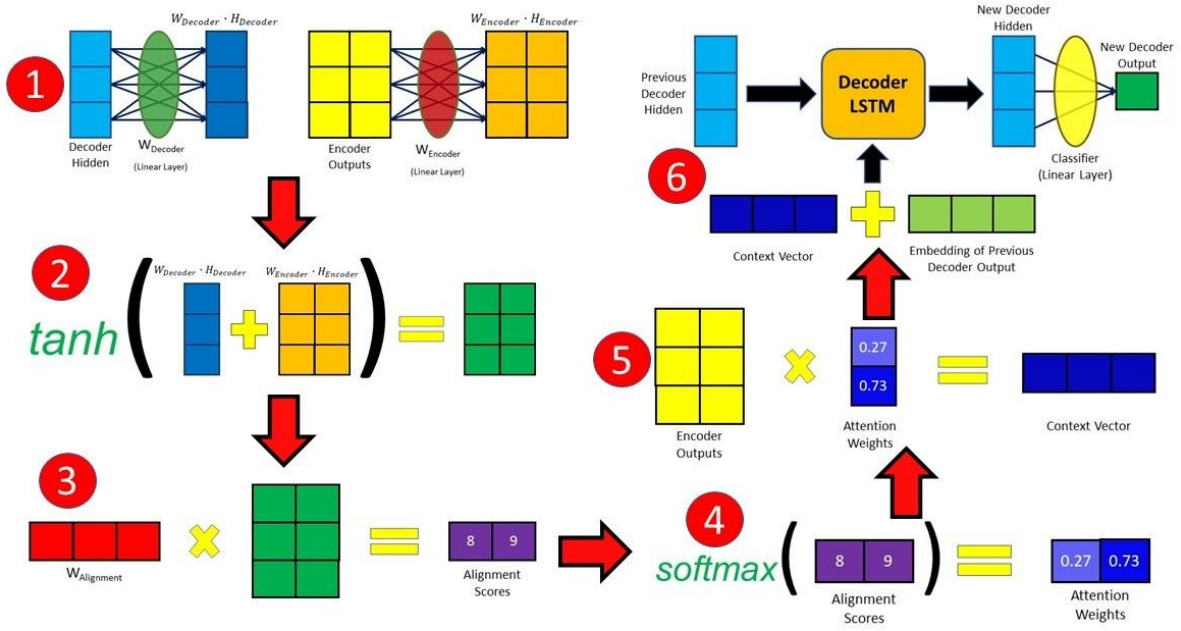
$$\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, \boldsymbol{c}_t, y_{t-1})$$

时刻$t$的隐状态$\boldsymbol s_{t-1}$

对编码器各时刻输出X XX的注意力分数为：

$$\boldsymbol{\alpha}_t(\boldsymbol{s}_{t-1}, X) = \mathrm{softmax}(\tanh\left(\boldsymbol{s}_{t-1}W_{decoder} + XW_{encoder}\right)W_{alignment}), \quad \boldsymbol{c}_t = \sum_i \alpha_{ti}\boldsymbol{x}_i$$

使用Bahdanau注意力机制的解码过程：




```
import mindspore.common.dtype as mstype
import mindspore.ops.operations as P
from mindspore import nn
from mindspore.common.tensor import Tensor
from mindspore.common.parameter import Parameter
from mindspore.common.initializer import Uniform

INF = 65504.0


class BahdanauAttention(nn.Cell):
    """
    Constructor for the BahdanauAttention.

    Args:
        is_training (bool): Whether to train.
        query_size (int): feature dimension for query.
        key_size (int): feature dimension for keys.
        num_units (int): internal feature dimension.
        normalize (bool): Whether to normalize.
        initializer_range: range for uniform initializer parameters.

    Returns:
        Tensor, shape (t_q_length, N, D).
    """

    def __init__(self,
                 is_training,
                 query_size,
                 key_size,
                 num_units,
                 normalize=False,
                 initializer_range=0.1,
                 compute_type=mstype.float16):
        super(BahdanauAttention, self).__init__()
        self.is_training = is_training
        self.mask = None
        self.query_size = query_size
        self.key_size = key_size
```

```
        self.normalize = normalize
        self.num_units = num_units
        self.linear_att = Parameter(Tensor(np.random.uniform(-initializer_range,
initializer_range, size=[num_units]),
                                           dtype=mstype.float32))
        if self.normalize:
            self.normalize_scalar = Parameter(Tensor(np.array([1.0 /
num_units]), dtype=mstype.float32))
            self.normalize_bias = Parameter(Tensor(np.zeros(num_units),
dtype=mstype.float32))
        self.transpose = P.Transpose()
        self.transpose_orders = (1, 0, 2)
        self.shape_op = P.Shape()

        self.linear_q = nn.Dense(query_size,
                                 num_units,
                                 has_bias=False,

weight_init=Uniform(initializer_range)).to_float(compute_type)

        self.linear_k = nn.Dense(key_size,
                                 num_units,
                                 has_bias=False,

weight_init=Uniform(initializer_range)).to_float(compute_type)
        self.expand = P.ExpandDims()
        self.tile = P.Tile()

        self.norm = nn.Norm(axis=-1)
        self.mul = P.Mul()
        self.matmul = P.MatMul()
        self.batchMatmul = P.BatchMatMul()
        self.tanh = nn.Tanh()

        self.matmul_trans_b = P.BatchMatMul(transpose_b=True)
        self.softmax = nn.Softmax(axis=-1)
        self.reshape = P.Reshape()
        self.cast = P.Cast()

    def construct(self, query, keys, attention_mask=None):
        """
        Construct attention block.

        Args:
            query (Tensor): Shape (t_q_length, N, D).
            keys (Tensor): Shape (t_k_length, N, D).
            attention_mask: Shape(N, t_k_length).
        Returns:
            Tensor, shape (t_q_length, N, D).
        """

        # (t_k_length, N, D) -> (N, t_k_length, D).
        keys = self.transpose(keys, self.transpose_orders)
        # (t_q_length, N, D) -> (N, t_q_length, D).
        query_trans = self.transpose(query, self.transpose_orders)

        query_shape = self.shape_op(query_trans)
        batch_size = query_shape[0]
```

```
        t_q_length = query_shape[1]
        t_k_length = self.shape_op(keys)[1]

        # (N, t_q_length, D)
        query_trans = self.reshape(query_trans, (batch_size * t_q_length,
self.query_size))
        if self.is_training:
            query_trans = self.cast(query_trans, mstype.float16)
        processed_query = self.linear_q(query_trans)
        if self.is_training:
            processed_query = self.cast(processed_query, mstype.float32)
        processed_query = self.reshape(processed_query, (batch_size, t_q_length,
self.num_units))
        # (N, t_k_length, D)
        keys = self.reshape(keys, (batch_size * t_k_length, self.key_size))
        if self.is_training:
            keys = self.cast(keys, mstype.float16)
        processed_key = self.linear_k(keys)
        if self.is_training:
            processed_key = self.cast(processed_key, mstype.float32)
        processed_key = self.reshape(processed_key, (batch_size, t_k_length,
self.num_units))

        # scores: (N, t_q_length, t_k_length)
        scores = self.obtain_score(processed_query, processed_key)
        # attention_mask: (N, t_k_length)
        mask = attention_mask
        if mask is not None:
            mask = 1.0 - mask
            mask = self.tile(self.expand(mask, 1), (1, t_q_length, 1))
            scores += mask * (-INF)
        # [batch_size, t_q_length, t_k_length]
        scores_softmax = self.softmax(scores)

        keys = self.reshape(keys, (batch_size, t_k_length, self.key_size))
        if self.is_training:
            keys = self.cast(keys, mstype.float16)
            scores_softmax_fp16 = self.cast(scores_softmax, mstype.float16)
        else:
            scores_softmax_fp16 = scores_softmax

        # (b, t_q_length, D)
        context_attention = self.batchMatmul(scores_softmax_fp16, keys)
        # [t_q_length, b, D]
        context_attention = self.transpose(context_attention,
self.transpose_orders)
        if self.is_training:
            context_attention = self.cast(context_attention, mstype.float32)

        return context_attention, scores_softmax

    def obtain_score(self, attention_q, attention_k):
        """
        Calculate Bahdanau score

        Args:
            attention_q: (batch_size, t_q_length, D).
            attention_k: (batch_size, t_k_length, D).
```

```
        returns:
            scores: (batch_size, t_q_length, t_k_length).
        """
        batch_size, t_k_length, D = self.shape_op(attention_k)
        t_q_length = self.shape_op(attention_q)[1]
        # (batch_size, t_q_length, t_k_length, n)
        attention_q = self.tile(self.expand(attention_q, 2), (1, 1, t_k_length,
1))
        attention_k = self.tile(self.expand(attention_k, 1), (1, t_q_length, 1,
1))
        # (batch_size, t_q_length, t_k_length, n)
        sum_qk_add = attention_q + attention_k

        if self.normalize:
            # (batch_size, t_q_length, t_k_length, n)
            sum_qk_add = sum_qk_add + self.normalize_bias
            linear_att_norm = self.linear_att / self.norm(self.linear_att)
            linear_att_norm = self.cast(linear_att_norm, mstype.float32)
            linear_att_norm = self.mul(linear_att_norm, self.normalize_scalar)
        else:
            linear_att_norm = self.linear_att

        linear_att_norm = self.expand(linear_att_norm, -1)
        sum_qk_add = self.reshape(sum_qk_add, (-1, D))

        tanh_sum_qk = self.tanh(sum_qk_add)
        if self.is_training:
            linear_att_norm = self.cast(linear_att_norm, mstype.float16)
            tanh_sum_qk = self.cast(tanh_sum_qk, mstype.float16)

        scores_out = self.matmul(tanh_sum_qk, linear_att_norm)

        # (N, t_q_length, t_k_length)
        scores_out = self.reshape(scores_out, (batch_size, t_q_length,
t_k_length))
        if self.is_training:
            scores_out = self.cast(scores_out, mstype.float32)
        return scores_out
```

