10.12 机器翻译

机器翻译是指将一段文本从一种语言自动翻译到另一种语言。因为一段文本序列在不同语言中的长度不一定相同，所以我们使用机器翻译为例来介绍编码器—解码器和注意力机制的应用。

10.12.1 读取和预处理数据

我们先定义一些特殊符号。其中“<pad>”（padding）符号用来添加在较短序列后，直到每个序列等长，而“<bos>”和“<eos>”符号分别表示序列的开始和结束。

```
import collections
import os
import io
import math
import torch
from torch import nn
import torch.nn.functional as F
import torchtext.vocab as Vocab
import torch.utils.data as Data

import sys
sys.path.append("..")
import d2lzh_pytorch as d2l

PAD, BOS, EOS = '<pad>', '<bos>', '<eos>'
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

接着定义两个辅助函数对后面读取的数据进行预处理。

```
# 将一个序列中所有的词记录在all_tokens中以便之后构造词典，然后在该序列后面添加PAD直到序列
# 长度变为max_seq_len，然后将序列保存在all_seqs中
def process_one_seq(seq_tokens, all_tokens, all_seqs, max_seq_len):
    all_tokens.extend(seq_tokens)
    seq_tokens += [EOS] + [PAD] * (max_seq_len - len(seq_tokens) - 1)
    all_seqs.append(seq_tokens)

# 使用所有的词来构造词典。并将所有序列中的词变换为词索引后构造Tensor
```

```
def build_data(all_tokens, all_seqs):
    vocab = Vocab.Vocab(collections.Counter(all_tokens),
                        specials=[PAD, BOS, EOS])
    indices = [[vocab.stoi[w] for w in seq] for seq in all_seqs]
    return vocab, torch.tensor(indices)
```

为了演示方便，我们在这里使用一个很小的法语—英语数据集。在这个数据集里，每一行是一对法语句子和它对应的英语句子，中间使用 `'\t'` 隔开。在读取数据时，我们在句末附上“<eos>”符号，并可能通过添加“<pad>”符号使每个序列的长度均为 `max_seq_len` 。我们为法语词和英语词分别创建词典。法语词的索引和英语词的索引相互独立。

```
def read_data(max_seq_len):
    # in和out分别是input和output的缩写
    in_tokens, out_tokens, in_seqs, out_seqs = [], [], [], []
    with io.open('../../data/fr-en-small.txt') as f:
        lines = f.readlines()
    for line in lines:
        in_seq, out_seq = line.rstrip().split('\t')
        in_seq_tokens, out_seq_tokens = in_seq.split(' '), out_seq.split(' ')
        if max(len(in_seq_tokens), len(out_seq_tokens)) > max_seq_len - 1:
            continue  # 如果加上EOS后长于max_seq_len，则忽略掉此样本
        process_one_seq(in_seq_tokens, in_tokens, in_seqs, max_seq_len)
        process_one_seq(out_seq_tokens, out_tokens, out_seqs, max_seq_len)
    in_vocab, in_data = build_data(in_tokens, in_seqs)
    out_vocab, out_data = build_data(out_tokens, out_seqs)
    return in_vocab, out_vocab, Data.TensorDataset(in_data, out_data)
```

将序列的最大长度设成7，然后查看读取到的第一个样本。该样本分别包含法语词索引序列和英语词索引序列。

```
max_seq_len = 7
in_vocab, out_vocab, dataset = read_data(max_seq_len)
dataset[0]
```

输出：

```
(tensor([ 5,  4, 45,  3,  2,  0,  0]), tensor([ 8,  4, 27,  3,  2,  0,  0]))
```

10.12.2 含注意力机制的编码器—解码器

我们将使用含注意力机制的编码器—解码器来将一段简短的法语翻译成英语。下面我们来介绍模型的实现。

10.12.2.1 编码器

在编码器中，我们将输入语言的词索引通过词嵌入层得到词的表征，然后输入到一个多层门控循环单元中。正如我们在6.5节（循环神经网络的简洁实现）中提到的，PyTorch的 `nn.GRU` 实例在前向计算后也会分别返回输出和最终时间步的多层隐藏状态。其中的输出指的是最后一层的隐藏层在各个时间步的隐藏状态，并不涉及输出层计算。注意力机制将这些输出作为键项和值项。

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 drop_prob=0, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers, dropout=drop_prob)

    def forward(self, inputs, state):
        # 输入形状是(批量大小, 时间步数)。将输出互换样本维和时间步维
        embedding = self.embedding(inputs.long()).permute(1, 0, 2) # (seq_len, batch,
        return self.rnn(embedding, state)

    def begin_state(self):
        return None # 隐藏态初始化为None时PyTorch会自动初始化为0
```

