用于预训练BERT的数据集

为了预训练BERT模型，需要以理想的格式生成数据集，以便于两个预训练任务：遮蔽语言模型和下一句预测。一方面，最初的BERT模型是在两个庞大的图书语料库和英语维基百科的合集上预训练的，这两个数据集十分的庞大。另一方面，现成的预训练BERT模型可能不适合医学等特定领域的应用。因此，在定制的数据集上对BERT进行预训练变得越来越流行。为了方便BERT预训练的演示，下面使用了较小的语料库WikiText-2。

与PTB数据集相比，WikiText-2（1）保留了原来的标点符号，适合于下一句预测；（2）保留了原来的大小写和数字；（3）大了一倍以上。

In [1]:

```
import os
import random
import torch
from d2l import torch as d2l
```

在WikiText-2数据集中，每行代表一个段落，其中在任意标点符号及其前面的词元之间插入空格。保留至少有两句话的段落。为了简单起见，仅使用句号作为分隔符来拆分句子。

In [2]:

```
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # 大写字母转换为小写字母
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

为预训练任务定义辅助函数

在下文中，首先为BERT的两个预训练任务实现辅助函数。这些辅助函数将在稍后将原始文本语料库转换为理想格式的数据集时调用，以预训练BERT。

生成下一句预测任务的数据

`_get_next_sentence` 函数生成二分类任务的训练样本。

In [3]:

```
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # paragraphs是三重列表的嵌套
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

下面的函数通过调用 _get_next_sentence 函数从输入 paragraph 生成用于下一句预测的训练样本。这里 paragraph 是句子列表，其中每个句子都是词元列表。自变量 max_len 指定预训练期间的BERT输入序列的最大长度。

In [4]:

```
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # 考虑1个'<cls>'词元和2个'<sep>'词元
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

生成遮蔽语言模型任务的数据

为了从BERT输入序列生成遮蔽语言模型的训练样本，定义了以下 _replace_mlm_tokens 函数。在其输入中， tokens 是表示BERT输入序列的词元的列表， candidate_pred_positions 是不包括特殊词元的BERT输入序列的词元索引的列表（特殊词元在遮蔽语言模型任务中不被预测），以及 num_mlm_preds 指示预测的数量（选择15%要预测的随机词元）。在每个预测位置，输入可以由特殊的“掩码”词元或随机词元替换，或者保持不变。最后，该函数返回可能替换后的输入词元、发生预测的词元索引和这些预测的标签。

In [5]:

```
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # 为遮蔽语言模型的输入创建新的词元副本，其中输入可能包含替换的“<mask>”或随机词元
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # 打乱后用于在遮蔽语言模型任务中获取15%的随机词元进行预测
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80%的时间：将词替换为“<mask>”词元
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10%的时间：保持词不变
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10%的时间：用随机词替换该词
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

通过调用前述的 _replace_mlm_tokens 函数，以下函数将BERT输入序列（ tokens ）作为输入，并返回输入词元的索引（在可能的词元替换之后）、发生预测的词元索引以及这些预测的标签索引。

In [6]:

```
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # tokens是一个字符串列表
    for i, token in enumerate(tokens):
        # 在遮蔽语言模型任务中不会预测特殊词元
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 遮蔽语言模型任务中预测15%的随机词元
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

将文本转换为预训练数据集

现在定义辅助函数 _pad_bert_inputs 来将特殊的“<mask>”词元附加到输入。它的参数 examples 包含来自两个预训练任务的辅助函数 _get_nsp_data_from_paragraph 和 _get_mlm_data_from_tokens 的输出。

In [7]:

```
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
        # valid_lens不包括'<pad>'的计数
        valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
        all_pred_positions.append(torch.tensor(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
        # 填充词元的预测将通过乘以0权重在损失中过滤掉
        all_mlm_weights.append(
            torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)),
                dtype=torch.float32))
        all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))
        nsp_labels.append(torch.tensor(is_next, dtype=torch.long))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

将用于生成两个预训练任务的训练样本的辅助函数和用于填充输入的辅助函数放在一起，定义以下 `_WikiTextDataset` 类为用于预训练BERT的WikiText-2数据集。通过实现 `__getitem__` 函数，来任意访问WikiText-2语料库的一对句子生成的预训练样本（遮蔽语言模型和下一句预测）样本。

最初的BERT模型使用词表大小为30000的WordPiece嵌入。WordPiece的词元化方法是对字节对编码算法稍作修改。为简单起见，下面使用 `d2l.tokenize` 函数进行词元化。出现次数少于5次的不频繁词元将被过滤掉。

In [8]:

```
class _WikiTextDataset(torch.utils.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # 输入paragraphs[i]是代表段落的句子字符串列表；
        # 而输出paragraphs[i]是代表段落的句子列表，其中每个句子都是词元列表
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # 获取下一句子预测任务的数据
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # 获取遮蔽语言模型任务的数据
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                     + (segments, is_next))
                    for tokens, segments, is_next in examples]
        # 填充输入
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

通过使用 _read_wiki 函数和 _WikiTextDataset 类，定义下面的 load_data_wiki 来下载并生成WikiText-2数据集，并从中生成预训练样本。

In [9]:

```
def load_data_wiki(batch_size, max_len):
    """加载WikiText-2数据集"""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                        shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab
```

将批量大小设置为512，将BERT输入序列的最大长度设置为64，我们打印出小批量的BERT预训练样本的形状。注意，在每个BERT输入序列中，为遮蔽语言模型任务预测10（64×0.15）个位置。

In [10]:

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

```
Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.metamin
d.io/wikitext/wikitext-2-v1.zip... (https://s3.amazonaws.com/research.metamind.io/wi
kitext/wikitext-2-v1.zip...)
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512, 10])
 torch.Size([512, 10]) torch.Size([512, 10]) torch.Size([512])
```

最后看一下词量。即使在过滤掉不频繁的词元之后，它仍然比PTB数据集的大两倍以上。

In [11]:

```
len(vocab)
```

Out[11]:

```
20256
```