### 3.3.2 RecurrentAttention

定义`RecurrentAttention`类，该类将上述定义的RNN和`BahdanauAttention`结合起来，封装成为一个适用于循环神经网络的`Attention`类。主要原理就是将RNN网络的输出输入到上述`BahdanauAttention`中，返回得到上下文注意力和 Bahdanau score。

```
class RecurrentAttention(nn.Cell):
    """
    Constructor for the RecurrentAttention.

    Args:
        input_size: number of features in input tensor.
        context_size: number of features in output from encoder.
        hidden_size: internal hidden size.
        num_layers: number of layers in LSTM.
        dropout: probability of dropout (on input to LSTM layer).
```

```
        initializer_range: range for the uniform initializer.

    Returns:
        Tensor, shape (N, T, D).
    """

    def __init__(self,
                 rnn,
                 is_training=True,
                 input_size=1024,
                 context_size=1024,
                 hidden_size=1024,
                 num_layers=1,
                 dropout=0.2,
                 initializer_range=0.1):
        super(RecurrentAttention, self).__init__()
        self.dropout = nn.Dropout(keep_prob=1.0 - dropout)
        self.rnn = rnn
        self.attn = BahdanauAttention(is_training=is_training,
                                      query_size=hidden_size,
                                      key_size=hidden_size,
                                      num_units=hidden_size,
                                      normalize=True,
                                      initializer_range=initializer_range,
                                      compute_type=mstype.float16)

    def construct(self, decoder_embedding, context_key, attention_mask=None,
rnn_init_state=None):
        # decoder_embedding: [t_q,N,D]
        # context: [t_k,N,D]
        # attention_mask: [N,t_k]
        # [t_q,N,D]
        decoder_embedding = self.dropout(decoder_embedding)
        rnn_outputs, rnn_state = self.rnn(decoder_embedding, rnn_init_state)
        # rnn_outputs:[t_q,b,D], attn_outputs:[t_q,b,D], scores:[b, t_q, t_k],
rnn_state:tuple([2,b,D]).
        attn_outputs, scores = self.attn(query=rnn_outputs, keys=context_key,
attention_mask=attention_mask)
        return rnn_outputs, attn_outputs, rnn_state, scores
```

### 3.3.3 GNMTDecoder

- 具有隐藏大小 1024 和全连接分类器的 4 层单向 LSTM
- 残差连接从第 3 层开始
- dropout 应用于所有 LSTM 层的输入，dropout 的概率设置为 0.2
- LSTM 层的隐藏状态由编码器的最后一个隐藏状态初始化
- LSTM 层的权重和偏差初始化为均匀 (-0.1, 0.1) 分布
- 全连接分类器的权重和偏差以均匀 (-0.1, 0.1) 分布初始化
- Decoder过程中采用beam search。

上文中提到过两个重要的改进，coverage penalty and length normalization。以往的beam search会有利于偏短的结果，谷歌认为这是不合理的，并对长度进行了标准化处理

$s(Y,X)=log(P(Y|X))/lp(Y)+cp(X;Y)$

```
class GNMTDecoder(nn.Cell):
    """
    Implements of Transformer decoder.

    Args:
        attn_embed_dim (int): Dimensions of attention layer.
        decoder_layers (int): Decoder layers.
        num_attn_heads (int): Attention heads number.
        intermediate_size (int): Hidden size of FFN.
        attn_dropout_prob (float): Dropout rate in attention. Default: 0.1.
        initializer_range (float): Initial range. Default: 0.02.
        dropout_prob (float): Dropout rate between layers. Default: 0.1.
        hidden_act (str): Non-linear activation function in FFN. Default:
"relu".
        compute_type (mstype): Mindspore data type. Default: mstype.float32.

    Returns:
        Tensor, shape of (N, T', D).
    """

    def __init__(self,
                 config,
                 is_training: bool,
                 use_one_hot_embeddings: bool = False,
                 initializer_range=0.1,
                 infer_beam_width=1,
                 compute_type=mstype.float16):
        super(GNMTDecoder, self).__init__()

        self.is_training = is_training
        self.attn_embed_dim = config.hidden_size
        self.num_layers = config.num_hidden_layers
        self.hidden_dropout_prob = config.hidden_dropout_prob
        self.vocab_size = config.vocab_size
        self.seq_length = config.max_decode_length
        # batchsize* beam_width for beam_search.
        self.batch_size = config.batch_size * infer_beam_width
        self.word_embed_dim = config.hidden_size
        self.transpose = P.Transpose()
        self.transpose_orders = (1, 0, 2)
        self.reshape = P.Reshape()
        self.concat = P.Concat(axis=-1)
        self.state_concat = P.Concat(axis=0)
        self.all_decoder_state = Tensor(np.zeros([self.num_layers, 2,
self.batch_size, config.hidden_size]),
                                        mstype.float32)

        decoder_layers = []
        for i in range(0, self.num_layers):
            if i == 0:
                # the inputs is [T,D,N]
                scaler = 1
            else:
                # the inputs is [T,D,2N]
                scaler = 2
            layer = DynamicRNNNet(seq_length=self.seq_length,
                                  batchsize=self.batch_size,
                                  word_embed_dim=scaler * self.word_embed_dim,
```

```
                                    hidden_size=self.word_embed_dim)
            decoder_layers.append(layer)
        self.decoder_layers = nn.CellList(decoder_layers)

        self.att_rnn = RecurrentAttention(rnn=self.decoder_layers[0],
                                          is_training=is_training,
                                          input_size=self.word_embed_dim,
                                          context_size=self.attn_embed_dim,
                                          hidden_size=self.attn_embed_dim,
                                          num_layers=1,
                                          dropout=config.attention_dropout_prob)

        self.dropout = nn.Dropout(keep_prob=1.0 - config.hidden_dropout_prob)

        self.classifier = nn.Dense(config.hidden_size,
                                   config.vocab_size,
                                   has_bias=True,
                                   weight_init=Uniform(initializer_range),

bias_init=Uniform(initializer_range)).to_float(compute_type)
        self.cast = P.Cast()
        self.shape_op = P.Shape()
        self.expand = P.ExpandDims()

    def construct(self, tgt_embeddings, encoder_outputs, attention_mask=None,
                  decoder_init_state=None):
        """Decoder."""
        # tgt_embeddings: [T',N,D], encoder_outputs: [T,N,D], attention_mask:
[N,T].
        query_shape = self.shape_op(tgt_embeddings)
        if decoder_init_state is None:
            hidden_state = self.all_decoder_state
        else:
            hidden_state = decoder_init_state
        # x:[t_q,b,D], attn:[t_q,b,D], scores:[b, t_q, t_k], state_0:[2,b,D].
        x, attn, state_0, scores =
self.att_rnn(decoder_embedding=tgt_embeddings, context_key=encoder_outputs,
                                                attention_mask=attention_mask,
rnn_init_state=hidden_state[0, :, :, :])
        x = self.concat((x, attn))
        x = self.dropout(x)
        decoder_outputs, state_1 = self.decoder_layers[1](x, hidden_state[1, :,
:, :])

        all_decoder_state = self.state_concat((self.expand(state_0, 0),
self.expand(state_1, 0)))

        for i in range(2, self.num_layers):
            residual = decoder_outputs
            decoder_outputs = self.concat((decoder_outputs, attn))

            decoder_outputs = self.dropout(decoder_outputs)
            # 1st unidirectional layer. encoder_outputs: [T,N,D]
            decoder_outputs, decoder_state = self.decoder_layers[i]
(decoder_outputs, hidden_state[i, :, :, :])
            decoder_outputs += residual
            all_decoder_state = self.state_concat((all_decoder_state,
self.expand(decoder_state, 0)))
```

```
        # [m, batch_size * beam_width, D]
        decoder_outputs = self.reshape(decoder_outputs, (-1,
self.word_embed_dim))
        if self.is_training:
            decoder_outputs = self.cast(decoder_outputs, mstype.float16)
        decoder_outputs = self.classifier(decoder_outputs)
        if self.is_training:
            decoder_outputs = self.cast(decoder_outputs, mstype.float32)
        # [m, batch_size * beam_width, V]
        decoder_outputs = self.reshape(decoder_outputs, (query_shape[0],
query_shape[1], self.vocab_size))
        # all_decoder_state:[4,2,b,D]
        return decoder_outputs, all_decoder_state, scores
```

## 3.4 推理过程

### 3.4.1 CreateAttentionPaddingsFromInputPaddings

首先封装 `CreateAttentionPaddingsFromInputPaddings` 类，该类负责产出一个[batch_size, seq_length, seq_length]的矩阵,主要为了在计算self-attention系数时,padding位置不参与attention系数更新,具体padding位置由input_mask负责记录。

可以用下图来解释，假设下图中的向量是8个句子做完embedding后的向量，后面的0代表句子的长度已结束。此时第一个句子的第一个编码在后面做self-Attention所需要和该句子中的其他向量计算，那么该和哪些向量计算呐？此处就是用新增加的纬度来表示需要计算的词向量，图中下部分是转换成3D后新增加的一个向量来表示和哪些词进行计算，1代表能计算，0代表不进行计算。

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 45 | 56 | 87 | 12 | 0 | 0 | 0 | 0 | ..... | 0 | 0 | 0 |
| . | | | | | | | | | | | |
| . | | | | | | | | | | | |
| 36 | 56 | 12 | 78 | 63 | 35 | 32 | 0 | ..... | 0 | 0 | 0 |

45: 1111000...000
56: 1111000...000
87: 1111000...000
12: 1111000...000

36: 11111110...000
56: 11111110...000
12: 11111110...000
78: 11111110...000
63: 11111110...000
35: 11111110...000
32: 11111110...000

```
class CreateAttentionPaddingsFromInputPaddings(nn.Cell):
    """
    Create attention mask according to input mask.

    Args:
        config: Config class.

    Returns:
        Tensor, shape of (N, T, T).
    """
```

```
    def __init__(self,
                 config,
                 is_training=True):
        super(CreateAttentionPaddingsFromInputPaddings, self).__init__()

        self.is_training = is_training
        self.input_mask = None
        self.cast = P.Cast()
        self.shape = P.Shape()
        self.reshape = P.Reshape()
        self.batch_matmul = P.BatchMatMul()
        self.multiply = P.Mul()
        self.shape = P.Shape()
        # mask future positions
        ones = np.ones(shape=(config.batch_size, config.seq_length,
config.seq_length))
        self.lower_triangle_mask = Tensor(np.tril(ones), dtype=mstype.float32)

    def construct(self, input_mask, mask_future=False):
        """
        Construct network.

        Args:
            input_mask (Tensor): Tensor mask vectors with shape (N, T).
            mask_future (bool): Whether mask future (for decoder training).

        Returns:
            Tensor, shape of (N, T, T).
        """
        input_shape = self.shape(input_mask)
        # Add this for infer as the seq_length will increase.
        shape_right = (input_shape[0], 1, input_shape[1])
        shape_left = input_shape + (1,)
        if self.is_training:
            input_mask = self.cast(input_mask, mstype.float16)
        mask_left = self.reshape(input_mask, shape_left)
        mask_right = self.reshape(input_mask, shape_right)

        attention_mask = self.batch_matmul(mask_left, mask_right)
        if self.is_training:
            attention_mask = self.cast(attention_mask, mstype.float32)

        if mask_future:
            attention_mask = self.multiply(attention_mask,
self.lower_triangle_mask)

        return attention_mask
```