下面我们来创建一个批量大小为4、时间步数为7的小批量序列输入。设门控循环单元的隐藏层个数为2，隐藏单元个数为16。编码器对该输入执行前向计算后返回的输出形状为(时间步数, 批量大小, 隐藏单元个数)。门控循环单元在最终时间步的多层隐藏状态的形状为(隐藏层个数, 批量大小, 隐藏单元个数)。对于门控循环单元来说， `state` 就是一个元素，即隐藏状态；如果使用长短期记忆， `state` 是一个元组，包含两个元素即隐藏状态和记忆细胞。

```
encoder = Encoder(vocab_size=10, embed_size=8, num_hiddens=16, num_layers=2)
output, state = encoder(torch.zeros((4, 7)), encoder.begin_state())
```

```
output.shape, state.shape # GRU的state是h, 而LSTM的是一个元组(h, c)
```

输出：

```
(torch.Size([7, 4, 16]), torch.Size([2, 4, 16]))
```

10.12.2.2 注意力机制

我们将实现10.11节（注意力机制）中定义的函数a：将输入连结后通过含单隐藏层的多层感知机变换。其中隐藏层的输入是解码器的隐藏状态与编码器在所有时间步上隐藏状态的一一连结，且使用tanh函数作为激活函数。输出层的输出个数为1。两个 `Linear` 实例均不使用偏差。其中函数a定义里向量$\boldsymbol{v}$的长度是一个超参数，即 `attention_size` 。

```
def attention_model(input_size, attention_size):
    model = nn.Sequential(nn.Linear(input_size,
                                    attention_size, bias=False),
                          nn.Tanh(),
                          nn.Linear(attention_size, 1, bias=False))
    return model
```

注意力机制的输入包括查询项、键项和值项。设编码器和解码器的隐藏单元个数相同。这里的查询项为解码器在上一时间步的隐藏状态，形状为(批量大小, 隐藏单元个数)；键项和值项均为编码器在所有时间步的隐藏状态，形状为(时间步数, 批量大小, 隐藏单元个数)。注意力机制返回当前时间步的背景变量，形状为(批量大小, 隐藏单元个数)。

```
def attention_forward(model, enc_states, dec_state):
    """
    enc_states: (时间步数, 批量大小, 隐藏单元个数)
    dec_state: (批量大小, 隐藏单元个数)
    """
    # 将解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
    dec_states = dec_state.unsqueeze(dim=0).expand_as(enc_states)
    enc_and_dec_states = torch.cat((enc_states, dec_states), dim=2)
    e = model(enc_and_dec_states)  # 形状为(时间步数, 批量大小, 1)
    alpha = F.softmax(e, dim=0)  # 在时间步维度做softmax运算
    return (alpha * enc_states).sum(dim=0)  # 返回背景变量
```

在下面的例子中，编码器的时间步数为10，批量大小为4，编码器和解码器的隐藏单元个数均为8。注意力机制返回一个小批量的背景向量，每个背景向量的长度等于编码器的隐藏单元个数。因此输出的形状为(4, 8)。

```
seq_len, batch_size, num_hiddens = 10, 4, 8
model = attention_model(2*num_hiddens, 10)
enc_states = torch.zeros((seq_len, batch_size, num_hiddens))
dec_state = torch.zeros((batch_size, num_hiddens))
attention_forward(model, enc_states, dec_state).shape # torch.Size([4, 8])
```

