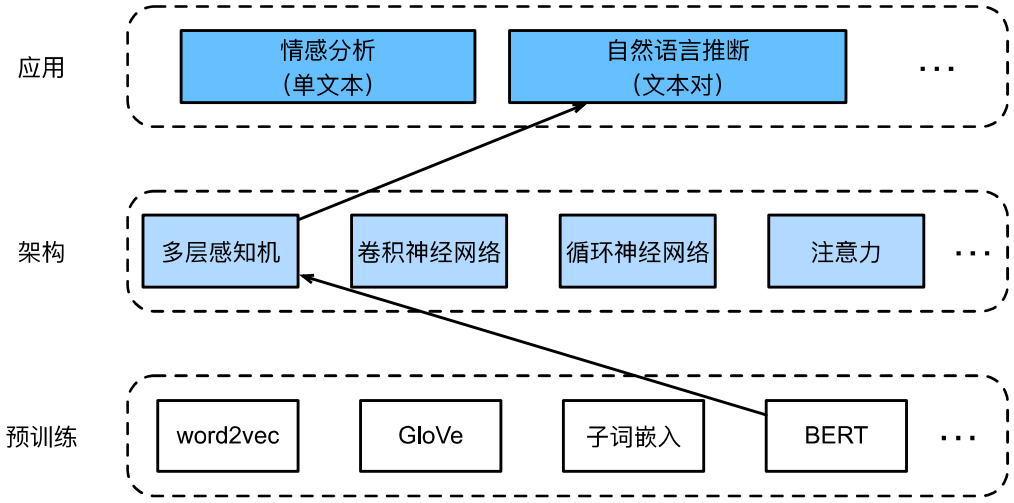
自然语言推断：微调BERT

现在，我们通过微调BERT来实现自然语言推断任务。自然语言推断是一个序列级别的文本对分类问题，而微调BERT只需要一个额外的基于多层感知机的架构，如下图所示。



下面将下载一个预训练好的小版本的BERT，然后对其进行微调，以便在SNLI数据集上进行自然语言推断。

In [1]:

```
import json
import multiprocessing
import os
import torch
from torch import nn
from d2l import torch as d2l
```

加载预训练的BERT

原始的BERT模型有数以亿计的参数。在下面，d2l提供了两个版本的预训练的BERT：“bert.base”与原始的BERT基础模型一样大，需要大量的计算资源才能进行微调，而“bert.small”是一个小版本，以便于演示。

In [2]:

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                              'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

两个预训练好的BERT模型都包含一个定义词表的“vocab.json”文件和一个预训练参数的“pretrained.params”文件。 `load_pretrained_model` 函数用来**加载预先训练好的BERT参数。**

In [3]:

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_layers, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # 定义空词表以加载预定义词表
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir,
        'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(len(vocab), num_hiddens, norm_shape=[256],
                         ffn_num_input=256, ffn_num_hiddens=ffn_num_hiddens,
                         num_heads=4, num_layers=2, dropout=0.2,
                         max_len=max_len, key_size=256, query_size=256,
                         value_size=256, hid_in_features=256,
                         mlm_in_features=256, nsp_in_features=256)
    # 加载预训练BERT参数
    bert.load_state_dict(torch.load(os.path.join(data_dir,
                                                 'pretrained.params')))
    return bert, vocab
```

为了便于在大多数机器上演示，之后将加载和微调经过预训练BERT的小版本（“bert.mall”）。

In [4]:

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_layers=2, dropout=0.1, max_len=512, devices=devices)
```

```
Downloading ../data/bert.small.torch.zip from http://d2l-data.s3-accelerate.amazonaw
s.com/bert.small.torch.zip... (http://d2l-data.s3-accelerate.amazonaws.com/bert.smal
l.torch.zip...)
```

微调BERT的数据集

对于SNLI数据集的下游任务自然语言推断，下面定义了一个定制的数据集类 SNLIBERTDataset 。在每个样本中，前提和假设形成一对文本序列，并被打包成一个BERT输入序列。片段索引用于区分BERT输入序列中的前提和假设。利用预定义的BERT输入序列的最大长度（ max_len ），持续移除输入文本对中较长文本的最后一个标记，直到满足 max_len 。为了加速生成用于微调BERT的SNLI数据集，使用4个工作进程并行生成训练或测试样本。

In [5]:

```
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]

        self.labels = torch.tensor(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # 使用4个进程
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (torch.tensor(all_token_ids, dtype=torch.long),
                torch.tensor(all_segments, dtype=torch.long),
                torch.tensor(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                              * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # 为BERT输入中的'<CLS>'、'<SEP>'和'<SEP>'词元保留位置
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

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

下载完SNLI数据集后，通过实例化 SNLIBERTDataset 类来**生成训练和测试样本**。这些样本将在自然语言推断的训练和测试期间进行小批量读取。

In [6]:

```
# 如果出现显存不足错误，请减少“batch_size”。在原始的BERT模型中，max_len=512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                         num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                        num_workers=num_workers)
```

```
Downloading ../data/snli_1.0.zip from https://nlp.stanford.edu/projects/snli/snli_1.
0.zip... (https://nlp.stanford.edu/projects/snli/snli_1.0.zip...)
read 549367 examples
read 9824 examples
```

微调BERT

用于自然语言推断的微调BERT只需要一个额外的多层感知机，该多层感知机由两个全连接层组成（请参见下面 BERTClassifier 类中的 self.hidden 和 self.output ）。**这个多层感知机将特殊的“<cls>”词元**的BERT表示进行了转换，该词元同时编码前提和假设的信息(**为自然语言推断的三个输出**)：蕴涵、矛盾和中性。

In [7]:

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Linear(256, 3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

预训练的BERT模型 bert 被送到用于下游应用的 BERTClassifier 实例 net 中。在BERT微调的常见实现中，只有额外的多层感知机（ net.output ）的输出层的参数将从零开始学习。预训练BERT编码器（ net.encoder ）和额外的多层感知机的隐藏层（ net.hidden ）的所有参数都将进行微调。

In [8]:

```
net = BERTClassifier(bert)
```

当BERT微调时， MaskLM 和 NextSentencePred 中采用的多层感知机的参数不会更新（陈旧的，staled），因为这些参数仅用于计算预训练过程中的遮蔽语言模型损失和下一句预测损失。

为了允许具有陈旧梯度的参数，标志 ignore_stale_grad=True 在 step 函数 d2l.train_batch_ch13 中被设置。通过该函数使用SNLI的训练集（ train_iter ）和测试集（ test_iter ）对 net 模型进行训练和评估。

In [9]:

```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
    devices)
```

```
loss 0.518, train acc 0.791, test acc 0.780
6357.0 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1),
device(type='cuda', index=2), device(type='cuda', index=3)]
```