### 3.4.2 TileBeam

定义`TileBeam`类来实现光束平铺操作，方便后续使用光速搜索算法。这里首先使用`ExpandDims()`对输入的`input_tensor`在给定的轴上添加额外维度，然后使用`Tile()`方法来按照给定的次数复制输入Tensor到指定`beam_width`的光速宽度，然后使用`Reshape()`操作重新划分向量形状。

```
class TileBeam(nn.Cell):
```

```
    """
    Beam Tile operation.

    Args:
        beam_width (int): The Number of beam.
        compute_type (mstype): Mindspore data type. Default: mstype.float32.
    """

    def __init__(self, beam_width, compute_type=mstype.float32):
        super(TileBeam, self).__init__()
        self.beam_width = beam_width

        self.expand = P.ExpandDims()
        self.tile = P.Tile()
        self.reshape = P.Reshape()
        self.shape = P.Shape()

    def construct(self, input_tensor):
        """
        Process source sentence

        Inputs:
            input_tensor (Tensor):  with shape (N, T, D).

        Returns:
            Tensor, tiled tensor.
        """
        shape = self.shape(input_tensor)
        # add an dim
        input_tensor = self.expand(input_tensor, 1)
        # get tile shape: [1, beam, ...]
        tile_shape = (1,) + (self.beam_width,)
        for _ in range(len(shape) - 1):
            tile_shape = tile_shape + (1,)
        # tile
        output = self.tile(input_tensor, tile_shape)
        # reshape to [batch*beam, ...]
        out_shape = (shape[0] * self.beam_width,) + shape[1:]
        output = self.reshape(output, out_shape)

        return output
```

### 3.4.3 SaturateCast

定义`SaturateCast`类，该类是一个工具类，主要是用来将一种数据类型安全的转换成另一种数据类型。

```
class SaturateCast(nn.Cell):
    """Cast wrapper."""

    def __init__(self, dst_type=mstype.float32):
        super(SaturateCast, self).__init__()
        self.cast = P.Cast()
        self.dst_type = dst_type

    def construct(self, x):
        return self.cast(x, self.dst_type)
```

### 3.4.4 PredLogProbs

定义`PredLogProbs`类，该类利用Log Softmax激活函数按元素计算，输入经Softmax函数、Log函数转换后得到预测的概率取值，值的范围在[-inf,0)。

```
class PredLogProbs(nn.Cell):
    """
    Get log probs.

    Args:
        batch_size (int): Batch size of input dataset.
        seq_length (int): The length of sequences.
        width (int): Number of parameters of a layer
        compute_type (int): Type of input type.
        dtype (int): Type of MindSpore output type.
    """

    def __init__(self,
                 batch_size,
                 seq_length,
                 width,
                 compute_type=mstype.float32,
                 dtype=mstype.float32):
        super(PredLogProbs, self).__init__()
        self.batch_size = batch_size
        self.seq_length = seq_length
        self.width = width
        self.compute_type = compute_type
        self.dtype = dtype
        self.log_softmax = nn.LogSoftmax(axis=-1)
        self.cast = P.Cast()

    def construct(self, logits):
        """
        Calculate the log_softmax.

        Inputs:
            input_tensor (Tensor): A batch of sentences with shape (N, T).
            output_weights (Tensor): A batch of masks with shape (N, T).

        Returns:
            Tensor, the prediction probability with shape (N, T').
        """
        log_probs = self.log_softmax(logits)
        return log_probs
```

### 3.4.5 BeamDecoderStep

```
class BeamDecoderStep(nn.Cell):
    """
    Multi-layer transformer decoder step.

    Args:
        config: The config of Transformer.
    """

    def __init__(self,
                 config,
                 use_one_hot_embeddings,
                 compute_type=mstype.float32):
        super(BeamDecoderStep, self).__init__(auto_prefix=True)

        self.vocab_size = config.vocab_size
        self.word_embed_dim = config.hidden_size
        self.embedding_lookup = EmbeddingLookup(
            is_training=False,
            vocab_size=config.vocab_size,
            embed_dim=self.word_embed_dim,
            use_one_hot_embeddings=use_one_hot_embeddings)

        self.projection = PredLogProbs(
            batch_size=config.batch_size * config.beam_width,
            seq_length=1,
            width=config.vocab_size,
            compute_type=config.compute_type)

        self.seq_length = config.max_decode_length
        self.decoder = GNMTDecoder(config,
                                   is_training=False,
                                   infer_beam_width=config.beam_width)

        self.ones_like = P.OnesLike()
        self.shape = P.Shape()

        self.create_att_paddings_from_input_paddings =
CreateAttentionPaddingsFromInputPaddings(config,

               is_training=False)
        self.expand = P.ExpandDims()
        self.multiply = P.Mul()

        ones = np.ones(shape=(config.max_decode_length,
config.max_decode_length))
        self.future_mask = Tensor(np.tril(ones), dtype=mstype.float32)

        self.cast_compute_type = SaturateCast(dst_type=compute_type)

        self.transpose = P.Transpose()
        self.transpose_orders = (1, 0, 2)
```

```
    def construct(self, input_ids, enc_states, enc_attention_mask,
decoder_hidden_state=None):
        """
        Get log probs.

        Args:
            input_ids: [batch_size * beam_width, m]
            enc_states: [batch_size * beam_width, T, D]
            enc_attention_mask: [batch_size * beam_width, T]
            decoder_hidden_state: [decoder_layers_nums, 2, batch_size *
beam_width, hidden_size].

        Returns:
            Tensor, the log_probs. [batch_size * beam_width, 1, vocabulary_size]
        """

        # 处理词嵌入过程. input_embedding: [batch_size * beam_width, m, D],
embedding_tables: [V, D]
        input_embedding, _ = self.embedding_lookup(input_ids)
        input_embedding = self.cast_compute_type(input_embedding)

        input_shape = self.shape(input_ids)
        input_len = input_shape[1]
        # [m, batch_size * beam_width, D]
        input_embedding = self.transpose(input_embedding, self.transpose_orders)
        enc_states = self.transpose(enc_states, self.transpose_orders)

        # decoder_output: [m, batch_size*beam_width, V], scores:[b, t_q, t_k],
all_decoder_state:[4,2,b*beam_width,D]
        decoder_output, all_decoder_state, scores =
self.decoder(input_embedding, enc_states, enc_attention_mask,

decoder_hidden_state)
        # [batch_size * beam_width, m, v]
        decoder_output = self.transpose(decoder_output, self.transpose_orders)

        # take the last step, [batch_size * beam_width, 1, V]
        decoder_output = decoder_output[:, (input_len - 1):input_len, :]

        # 投影并计算log的概率
        log_probs = self.projection(decoder_output)

        # [batch_size * beam_width, 1, vocabulary_size]
        return log_probs, all_decoder_state, scores
```

### 3.4.6 BeamSearchDecoder

beam search 增加了 长度归一化 和 覆盖惩罚。

beam search倾向较短的结果，并且在decoder中需要比较长度不同的句子。所以需要使用长度归一化

覆盖机制coverage penalty，使注意力模块完全覆盖源句。

$$s(Y,X) = \log(P(Y|X))/lp(Y) + cp(X;Y)$$
$$lp(Y) = \frac{(5+|Y|)^\alpha}{(5+1)^\alpha}$$
$$cp(X;Y) = \beta * \sum_{i=1}^{|X|} \log(\min(\sum_{j=1}^{|Y|} p_{i,j}, 1.0)),$$