10.12.2.3 含注意力机制的解码器

我们直接将编码器在最终时间步的隐藏状态作为解码器的初始隐藏状态。这要求编码器和解码器的循环神经网络使用相同的隐藏层个数和隐藏单元个数。

在解码器的前向计算中，我们先通过刚刚介绍的注意力机制计算得到当前时间步的背景向量。由于解码器的输入来自输出语言的词索引，我们将输入通过词嵌入层得到表征，然后和背景向量在特征维连结。我们将连结后的结果与上一时间步的隐藏状态通过门控循环单元计算出当前时间步的输出与隐藏状态。最后，我们将输出通过全连接层变换为有关各个输出词的预测，形状为(批量大小, 输出词典大小)。

```
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 attention_size, drop_prob=0):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.attention = attention_model(2*num_hiddens, attention_size)
        # GRU的输入包含attention输出的c和实际输入, 所以尺寸是 num_hiddens+embed_size
        self.rnn = nn.GRU(num_hiddens + embed_size, num_hiddens,
                          num_layers, dropout=drop_prob)
        self.out = nn.Linear(num_hiddens, vocab_size)

    def forward(self, cur_input, state, enc_states):
        """
        cur_input shape: (batch, )
        state shape: (num_layers, batch, num_hiddens)
        """
        # 使用注意力机制计算背景向量
        c = attention_forward(self.attention, enc_states, state[-1])
```

```
        # 将嵌入后的输入和背景向量在特征维连结，(批量大小，num_hiddens+embed_size)
        input_and_c = torch.cat((self.embedding(cur_input), c), dim=1)
        # 为输入和背景向量的连结增加时间步维，时间步个数为1
        output, state = self.rnn(input_and_c.unsqueeze(0), state)
        # 移除时间步维，输出形状为(批量大小，输出词典大小)
        output = self.out(output).squeeze(dim=0)
        return output, state

    def begin_state(self, enc_state):
        # 直接将编码器最终时间步的隐藏状态作为解码器的初始隐藏状态
        return enc_state
```

10.12.3 训练模型

我们先实现 `batch_loss` 函数计算一个小批量的损失。解码器在最初时间步的输入是特殊字符 `BOS` 。之后，解码器在某时间步的输入为样本输出序列在上一时间步的词，即强制教学。此外，同10.3节（word2vec的实现）中的实现一样，我们在这里也使用掩码变量避免填充项对损失函数计算的影响。

```
def batch_loss(encoder, decoder, X, Y, loss):
    batch_size = X.shape[0]
    enc_state = encoder.begin_state()
    enc_outputs, enc_state = encoder(X, enc_state)
    # 初始化解码器的隐藏状态
    dec_state = decoder.begin_state(enc_state)
    # 解码器在最初时间步的输入是BOS
    dec_input = torch.tensor([out_vocab.stoi[BOS]] * batch_size)
    # 我们将使用掩码变量mask来忽略掉标签为填充项PAD的损失
    mask, num_not_pad_tokens = torch.ones(batch_size,), 0
    l = torch.tensor([0.0])
    for y in Y.permute(1,0): # Y shape: (batch, seq_len)
        dec_output, dec_state = decoder(dec_input, dec_state, enc_outputs)
        l = l + (mask * loss(dec_output, y)).sum()
        dec_input = y  # 使用强制教学
        num_not_pad_tokens += mask.sum().item()
        # EOS后面全是PAD. 下面一行保证一旦遇到EOS接下来的循环中mask就一直是0
        mask = mask * (y != out_vocab.stoi[EOS]).float()
    return l / num_not_pad_tokens
```

在训练函数中，我们需要同时迭代编码器和解码器的模型参数。

```
def train(encoder, decoder, dataset, lr, batch_size, num_epochs):
    enc_optimizer = torch.optim.Adam(encoder.parameters(), lr=lr)
    dec_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)

    loss = nn.CrossEntropyLoss(reduction='none')
    data_iter = Data.DataLoader(dataset, batch_size, shuffle=True)
    for epoch in range(num_epochs):
        l_sum = 0.0
        for X, Y in data_iter:
            enc_optimizer.zero_grad()
            dec_optimizer.zero_grad()
            l = batch_loss(encoder, decoder, X, Y, loss)
            l.backward()
            enc_optimizer.step()
            dec_optimizer.step()
            l_sum += l.item()
        if (epoch + 1) % 10 == 0:
            print("epoch %d, loss %.3f" % (epoch + 1, l_sum / len(data_iter)))
```