$p_{ij}$使第j个target word在第i个source word上的概率。

BLEU对BLEU评分而言

$\alpha=0.2, \beta=0.2$可以将原始BLEU评分增加1.1

RL之后的MT模型在优化后的beam search上BLEU没有提升

```
class BeamSearchDecoder(nn.Cell):
    """
    Beam search decoder.

    Args:
        batch_size (int): Batch size of input dataset.
        seq_length (int): Length of input sequence.
        vocab_size (int): The shape of each embedding vector.
        decoder (Cell): The GNMT decoder.
        beam_width (int): Beam width for beam search in inferring. Default: 4.
        decoder_layers_nums (int): The nums of decoder layers.
        length_penalty_weight (float): Penalty for sentence length. Default:
0.6.
        max_decode_length (int): Max decode length for inferring. Default: 64.
        sos_id (int): The index of start label <SOS>. Default: 1.
        eos_id (int): The index of end label <EOS>. Default: 2.
        compute_type (:class:`mindspore.dtype`): Compute type. Default:
mstype.float32.

    Returns:
        Tensor, predictions output.
    """

    def __init__(self,
                 batch_size,
                 seq_length,
                 vocab_size,
                 decoder,
                 beam_width=4,
                 decoder_layers_nums=4,
                 length_penalty_weight=0.6,
                 cov_penalty_factor=0.1,
                 hidden_size=1024,
                 max_decode_length=64,
                 sos_id=2,
                 eos_id=3,
                 is_using_while=True,
                 compute_type=mstype.float32):
        super(BeamSearchDecoder, self).__init__()
```

```
        self.encoder_length = seq_length
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.vocab_size = vocab_size
        self.beam_width = beam_width
        self.decoder_layers_nums = decoder_layers_nums
        self.length_penalty_weight = length_penalty_weight
        self.cov_penalty_factor = cov_penalty_factor
        self.max_decode_length = max_decode_length
        self.decoder = decoder
        self.is_using_while = is_using_while

        self.add = P.Add()
        self.expand = P.ExpandDims()
        self.reshape = P.Reshape()
        self.shape_flat = (-1,)
        self.shape = P.Shape()

        self.zero_tensor = Tensor(np.zeros([batch_size, beam_width]),
mstype.float32)
        self.ninf_tensor = Tensor(np.full([batch_size, beam_width], -INF),
mstype.float32)

        self.select = P.Select()
        self.flat_shape = (batch_size, beam_width * vocab_size)
        self.topk = P.TopK(sorted=True)
        self.vocab_size_tensor = Tensor(self.vocab_size, mstype.int32)
        self.real_div = P.RealDiv()
        self.equal = P.Equal()
        self.eos_ids = Tensor(np.full([batch_size, beam_width], eos_id),
mstype.int32)

        beam_ids = np.tile(np.arange(beam_width).reshape((1, beam_width)),
[batch_size, 1])
        self.beam_ids = Tensor(beam_ids, mstype.int32)

        batch_ids = np.arange(batch_size * beam_width).reshape((batch_size,
beam_width)) // beam_width
        self.batch_ids = Tensor(batch_ids, mstype.int32)

        self.concat = P.Concat(axis=-1)
        self.gather_nd = P.GatherNd()

        self.start_ids = Tensor(np.full([batch_size * beam_width, 1], sos_id),
mstype.int32)
        if self.is_using_while:
            self.start = Tensor(0, dtype=mstype.int32)
            self.init_seq = Tensor(np.full([batch_size, beam_width,
self.max_decode_length + 1], sos_id),
                                   mstype.int32)
        else:
            self.init_seq = Tensor(np.full([batch_size, beam_width, 1], sos_id),
mstype.int32)

        init_scores = np.tile(np.array([[0.] + [-INF] * (beam_width - 1)]),
[batch_size, 1])
        self.init_scores = Tensor(init_scores, mstype.float32)
```

```
        self.init_finished = Tensor(np.zeros([batch_size, beam_width],
dtype=np.bool))
        self.init_length = Tensor(np.zeros([batch_size, beam_width],
dtype=np.int32))

        self.length_penalty = LengthPenalty(weight=length_penalty_weight)

        self.one = Tensor(1, mstype.int32)
        self.prob_concat = P.Concat(axis=1)
        self.cast = P.Cast()
        self.decoder_hidden_state = Tensor(np.zeros([self.decoder_layers_nums,
2,
                                                     self.batch_size *
self.beam_width,
                                                     hidden_size]),
mstype.float32)

        self.zeros_scores = Tensor(np.zeros([batch_size, beam_width],
dtype=np.float))
        self.active_index = Tensor(np.ones([batch_size, beam_width],
dtype=np.int32))
        self.init_zeros = Tensor(np.zeros([batch_size, beam_width],
dtype=np.int32))
        self.init_ones = Tensor(np.ones([batch_size, beam_width],
dtype=np.float32))

        self.accu_attn_scores = Tensor(np.zeros([batch_size, beam_width,
self.encoder_length], dtype=np.float32))

        self.zeros = Tensor([0], mstype.int32)
        self.eos_tensor = Tensor(np.full([batch_size, beam_width, beam_width],
eos_id), mstype.int32)

        self.ones_3d = Tensor(np.full([batch_size, beam_width,
self.encoder_length], 1), mstype.float32)
        self.neg_inf_3d = Tensor(np.full([batch_size, beam_width,
self.encoder_length], -INF), mstype.float32)
        self.zeros_3d = Tensor(np.full([batch_size, beam_width,
self.encoder_length], 0), mstype.float32)
        self.zeros_2d = Tensor(np.full([batch_size * beam_width,
self.encoder_length], 0), mstype.int32)
        self.argmin = P.ArgMinWithValue(axis=1)
        self.reducesum = P.ReduceSum()
        self.div = P.Div()
        self.shape_op = P.Shape()
        self.mul = P.Mul()
        self.log = P.Log()
        self.less = P.Less()
        self.tile = P.Tile()
        self.noteq = P.Neg()
        self.zeroslike = P.ZerosLike()
        self.greater_equal = P.GreaterEqual()
        self.sub = P.Sub()

    def one_step(self, cur_input_ids, enc_states, enc_attention_mask,
state_log_probs,
                 state_seq, state_length, idx=None, decoder_hidden_state=None,
accu_attn_scores=None,
```

```
                  state_finished=None):
        """
        Beam search one_step output.

        Inputs:
            cur_input_ids (Tensor):  with shape (batch_size * beam_width, 1).
            enc_states (Tensor):  with shape (batch_size * beam_width, T, D).
            enc_attention_mask (Tensor):  with shape (batch_size * beam_width,
T).
            state_log_probs (Tensor):  with shape (batch_size, beam_width).
            state_seq (Tensor):  with shape (batch_size, beam_width, m).
            state_length (Tensor):  with shape (batch_size, beam_width).
            idx (Tensor):  with shape ().
            decoder_hidden_state (Tensor): with shape (decoder_layer_num, 2,
batch_size * beam_width, D).
            accu_attn_scores (Tensor): with shape (batchsize, beam_width,
seq_length).
            state_finished (Tensor):  with shape (batch_size, beam_width).
        """

        # log_probs, [batch_size * beam_width, 1, V]
        log_probs, all_decoder_state, attn = self.decoder(cur_input_ids,
enc_states, enc_attention_mask,
                                                          decoder_hidden_state)
        # consider attention_scores
        attn = self.reshape(attn, (-1, self.beam_width, self.encoder_length))
        state_finished_attn = self.cast(state_finished, mstype.int32)
        attn_mask_0 = self.tile(self.expand(state_finished_attn, 2), (1, 1,
self.encoder_length))
        attn_mask_0 = self.cast(attn_mask_0, mstype.bool_)
        attn_new = self.select(attn_mask_0, self.zeros_3d, attn)
        accu_attn_scores = self.add(accu_attn_scores, attn_new)

        # log_probs: [batch_size, beam_width, V]
        log_probs = self.reshape(log_probs, (-1, self.beam_width,
self.vocab_size))
        # select topk indices, [batch_size, beam_width, V]
        total_log_probs = self.add(log_probs, self.expand(state_log_probs, -1))
        # mask finished beams, [batch_size, beam_width]
        # t-1 has finished
        mask_tensor = self.select(state_finished, self.ninf_tensor,
self.zero_tensor)
        # save the t-1 probability
        total_log_probs = self.add(total_log_probs, self.expand(mask_tensor,
-1))
        # [batch, beam*vocab]
        flat_scores = self.reshape(total_log_probs, (-1, self.beam_width *
self.vocab_size))
        # 选择top-k, [batch, beam]
        topk_scores, topk_indices = self.topk(flat_scores, self.beam_width)

        temp = topk_indices
        beam_indices = self.zeroslike(topk_indices)
        for _ in range(self.beam_width - 1):
            temp = self.sub(temp, self.vocab_size_tensor)
            res = self.cast(self.greater_equal(temp, 0), mstype.int32)
            beam_indices = beam_indices + res
        word_indices = topk_indices - beam_indices * self.vocab_size_tensor
```

```
        # ============================================================================

        # mask finished indices, [batch, beam]
        beam_indices = self.select(state_finished, self.beam_ids, beam_indices)
        word_indices = self.select(state_finished, self.eos_ids, word_indices)
        topk_scores = self.select(state_finished, state_log_probs, topk_scores)

        # sort according to scores with -inf for finished beams, [batch, beam]
        # t ends
        tmp_log_probs = self.select(
            self.equal(word_indices, self.eos_ids),
            self.ninf_tensor,
            topk_scores)

        _, tmp_indices = self.topk(tmp_log_probs, self.beam_width)
        # 更新, [batch_size, beam_width, 2]
        tmp_gather_indices = self.concat((self.expand(self.batch_ids, -1),
self.expand(tmp_indices, -1)))
        # [batch_size, beam_width]
        beam_indices = self.gather_nd(beam_indices, tmp_gather_indices)
        word_indices = self.gather_nd(word_indices, tmp_gather_indices)
        topk_scores = self.gather_nd(topk_scores, tmp_gather_indices)

        # 用于选择活动光束的聚集索引
        gather_indices = self.concat((self.expand(self.batch_ids, -1),
self.expand(beam_indices, -1)))

        # 如果在前一步中没有完成，长度加1, [batch_size, beam_width]
        length_add = self.add(state_length, self.one)
        state_length = self.select(state_finished, state_length, length_add)
        state_length = self.gather_nd(state_length, gather_indices)
        # 拼接 seq
        seq = self.gather_nd(state_seq, gather_indices)
        # 更新 accu_attn_scores
        accu_attn_scores = self.gather_nd(accu_attn_scores, gather_indices)
        # 更新 all_decoder_state
        all_decoder_state = self.reshape(all_decoder_state,
                                         (self.decoder_layers_nums * 2,
self.batch_size, self.beam_width,
                                          self.hidden_size))
        for i in range(self.decoder_layers_nums * 2):
            all_decoder_state[i, :, :, :] = self.gather_nd(all_decoder_state[i,
:, :, :], gather_indices)
        all_decoder_state = self.reshape(all_decoder_state,
                                         (self.decoder_layers_nums, 2,
self.batch_size * self.beam_width,
                                          self.hidden_size))

        # 更新 state_seq
        if self.is_using_while:
            state_seq_new = self.cast(seq, mstype.float32)
            word_indices_fp32 = self.cast(word_indices, mstype.float32)
            state_seq_new[:, :, idx] = word_indices_fp32
            state_seq = self.cast(state_seq_new, mstype.int32)
        else:
            state_seq = self.concat((seq, self.expand(word_indices, -1)))

        cur_input_ids = self.reshape(word_indices, (-1, 1))
```

```
        state_log_probs = topk_scores
        state_finished = self.equal(word_indices, self.eos_ids)

        return cur_input_ids, state_log_probs, state_seq, state_length, \
               all_decoder_state, accu_attn_scores, state_finished

    def construct(self, enc_states, enc_attention_mask):
        """
        Process source sentence

        Inputs:
            enc_states (Tensor): Output of transformer encoder with shape
(batch_size * beam_width, T, D).
            enc_attention_mask (Tensor): encoder attention mask with shape
(batch_size * beam_width, T).

        Returns:
            Tensor, predictions output.
        """
        # 开始beam search 算法
        cur_input_ids = self.start_ids
        state_log_probs = self.init_scores
        state_seq = self.init_seq
        state_finished = self.init_finished
        state_length = self.init_length
        decoder_hidden_state = self.decoder_hidden_state
        accu_attn_scores = self.accu_attn_scores

        if not self.is_using_while:
            for _ in range(self.max_decode_length):
                cur_input_ids, state_log_probs, state_seq, state_length,
decoder_hidden_state, accu_attn_scores, \
                state_finished = self.one_step(cur_input_ids, enc_states,
enc_attention_mask, state_log_probs,
                                               state_seq, state_length, None,
decoder_hidden_state, accu_attn_scores,
                                               state_finished)
        else:
            # At present, only ascend910 supports while operation.
            idx = self.start + 1
            ends = self.start + self.max_decode_length + 1
            while idx < ends:
                cur_input_ids, state_log_probs, state_seq, state_length,
decoder_hidden_state, accu_attn_scores, \
                state_finished = self.one_step(cur_input_ids, enc_states,
enc_attention_mask, state_log_probs,
                                               state_seq, state_length, idx,
decoder_hidden_state, accu_attn_scores,
                                               state_finished)
                idx = idx + 1

        # 添加长度惩罚系数
        penalty_len = self.length_penalty(state_length)
        log_probs = self.real_div(state_log_probs, penalty_len)
        penalty_cov = C.clip_by_value(accu_attn_scores, 0.0, 1.0)
        penalty_cov = self.log(penalty_cov)
        penalty_less = self.less(penalty_cov, self.neg_inf_3d)
        penalty = self.select(penalty_less, self.zeros_3d, penalty_cov)
```

```
        penalty = self.reducesum(penalty, 2)
        log_probs = log_probs + penalty * self.cov_penalty_factor
        # 根据惩罚系数排列
        _, top_beam_indices = self.topk(log_probs, self.beam_width)
        gather_indices = self.concat((self.expand(self.batch_ids, -1),
self.expand(top_beam_indices, -1)))
        # 按照序列顺序和Attention分数排列
        predicted_ids = self.gather_nd(state_seq, gather_indices)
        if not self.is_using_while:
            predicted_ids = predicted_ids[:, 0:1, 1:(self.max_decode_length +
1)]
        else:
            predicted_ids = predicted_ids[:, 0:1, :self.max_decode_length]

        return predicted_ids
```

### 3.4.7 GNMT

GNMT和通常的模型一样，拥有3个组成部分 -- 一个encoder，一个decoder，和一个attention network 。encoder将输入语句变成一系列的向量，每个向量代表原语句的一个词，decoder会使用这些向量以及其自身已经生成的词，生成下一个词。encoder和decoder通过attention network连接，这使得decoder可以在产生目标词时关注原语句的不同部分。

```
import copy

class GNMT(nn.Cell):
    """
    GNMT with encoder and decoder.

    In GNMT, we define T = src_max_len, T' = tgt_max_len.

    Args:
        config: Model config.
        is_training (bool): Whether is training.
        use_one_hot_embeddings (bool): Whether use one-hot embedding.

    Returns:
        Tuple[Tensor], network outputs.
    """

    def __init__(self,
                 config,
                 is_training: bool = False,
                 use_one_hot_embeddings: bool = False,
                 use_positional_embedding: bool = True,
                 compute_type=mstype.float32):
        super(GNMT, self).__init__()

        self.input_mask_from_dataset = config.input_mask_from_dataset
        self.max_positions = config.seq_length
        self.attn_embed_dim = config.hidden_size

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

```
            config.hidden_dropout_prob = 0.0
            config.attention_dropout_prob = 0.0
        self.is_training = is_training
        self.num_layers = config.num_hidden_layers
        self.hidden_dropout_prob = config.hidden_dropout_prob
        self.vocab_size = config.vocab_size
        self.seq_length = config.seq_length
        self.batch_size = config.batch_size
        self.max_decode_length = config.max_decode_length
        self.word_embed_dim = config.hidden_size

        self.beam_width = config.beam_width

        self.transpose = P.Transpose()
        self.transpose_orders = (1, 0, 2)
        self.embedding_lookup = EmbeddingLookup(
            is_training=self.is_training,
            vocab_size=self.vocab_size,
            embed_dim=self.word_embed_dim,
            use_one_hot_embeddings=use_one_hot_embeddings)

        self.gnmt_encoder = GNMTEncoder(config, is_training)

        if self.is_training:
            # use for train.
            self.gnmt_decoder = GNMTDecoder(config, is_training)
        else:
            # use for infer.
            self.expand = P.ExpandDims()
            self.multiply = P.Mul()
            self.reshape = P.Reshape()
            self.create_att_paddings_from_input_paddings =
CreateAttentionPaddingsFromInputPaddings(config,

                   is_training=False)
            self.tile_beam = TileBeam(beam_width=config.beam_width)
            self.cast_compute_type = SaturateCast(dst_type=compute_type)

            beam_decoder_cell = BeamDecoderStep(config,
use_one_hot_embeddings=use_one_hot_embeddings)
            # link beam_search after decoder
            self.beam_decoder = BeamSearchDecoder(
                batch_size=config.batch_size,
                seq_length=config.seq_length,
                vocab_size=config.vocab_size,
                decoder=beam_decoder_cell,
                beam_width=config.beam_width,
                decoder_layers_nums=config.num_hidden_layers,
                length_penalty_weight=config.length_penalty_weight,
                hidden_size=config.hidden_size,
                max_decode_length=config.max_decode_length)
            self.beam_decoder.add_flags(loop_can_unroll=True)
        self.shape = P.Shape()

    def construct(self, source_ids, source_mask=None, target_ids=None):
        """
        Construct network.
```

```
        In this method, T = src_max_len, T' = tgt_max_len.

        Args:
            source_ids (Tensor): Source sentences with shape (N, T).
            source_mask (Tensor): Source sentences padding mask with shape (N,
T),
                where 0 indicates padding position.
            target_ids (Tensor): Target sentences with shape (N, T').

        Returns:
            Tuple[Tensor], network outputs.
        """

        # Process source sentences. src_embeddings:[N, T, D].
        src_embeddings, _ = self.embedding_lookup(source_ids)
        # T, N, D
        inputs = self.transpose(src_embeddings, self.transpose_orders)
        # encoder. encoder_outputs: [T, N, D]
        encoder_outputs = self.gnmt_encoder(inputs)

        # decoder.
        if self.is_training:
            # training
            # process target input sentences. N, T, D
            tgt_embeddings, _ = self.embedding_lookup(target_ids)
            # T, N, D
            tgt_embeddings = self.transpose(tgt_embeddings,
self.transpose_orders)
            # cell: [T,N,D].
            cell, _, _ = self.gnmt_decoder(tgt_embeddings,
                                           encoder_outputs,
                                           attention_mask=source_mask)
            # decoder_output: (N, T', V).
            decoder_outputs = self.transpose(cell, self.transpose_orders)
            out = decoder_outputs
        else:
            # infer
            # encoder_output:  [T, N, D] -> [N, T, D].
            beam_encoder_output = self.transpose(encoder_outputs,
self.transpose_orders)
            # bean search for encoder output, [N*beam_width, T, D]
            beam_encoder_output = self.tile_beam(beam_encoder_output)

            # (N*beam_width, T)
            beam_enc_attention_pad = self.tile_beam(source_mask)

            predicted_ids = self.beam_decoder(beam_encoder_output,
beam_enc_attention_pad)
            predicted_ids = self.reshape(predicted_ids, (-1,
self.max_decode_length))
            out = predicted_ids

        return out
```

### 3.5 GNMTTraining

定义 `GNMTTraining` 类，该类将上述定义的 `GNMT` 模型和 `PredLogProbs` 模型封装，能够对 `GNMT` 模型进行训练并利用 `PredLogProbs` 类返回预测结果概率。

```
class GNMTTraining(nn.Cell):
    """
    GNMT training network.

    Args:
        config: The config of GNMT.
        is_training (bool): Specifies whether to use the training mode.
        use_one_hot_embeddings (bool): Specifies whether to use one-hot for
embeddings.

    Returns:
        Tensor, prediction_scores.
    """

    def __init__(self, config, is_training, use_one_hot_embeddings):
        super(GNMTTraining, self).__init__()
        self.gnmt = GNMT(config, is_training, use_one_hot_embeddings)
        self.projection = PredLogProbs(config)

    def construct(self, source_ids, source_mask, target_ids):
        """
        Construct network.

        Args:
            source_ids (Tensor): Source sentence.
            source_mask (Tensor): Source padding mask.
            target_ids (Tensor): Target sentence.

        Returns:
            Tensor, prediction_scores.
        """
        decoder_outputs = self.gnmt(source_ids, source_mask, target_ids)
        prediction_scores = self.projection(decoder_outputs)
        return prediction_scores
```

# 4. 损失函数与优化器

## 4.1 LabelSmoothedCrossEntropyCriterion

在将深度学习模型用于分类任务时，我们通常会遇到以下问题：过度拟合和过度自信。对过度拟合的研究非常深入，可以通过早期停止， 辍学，体重调整等方法解决。另一方面，我们缺乏解决过度自信的工具。标签平滑 是解决这两个问题的正则化技术。通过对 label 进行 weighted sum，能够取得比 one hot label 更好的效果。

label smoothing 将 label 由 $y_k$ 转化为 $y_k^{LS}$，公式为：

$$y_k^{LS} = y_k(1-\alpha) + \frac{\alpha}{K}$$

标签平滑用y_hot和均匀分布的混合替换一键编码的标签矢量y_hot：

y_ls = （1- α）* y_hot + α / K
其中K是标签类别的数量，而α是确定平滑量的超参数。如果α = 0，我们获得原始的一热编码y_hot。如果α = 1，我们得到均匀分布。

当损失函数是交叉熵时，使用标签平滑，模型将softmax函数应用于倒数第二层的对数向量z，以计算其输出概率p。在这种设置下，交叉熵损失函数相对于对数的梯度很简单

`∇CE= p - y = softmax（z）-y`

其中y是标签分布。特别是，我们可以看到

- 梯度下降将尝试使p尽可能接近y。
- 渐变范围介于-1和1。
  一键编码的标签鼓励将最大的logit间隙输入到softmax函数中。直观地，大的logit间隙与有限梯度相结合将使模型的适应性降低，并且对其预测过于自信。

```
class LabelSmoothedCrossEntropyCriterion(nn.Cell):
    """
    Label Smoothed Cross-Entropy Criterion.

    Args:
        config: The config of GNMT.

    Returns:
        Tensor, final loss.
    """

    def __init__(self, config):
        super(LabelSmoothedCrossEntropyCriterion, self).__init__()
        self.vocab_size = config.vocab_size
        self.batch_size = config.batch_size
        self.smoothing = 0.1
        self.confidence = 0.9
        self.last_idx = (-1,)
        self.reduce_sum = P.ReduceSum()
        self.reduce_mean = P.ReduceMean()
        self.reshape = P.Reshape()
        self.neg = P.Neg()
        self.cast = P.Cast()
        self.index_ids = Tensor(np.arange(config.batch_size *
config.max_decode_length).reshape((-1, 1)), mstype.int32)
        self.gather_nd = P.GatherNd()
        self.expand = P.ExpandDims()
        self.concat = P.Concat(axis=-1)

    def construct(self, prediction_scores, label_ids, label_weights):
        """
        Construct network to calculate loss.

        Args:
            prediction_scores (Tensor): Prediction scores. [batchsize, seq_len,
vocab_size]
            label_ids (Tensor): Labels. [batchsize, seq_len]
            label_weights (Tensor): Mask tensor. [batchsize, seq_len]

        Returns:
```

```
            Tensor, final loss.
        """
        prediction_scores = self.reshape(prediction_scores, (-1,
self.vocab_size))
        label_ids = self.reshape(label_ids, (-1, 1))
        label_weights = self.reshape(label_weights, (-1,))
        tmp_gather_indices = self.concat((self.index_ids, label_ids))
        nll_loss = self.neg(self.gather_nd(prediction_scores,
tmp_gather_indices))
        nll_loss = label_weights * nll_loss
        smooth_loss = self.neg(self.reduce_mean(prediction_scores,
self.last_idx))
        smooth_loss = label_weights * smooth_loss
        loss = self.reduce_sum(self.confidence * nll_loss + self.smoothing *
smooth_loss, ())
        loss = loss / self.batch_size
        return loss
```

## 4.2 优化器

这里利用Adam(Adaptive Moment Estimation，自适应矩估计)优化器自定义实现了AdamWeightDecacy类。

```
from mindspore._checkparam import Validator as validator
from mindspore.common.initializer import initializer
from mindspore.nn import Optimizer


adam_opt = C.MultitypeFuncGraph("adam_opt")

class Adam(Optimizer):
    def __init__(self, params, learning_rate=1e-3, beta1=0.9, beta2=0.999,
eps=1e-8, use_locking=False,
                 use_nesterov=False, weight_decay=0.0, loss_scale=1.0):
        super(Adam, self).__init__(learning_rate, params, weight_decay,
loss_scale)
        _check_param_value(beta1, beta2, eps, weight_decay, self.cls_name)
        validator.check_value_type("use_locking", use_locking, [bool],
self.cls_name)
        validator.check_value_type("use_nesterov", use_nesterov, [bool],
self.cls_name)
        validator.check_value_type("loss_scale", loss_scale, [float],
self.cls_name)

        self.beta1 = Tensor(beta1, mstype.float32)
        self.beta2 = Tensor(beta2, mstype.float32)
        self.beta1_power = Parameter(initializer(1, [1], mstype.float32))
        self.beta2_power = Parameter(initializer(1, [1], mstype.float32))
        self.eps = eps

        self.moment1 = self.parameters.clone(prefix="moment1", init='zeros')
        self.moment2 = self.parameters.clone(prefix="moment2", init='zeros')

        self.hyper_map = C.HyperMap()
        self.opt = P.Adam(use_locking, use_nesterov)
```

```
        self.pow = P.Pow()
        self.sqrt = P.Sqrt()
        self.one = Tensor(np.array([1.0]).astype(np.float32))
        self.realdiv = P.RealDiv()

        self.lr_scalar = P.ScalarSummary()

        self.exec_mode = context.get_context("mode")

    def construct(self, gradients):
        """Adam optimizer."""
        params = self.parameters
        moment1 = self.moment1
        moment2 = self.moment2
        gradients = self.decay_weight(gradients)
        gradients = self.scale_grad(gradients)
        lr = self.get_lr()

        #currently, Summary operators only support graph mode
        if self.exec_mode == context.GRAPH_MODE:
            self.lr_scalar("learning_rate", lr)

        beta1_power = self.beta1_power * self.beta1
        self.beta1_power = beta1_power
        beta2_power = self.beta2_power * self.beta2
        self.beta2_power = beta2_power
        if self.is_group_lr:
            success = self.hyper_map(F.partial(adam_opt, self.opt, beta1_power,
beta2_power, self.beta1,
                                               self.beta2, self.eps),
                                     lr, gradients, params, moment1, moment2)
        else:
            success = self.hyper_map(F.partial(adam_opt, self.opt, beta1_power,
beta2_power, self.beta1,
                                               self.beta2, self.eps, lr),
                                     gradients, params, moment1, moment2)
        return success

def _check_param_value(beta1, beta2, eps, weight_decay, prim_name):
    """Check the type of inputs."""
    validator.check_value_type("beta1", beta1, [float], prim_name)
    validator.check_value_type("beta2", beta2, [float], prim_name)
    validator.check_value_type("eps", eps, [float], prim_name)
    validator.check_value_type("weight_dacay", weight_decay, [float], prim_name)

def _get_optimizer(config, network, lr):
    """get gnmt optimizer, support Adam, Lamb, Momentum."""
    optimizer = Adam(network.trainable_params(), lr, beta1=0.9, beta2=0.98)
    return optimizer
```

## 4.3 学习率

模型初始训练时，模型的权重是随机初始化的，初始学习率太大，可能会导致模型的震荡（不稳定），选择学习率预热方法，在初始的几个epoch或者是steps内用一个比较小的学习率，训练完制定的epoch或者steps之后恢复设置的初始学习率

为了防止从较小学习率到指定的初始学习率变化较大引起误差增大。gradual warmup可以缓解这个问题，通过每个steps逐渐增大lr，知道达到指定的初始学习率后再开始学习率的下降。

```
import math

def Warmup_MultiStepLR_scheduler(base_lr=0.002, total_update_num=200,
warmup_steps=200, remain_steps=1.0,
                                 decay_interval=-1, decay_steps=4,
decay_factor=0.5):
    """
    Implements of polynomial decay learning rate scheduler which cycles by
default.

    Args:
        base_lr (float): Initial learning rate.
        total_update_num (int): Total update steps.
        warmup_steps (int or float): Warmup steps.
        remain_steps (int or float): start decay at 'remain_steps' iteration
        decay_interval (int): interval between LR decay steps
        decay_steps (int): Decay steps.
        decay_factor (float): decay factor

    Returns:
        np.ndarray, learning rate of each step.
    """

    if decay_steps <= 0:
        raise ValueError("`decay_steps` must larger than 1.")
    remain_steps = convert_float2int(remain_steps, total_update_num)
    warmup_steps = convert_float2int(warmup_steps, total_update_num)
    if warmup_steps > remain_steps:
        warmup_steps = remain_steps

    if decay_interval < 0:
        decay_iterations = total_update_num - remain_steps
        decay_interval = decay_iterations // decay_steps
        decay_interval = max(decay_interval, 1)
    else:
        decay_interval = convert_float2int(decay_interval, total_update_num)

    lrs = np.zeros(shape=total_update_num, dtype=np.float32)
    _start_step = 0
    for last_epoch in range(_start_step, total_update_num):
        if last_epoch < warmup_steps:
            if warmup_steps != 0:
                warmup_factor = math.exp(math.log(0.01) / warmup_steps)
            else:
                warmup_factor = 1.0
            inv_decay = warmup_factor ** (warmup_steps - last_epoch)
            lrs[last_epoch] = base_lr * inv_decay
        elif last_epoch >= remain_steps:
```

```
            decay_iter = last_epoch - remain_steps
            num_decay_step = decay_iter // decay_interval + 1
            num_decay_step = min(num_decay_step, decay_steps)
            lrs[last_epoch] = base_lr * (decay_factor ** num_decay_step)
        else:
            lrs[last_epoch] = base_lr
    return lrs

def convert_float2int(values, total_steps):
    if isinstance(values, float):
        values = int(values * total_steps)
    return values

def _get_lr(config, update_steps):
    """generate learning rate."""
    lr = Tensor(Warmup_MultiStepLR_scheduler(base_lr=config.lr,
                                             total_update_num=update_steps,

warmup_steps=config.warmup_steps,

remain_steps=config.warmup_lr_remain_steps,

decay_interval=config.warmup_lr_decay_interval,
                                                   decay_steps=config.decay_steps,

decay_factor=config.lr_scheduler_power), dtype=mstype.float32)
    return lr
```

# 5. 模型训练与保存

## 5.1 模型训练

首先定义模型训练函数，该函数加载需要训练的模型、模型配置参数、模型训练过程中的回调函数和数据集，判断是预训练还是微调后，使用`mindspore`的`Model`类开始训练。

```
def _train(model, config,
           pre_training_dataset=None, fine_tune_dataset=None, test_dataset=None,
           callbacks: list = None):
    """
    Train model.

    Args:
        model (Model): MindSpore model instance.
        config: Config of mass model.
        pre_training_dataset (Dataset): Pre-training dataset.
        fine_tune_dataset (Dataset): Fine-tune dataset.
        test_dataset (Dataset): Test dataset.
        callbacks (list): A list of callbacks.
    """
    callbacks = callbacks if callbacks else []

    if pre_training_dataset is not None:
        print(" | Start pre-training job.")
        epoch_size = pre_training_dataset.get_repeat_count()
        print("epoch size ", epoch_size)
```

```
        model.train(config.epochs, pre_training_dataset,
                    callbacks=callbacks,
dataset_sink_mode=config.dataset_sink_mode)

    if fine_tune_dataset is not None:
        print(" | Start fine-tuning job.")
        epoch_size = fine_tune_dataset.get_repeat_count()

        model.train(config.epochs, fine_tune_dataset,
                    callbacks=callbacks,
dataset_sink_mode=config.dataset_sink_mode)
```

## 5.2 GNMTNetworkWithLoss

为了更好的实现模型训练，我们这里将`GNMTTraining`训练对象和`LabelSmoothedCrossEntropyCriterion`损失函数对象封装在一个对象中，可以方便的计算模型训练过程中损失。

```
class GNMTNetworkWithLoss(nn.Cell):
    """
    Provide  GNMT training loss through network.

    Args:
        config (BertConfig): The config of GNMT.
        is_training (bool): Specifies whether to use the training mode.
        use_one_hot_embeddings (bool): Specifies whether to use one-hot for
embeddings. Default: False.

    Returns:
        Tensor, the loss of the network.
    """

    def __init__(self, config, is_training, use_one_hot_embeddings=False):
        super(GNMTNetworkWithLoss, self).__init__()
        self.gnmt = GNMTTraining(config, is_training, use_one_hot_embeddings)
        self.loss = LabelSmoothedCrossEntropyCriterion(config)
        self.cast = P.Cast()

    def construct(self,
                  source_ids,
                  source_mask,
                  target_ids,
                  label_ids,
                  label_weights):
        prediction_scores = self.gnmt(source_ids, source_mask, target_ids)
        total_loss = self.loss(prediction_scores, label_ids, label_weights)
        return self.cast(total_loss, mstype.float32)
```

## 5.3 GNMTTrainOneStepWithLossScaleCell

定义`GNMTTrainOneStepWithLossScaleCell`类型继承于`TrainOneStepWithLossScaleCell`，该类是可以使用混合精度功能的训练网络，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。

这里我们实现了对于GNMT模型的包含损失缩放（loss scale）的单次训练类，在实例化该对象之后，可以将优化器附加到训练网络函数来创建反向图。

```
from mindspore.nn.wrap.grad_reducer import DistributedGradReducer
from mindspore.communication.management import get_group_size

grad_scale = C.MultitypeFuncGraph("grad_scale")
reciprocal = P.Reciprocal()


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


_grad_overflow = C.MultitypeFuncGraph("_grad_overflow")
grad_overflow = P.FloatStatus()


@_grad_overflow.register("Tensor")
def _tensor_grad_overflow(grad):
    return grad_overflow(grad)


class GNMTTrainOneStepWithLossScaleCell(nn.TrainOneStepWithLossScaleCell):
    """
    Encapsulation class of GNMT network training.

    Append an optimizer to the training network after that the construct
    function can be called to create the backward graph.

    Args:
        network: Cell. The training network. Note that loss function should have
            been added.
        optimizer: Optimizer. Optimizer for updating the weights.

    Returns:
        Tuple[Tensor, Tensor, Tensor], loss, overflow, sen.
    """

    def __init__(self, network, optimizer, scale_update_cell=None):

        super(GNMTTrainOneStepWithLossScaleCell, self).__init__(network,
optimizer, scale_update_cell)
        self.cast = P.Cast()
        self.degree = 1
        if self.reducer_flag:
            self.degree = get_group_size()
            self.grad_reducer = DistributedGradReducer(optimizer.parameters,
False, self.degree)

        self.loss_scale = None
        self.loss_scaling_manager = scale_update_cell
        if scale_update_cell:
            self.loss_scale =
Parameter(Tensor(scale_update_cell.get_loss_scale(), dtype=mstype.float32))
```

```
    def construct(self,
                  source_eos_ids,
                  source_eos_mask,
                  target_sos_ids,
                  target_eos_ids,
                  target_eos_mask,
                  sens=None):
        """
        Construct network.

        Args:
            source_eos_ids (Tensor): Source sentence.
            source_eos_mask (Tensor): Source padding mask.
            target_sos_ids (Tensor): Target sentence.
            target_eos_ids (Tensor): Prediction sentence.
            target_eos_mask (Tensor): Prediction padding mask.
            sens (Tensor): Loss sen.

        Returns:
            Tuple[Tensor, Tensor, Tensor], loss, overflow, sen.
        """
        source_ids = source_eos_ids
        source_mask = source_eos_mask
        target_ids = target_sos_ids
        label_ids = target_eos_ids
        label_weights = target_eos_mask

        weights = self.weights
        loss = self.network(source_ids,
                            source_mask,
                            target_ids,
                            label_ids,
                            label_weights)
        if sens is None:
            scaling_sens = self.loss_scale
        else:
            scaling_sens = sens

        status, scaling_sens = self.start_overflow_check(loss, scaling_sens)

        grads = self.grad(self.network, weights)(source_ids,
                                                 source_mask,
                                                 target_ids,
                                                 label_ids,
                                                 label_weights,
                                                 self.cast(scaling_sens,
                                                           mstype.float32))
        # apply grad reducer on grads
        grads = self.grad_reducer(grads)
        grads = self.hyper_map(F.partial(grad_scale, scaling_sens *
self.degree), grads)
        grads = self.hyper_map(F.partial(clip_grad, GRADIENT_CLIP_TYPE,
GRADIENT_CLIP_VALUE), grads)

        cond = self.get_overflow_status(status, grads)
        overflow = cond
        if sens is None:
            overflow = self.loss_scaling_manager(self.loss_scale, cond)
```

```
        if not overflow:
            self.optimizer(grads)
        return (loss, cond, scaling_sens)
```

## 5.4 损失函数回调接口

定义`LossCallBack`类在每一轮训练完成时，将花费时间、当前轮次、当前step、当前模型的loss等信息输出到日志文件中。

```
import time
from mindspore.train.callback import Callback

class LossCallBack(Callback):
    """
    Monitor the loss in training.

    If the loss is 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.
    """
    time_stamp_init = False
    time_stamp_first = 0

    def __init__(self, config, per_print_times: int = 1):
        super(LossCallBack, self).__init__()
        if not isinstance(per_print_times, int) or per_print_times < 0:
            raise ValueError("print_step must be int and >= 0.")
        self.config = config
        self._per_print_times = per_print_times

        if not self.time_stamp_init:
            self.time_stamp_first = self._get_ms_timestamp()
            self.time_stamp_init = True

    def step_end(self, run_context):
        """step end."""
        cb_params = run_context.original_args()
        file_name = "./loss.log"
        with open(file_name, "a+") as f:
            time_stamp_current = self._get_ms_timestamp()
            f.write("time: {}, epoch: {}, step: {}, outputs: [loss: {},
overflow: {}, loss scale value: {} ].\n".format(
                time_stamp_current - self.time_stamp_first,
                cb_params.cur_epoch_num,
                cb_params.cur_step_num,
                str(cb_params.net_outputs[0].asnumpy()),
                str(cb_params.net_outputs[1].asnumpy()),
                str(cb_params.net_outputs[2].asnumpy())
            ))

    @staticmethod
    def _get_ms_timestamp():
```

```
        t = time.time()
        return int(round(t * 1000))
```

## 5.5 模型训练保存配置

模型加载`config`文件中的各项参数，初始化`GNMTNetworkWithLoss`参数，将这个带有损失函数的模型加载到`GNMTTrainOneStepWithLossScaleCell`模型训练`Cell`中，随后将该模型训练`Cell`送入`Model`类中作为一个整体模型类。最后为其配置`CheckPoint`保存点和每一个`step`需要得一些回调函数即完成全部模型装配，调用`_train`方法即可开始训练。

```
from mindspore.train.model import Model
from mindspore.train.loss_scale_manager import DynamicLossScaleManager
from mindspore.train.callback import CheckpointConfig, ModelCheckpoint,
SummaryCollector, TimeMonitor

def _build_training_pipeline(config,
                             pre_training_dataset=None,
                             fine_tune_dataset=None,
                             test_dataset=None):
    """
    Build training pipeline.

    Args:
        config: Config of mass model.
        pre_training_dataset (Dataset): Pre-training dataset.
        fine_tune_dataset (Dataset): Fine-tune dataset.
        test_dataset (Dataset): Test dataset.
    """
    net_with_loss = GNMTNetworkWithLoss(config, is_training=True,
use_one_hot_embeddings=True)
    net_with_loss.init_parameters_data()
    # _load_checkpoint_to_net(config, net_with_loss)

    dataset = pre_training_dataset if pre_training_dataset is not None \
        else fine_tune_dataset

    if dataset is None:
        raise ValueError("pre-training dataset or fine-tuning dataset must be
provided one.")

    update_steps = config.epochs * dataset.get_dataset_size()

    lr = _get_lr(config, update_steps)
    optimizer = _get_optimizer(config, net_with_loss, lr)

    # Dynamic loss scale.
    scale_manager =
DynamicLossScaleManager(init_loss_scale=config.init_loss_scale,

 scale_factor=config.loss_scale_factor,
                                            scale_window=config.scale_window)
    net_with_grads = GNMTTrainOneStepWithLossScaleCell(
        network=net_with_loss, optimizer=optimizer,
        scale_update_cell=scale_manager.get_update_cell()
    )
```

```
    net_with_grads.set_train(True)
    model = Model(net_with_grads)
    loss_monitor = LossCallBack(config)
    dataset_size = dataset.get_dataset_size()
    time_cb = TimeMonitor(data_size=dataset_size)
    ckpt_config = CheckpointConfig(save_checkpoint_steps=config.save_ckpt_steps,
                                   keep_checkpoint_max=config.keep_ckpt_max)

    rank_size = os.getenv('RANK_SIZE')
    callbacks = [time_cb, loss_monitor]

    if rank_size is None or int(rank_size) == 1:
        ckpt_callback = ModelCheckpoint(
            prefix=config.ckpt_prefix,
            directory=os.path.join(config.ckpt_path,
'ckpt_{}'.format(config.device_id)),
            config=ckpt_config)
        callbacks.append(ckpt_callback)
        summary_callback = SummaryCollector(summary_dir="./summary",
collect_freq=50)
        callbacks.append(summary_callback)

    print(f" | ALL SET, PREPARE TO TRAIN.")
    _train(model=model, config=config,
           pre_training_dataset=pre_training_dataset,
           fine_tune_dataset=fine_tune_dataset,
           test_dataset=test_dataset,
           callbacks=callbacks)
```

## 5.6 开始训练

将配置文件和数据集加载进入context，接着将数据集划分为训练数据集和测试数据集。准备好相关数据后即可开始启动训练。

```
from mindspore.common import set_seed

def do_train():
    config = get_config(default_config)
    config.pre_train_dataset = default_config.pre_train_dataset

    context.set_context(mode=context.GRAPH_MODE, save_graphs=False,
device_target=config.device_target,
                        reserve_class_name_in_scope=True,
device_id=config.device_id)
    _rank_size = os.getenv('RANK_SIZE')
    set_seed(config.random_seed)

    print(" | Starting training on single device.")
    pre_train_dataset = load_dataset(data_files=config.pre_train_dataset,
                                     batch_size=config.batch_size,
                                     sink_mode=config.dataset_sink_mode) if
config.pre_train_dataset else None
    fine_tune_dataset = load_dataset(data_files=config.fine_tune_dataset,
                                     batch_size=config.batch_size,
                                     sink_mode=config.dataset_sink_mode) if
config.fine_tune_dataset else None
    test_dataset = load_dataset(data_files=config.test_dataset,
```

```
                                batch_size=config.batch_size,
                                sink_mode=config.dataset_sink_mode) if
config.test_dataset else None

    _build_training_pipeline(config=config,
                             pre_training_dataset=pre_train_dataset,
                             fine_tune_dataset=fine_tune_dataset,
                             test_dataset=test_dataset)

do_train()
```

# 6. 模型加载与评估

## 6.1 模型加载

首先定义模型加载方法，如果存在之前训练得到的ckpt格式模型参数文件，那么就从该模型文件中加载权重，如果是Parameter类型，是模型训练的`weights`参数，需要调用data属性在通过`set_data()`方法直接载入；如果是Tensor类型，需要给先将参数转成numpy对应类型，然后在转换成配置文件中的`dtype`类型，在调用`set_data()`方法装配；如果是numpy的`ndarray`数组类型，也需要转换成配置文件中的`dtype`类型，在利用set_data()装配。如果不存在训练好的ckpt模型参数文件，对于gamma参数会产生一个全1矩阵，而其他参数则是通过`random.uniform`随机产生。

```
from mindspore.train.serialization import load_checkpoint

def _load_checkpoint_to_net(config, network):
    """load parameters to network from checkpoint."""
    if config.existed_ckpt:
        if config.existed_ckpt.endswith(".npz"):
            weights = np.load(config.existed_ckpt)
        else:
            weights = load_checkpoint(config.existed_ckpt)
        for param in network.trainable_params():
            weights_name = param.name
            if weights_name not in weights:
                raise ValueError(f"Param {weights_name} is not found in ckpt
file.")

            if isinstance(weights[weights_name], Parameter):
                param.set_data(weights[weights_name].data)
            elif isinstance(weights[weights_name], Tensor):
                param.set_data(Tensor(weights[weights_name].asnumpy(),
config.dtype))
            elif isinstance(weights[weights_name], np.ndarray):
                param.set_data(Tensor(weights[weights_name], config.dtype))
            else:
                param.set_data(weights[weights_name])
    else:
        for param in network.trainable_params():
            name = param.name
            value = param.data
            if isinstance(value, Tensor):
                if name.endswith(".gamma"):
                    param.set_data(one_weight(value.asnumpy().shape))
                elif name.endswith(".beta") or name.endswith(".bias"):
                    if param.data.dtype == "Float32":
```

```
 param.set_data((weight_variable(value.asnumpy().shape).astype(np.float32)))
                    elif param.data.dtype == "Float16":

 param.set_data((weight_variable(value.asnumpy().shape).astype(np.float16)))
                else:
                    if param.data.dtype == "Float32":

 param.set_data(Tensor(weight_variable(value.asnumpy().shape).astype(np.float32)
))
                    elif param.data.dtype == "Float16":

 param.set_data(Tensor(weight_variable(value.asnumpy().shape).astype(np.float16)
))

def weight_variable(shape):
    """
    Generate weight var.

    Args:
        shape (tuple): Shape.

    Returns:
        Tensor, var.
    """
    limit = 0.1
    values = np.random.uniform(-limit, limit, shape)
    return values


def one_weight(shape):
    """
    Generate weight with ones.

    Args:
        shape (tuple): Shape.

    Returns:
        Tensor, var.
    """
    ones = np.ones(shape).astype(np.float32)
    return Tensor(ones)
```

## 6.2 推理计算

### 6.2.1 推理权重加载

模型推理权重与模型训练权重在参数上并不一致，因此我们单独定义加载推理权重方法`load_infer_weights()`方法。如果是numpy的权重文件，我们可以直接将其加载进模型参数；如果是ckpt的Tensor参数，我们需要先将其转换为numpy，在从numpy中加载配置文件中要求的Tensor向量。

```
def load_infer_weights(config):
    """
    Load weights from ckpt or npz.
```

```
    Args:
        config: Config.

    Returns:
        dict, weights.
    """
    model_path = config.existed_ckpt
    if model_path.endswith(".npz"):
        ms_ckpt = np.load(model_path)
        is_npz = True
    else:
        ms_ckpt = load_checkpoint(model_path)
        is_npz = False
    weights = {}
    for param_name in ms_ckpt:
        infer_name = param_name.replace("gnmt.gnmt.", "")
        if infer_name.startswith("embedding_lookup."):
            if is_npz:
                weights[infer_name] = ms_ckpt[param_name]
            else:
                weights[infer_name] = ms_ckpt[param_name].data.asnumpy()
            infer_name = "beam_decoder.decoder." + infer_name
            if is_npz:
                weights[infer_name] = ms_ckpt[param_name]
            else:
                weights[infer_name] = ms_ckpt[param_name].data.asnumpy()
            continue
        elif not infer_name.startswith("gnmt_encoder"):
            if infer_name.startswith("gnmt_decoder."):
                infer_name = infer_name.replace("gnmt_decoder.", "decoder.")
            infer_name = "beam_decoder.decoder." + infer_name

        if is_npz:
            weights[infer_name] = ms_ckpt[param_name]
        else:
            weights[infer_name] = ms_ckpt[param_name].data.asnumpy()
    return weights
```

### 6.2.2 GNMTInferCell推理类

将`GNMT`模型的推理封装进`GNMTInferCell`推理类，使用该类能够装配`Model`后直接开始推理。

```
class GNMTInferCell(nn.Cell):
    """
    Encapsulation class of GNMT network infer.

    Args:
        network (nn.Cell): GNMT model.

    Returns:
        Tuple[Tensor, Tensor], predicted_ids and predicted_probs.
    """

    def __init__(self, network):
        super(GNMTInferCell, self).__init__(auto_prefix=False)
        self.network = network
```

```
    def construct(self,
                  source_ids,
                  source_mask):
        """Defines the computation performed."""

        predicted_ids = self.network(source_ids,
                                     source_mask)

        return predicted_ids
```

### 6.2.3 gnmt_infer推理方法

首先定义`gnmt_infer()`方法来配置模型推理所需要的各种参数。

1. 首先实例化GNMT模型，并将之前训练好保存的模型权重和偏置等参数加载到实例化模型中。
2. 使用`dataset.create_dict_iterator()`从数据集中创建训练`batch`。值得注意的是，如果最后一轮`batch`划分的数据不够`batch_size`，这里会为其填充无意义的`source_ids_pad`和无意义的`source_mask_pad`。
3. 模型推理完成后，将其组成{"source": '', "prediction": ''}的字典列表返回到`infer()`方法。

```
import time

def infer(config):
    """
    GNMT infer api.

    Args:
        config: Config.

    Returns:
        list, result with
    """
    eval_dataset = load_dataset(data_files=config.test_dataset,
                                batch_size=config.batch_size,
                                sink_mode=config.dataset_sink_mode,
                                drop_remainder=False,
                                is_translate=True,
                                shuffle=False) if config.test_dataset else None
    prediction = gnmt_infer(config, eval_dataset)
    return prediction

def gnmt_infer(config, dataset):
    """
    Run infer with GNMT.

    Args:
        config: Config.
        dataset (Dataset): Dataset.

    Returns:
        List[Dict], prediction, each example has 4 keys, "source",
        "target", "prediction" and "prediction_prob".
    """
    tfm_model = GNMT(config=config,
                     is_training=False,
                     use_one_hot_embeddings=False)
```

```
    params = tfm_model.trainable_params()
    weights = load_infer_weights(config)
    for param in params:
        value = param.data
        weights_name = param.name
        if weights_name not in weights:
            raise ValueError(f"{weights_name} is not found in weights.")
        if isinstance(value, Tensor):
            if weights_name in weights:
                assert weights_name in weights
                if isinstance(weights[weights_name], Parameter):
                    if param.data.dtype == "Float32":

 param.set_data(Tensor(weights[weights_name].data.asnumpy(), mstype.float32))
                    elif param.data.dtype == "Float16":

 param.set_data(Tensor(weights[weights_name].data.asnumpy(), mstype.float16))

                elif isinstance(weights[weights_name], Tensor):
                    param.set_data(Tensor(weights[weights_name].asnumpy(),
config.dtype))
                elif isinstance(weights[weights_name], np.ndarray):
                    param.set_data(Tensor(weights[weights_name], config.dtype))
                else:
                    param.set_data(weights[weights_name])
            else:
                print("weight not found in checkpoint: " + weights_name)
                param.set_data(zero_weight(value.asnumpy().shape))

    print(" | Load weights successfully.")
    tfm_infer = GNMTInferCell(tfm_model)
    model = Model(tfm_infer)

    predictions = []
    source_sentences = []

    shape = P.Shape()
    concat = P.Concat(axis=0)
    batch_index = 1
    pad_idx = 0
    sos_idx = 2
    eos_idx = 3
    source_ids_pad = Tensor(np.tile(np.array([[sos_idx, eos_idx] + [pad_idx] *
(config.seq_length - 2)]),
                                    [config.batch_size, 1]), mstype.int32)
    source_mask_pad = Tensor(np.tile(np.array([[1, 1] + [0] * (config.seq_length
- 2)]),
                                     [config.batch_size, 1]), mstype.int32)
    for batch in dataset.create_dict_iterator():
        source_sentences.append(batch["source_eos_ids"].asnumpy())
        source_ids = Tensor(batch["source_eos_ids"], mstype.int32)
        source_mask = Tensor(batch["source_eos_mask"], mstype.int32)

        active_num = shape(source_ids)[0]
        if active_num < config.batch_size:
            source_ids = concat((source_ids, source_ids_pad[active_num:, :]))
            source_mask = concat((source_mask, source_mask_pad[active_num:, :]))
```

```
        start_time = time.time()
        predicted_ids = model.predict(source_ids, source_mask)

        print(f" | BatchIndex = {batch_index}, Batch size: {config.batch_size},
active_num={active_num}, "
              f"Time cost: {time.time() - start_time}.")
        if active_num < config.batch_size:
            predicted_ids = predicted_ids[:active_num, :]
        batch_index = batch_index + 1
        predictions.append(predicted_ids.asnumpy())

    output = []
    for inputs, batch_out in zip(source_sentences, predictions):
        for i, _ in enumerate(batch_out):
            if batch_out.ndim == 3:
                batch_out = batch_out[:, 0]

            example = {
                "source": inputs[i].tolist(),
                "prediction": batch_out[i].tolist()
            }
            output.append(example)

    return output

def zero_weight(shape):
    """
    Generate weight with zeros.

    Args:
        shape (tuple): Shape.

    Returns:
        Tensor, var.
    """
    zeros = np.zeros(shape).astype(np.float32)
    return Tensor(zeros)
```

## 6.3 衡量指标

这里我们还是使用BLEU评价指标来衡量我们的模型性能。首先加载进来的文本数据使用`Tokenizer`分词，随后将分词后数据使用sacrebleu快速计算得到`BLEU`分数。

```
import subprocess

def load_result_data(result_npy_addr):
    # load the numpy to list.
    result = np.load(result_npy_addr, allow_pickle=True)
    return result

def get_bleu_data(tokenizer: Tokenizer, result_npy_addr):
    """
    Detokenizer the prediction.

    Args:
        tokenizer (Tokenizer): tokenizer operations.
```

```
        result_npy_addr (string): Path to the predict file.

    Returns:
        List, the predict text context.
    """

    result = load_result_data(result_npy_addr)
    prediction_list = []
    for _, info in enumerate(result):
        # prediction detokenize
        prediction = info["prediction"]
        prediction_str = tokenizer.detokenize(prediction)
        prediction_list.append(prediction_str)

    return prediction_list


def calculate_sacrebleu(predict_path, target_path):
    """
    Calculate the BLEU scores.

    Args:
        predict_path (string): Path to the predict file.
        target_path (string): Path to the target file.

    Returns:
        Float32, bleu scores.
    """

    sacrebleu_params = '--score-only -lc --tokenize intl'
    sacrebleu = subprocess.run([f'sacrebleu --input {predict_path} \
                                {target_path} {sacrebleu_params}'],
                               stdout=subprocess.PIPE, shell=True)
    bleu_scores = round(float(sacrebleu.stdout.strip()), 2)
    return bleu_scores


def bleu_calculate(tokenizer, result_npy_addr, target_addr=None):
    """
    Calculate the BLEU scores.

    Args:
        tokenizer (Tokenizer): tokenizer operations.
        result_npy_addr (string): Path to the predict file.
        target_addr (string): Path to the target file.

    Returns:
        Float32, bleu scores.
    """

    prediction = get_bleu_data(tokenizer, result_npy_addr)
    print("predict:\n", prediction)

    eval_path = './predict.txt'
    with open(eval_path, 'w') as eval_file:
        lines = [line + '\n' for line in prediction]
        eval_file.writelines(lines)
    reference_path = target_addr
```

```
    bleu_scores = calculate_sacrebleu(eval_path, reference_path)
    return bleu_scores
```

### 6.4 开始推理

推理过程比较简单，加载配置参数即可开始推理。将推理输出数据从`infer()`函数中取出后，将推理输出数据放入上述实现的`bleu_calculate()`中即可得到推理结果分数。

```
import pickle

def run_eval():
    '''run eval.'''
    _config = get_config(default_config)
    result = infer(_config)
    context.set_context(
        mode=context.GRAPH_MODE,
        save_graphs=False,
        device_target=_config.device_target,
        device_id=_config.device_id,
        reserve_class_name_in_scope=False)

    with open(_config.output, "wb") as f:
        pickle.dump(result, f, 1)

    result_npy_addr = _config.output
    vocab = _config.vocab
    bpe_codes = _config.bpe_codes
    test_tgt = _config.test_tgt
    tokenizer = Tokenizer(vocab, bpe_codes, 'en', 'de')
    scores = bleu_calculate(tokenizer, result_npy_addr, test_tgt)
    print(f"BLEU scores is :{scores}")
```

## 7. 总结

本文基于MindSpore框架，从数据集构建、模型构建、训练和评估等内容完整实现了GNMT V2模型，并根据机器翻译模型的特定评价指标，实现了BULE指标的计算，最后利用wmt16英德双语数据集完成模型训练，使用NEWS16英语数据集完成模型评估。通过此模型案例，能够进一步加深对机器翻译领域、GNMT模型的理解，同时由于GNMT v2中的模型结构的Encoder部分与Google后续在2018年发布的BERT模型十分相似，因此该模型也能作为理解BERT模型的一个前置案例；更重要的是，通过使用MindSpore框架实现GNMT v2模型，更深层次的理解了MindSpore的运行原理和特点，对于后续在更多模型上探索MindSpore训练的可行性打下来夯实的基础，相信在MindSpore的不断进步下，会有更多更好的模型涌现。

## 8. 引用