接下来，创建模型实例并设置超参数。然后，我们就可以训练模型了。

```
embed_size, num_hiddens, num_layers = 64, 64, 2
attention_size, drop_prob, lr, batch_size, num_epochs = 10, 0.5, 0.01, 2, 50
encoder = Encoder(len(in_vocab), embed_size, num_hiddens, num_layers,
                  drop_prob)
decoder = Decoder(len(out_vocab), embed_size, num_hiddens, num_layers,
                  attention_size, drop_prob)
train(encoder, decoder, dataset, lr, batch_size, num_epochs)
```

输出：

```
epoch 10, loss 0.441
epoch 20, loss 0.183
epoch 30, loss 0.100
epoch 40, loss 0.046
epoch 50, loss 0.025
```

10.12.4 预测不定长的序列

在10.10节（束搜索）中我们介绍了3种方法来生成解码器在每个时间步的输出。这里我们实现最简单的贪婪搜索。

```
def translate(encoder, decoder, input_seq, max_seq_len):
    in_tokens = input_seq.split(' ')
    in_tokens += [EOS] + [PAD] * (max_seq_len - len(in_tokens) - 1)
    enc_input = torch.tensor([[in_vocab.stoi[tk] for tk in in_tokens]]) # batch=1
    enc_state = encoder.begin_state()
    enc_output, enc_state = encoder(enc_input, enc_state)
    dec_input = torch.tensor([out_vocab.stoi[BOS]])
    dec_state = decoder.begin_state(enc_state)
    output_tokens = []
    for _ in range(max_seq_len):
        dec_output, dec_state = decoder(dec_input, dec_state, enc_output)
        pred = dec_output.argmax(dim=1)
        pred_token = out_vocab.itos[int(pred.item())]
        if pred_token == EOS:  # 当任一时间步搜索出EOS时，输出序列即完成
            break
        else:
            output_tokens.append(pred_token)
            dec_input = pred
    return output_tokens
```

简单测试一下模型。输入法语句子“ils regardent.”，翻译后的英语句子应该是“they are watching.”。

```
input_seq = 'ils regardent .'
translate(encoder, decoder, input_seq, max_seq_len)
```

输出：

```
['they', 'are', 'watching', '.']
```

10.12.5 评价翻译结果

评价机器翻译结果通常使用BLEU（Bilingual Evaluation Understudy）[1]。对于模型预测序列中任意的子序列，BLEU考察这个子序列是否出现在标签序列中。

具体来说，设词数为n的子序列的精度为p_n。它是预测序列与标签序列匹配词数为n的子序列的数量与预测序列中词数为n的子序列的数量之比。举个例子，假设标签序列为A、B、C、D、E、F，预测序列为A、B、B、C、D，那么$p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$。设$len_{\text{label}}$和$len_{\text{pred}}$分别为标签序列和预测序列的词数，那么，BLEU的定义为

$$\exp\left(\min\left(0, 1 - \frac{len_{\text{label}}}{len_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n},$$

其中k是我们希望匹配的子序列的最大词数。可以看到当预测序列和标签序列完全一致时，BLEU为1。

因为匹配较长子序列比匹配较短子序列更难，BLEU对匹配较长子序列的精度赋予了更大权重。例如，当p_n固定在0.5时，随着n的增大，$0.5^{1/2} \approx 0.7, 0.5^{1/4} \approx 0.84, 0.5^{1/8} \approx 0.92, 0.5^{1/16} \approx 0.96$。另外，模型预测较短序列往往会得到较高$p_n$值。因此，上式中连乘项前面的系数是为了惩罚较短的输出而设的。举个例子，当$k = 2$时，假设标签序列为A、B、C、D、E、F，而预测序列为A、B。虽然$p_1 = p_2 = 1$，但惩罚系数$\exp(1 - 6/2) \approx 0.14$，因此BLEU也接近0.14。

下面来实现BLEU的计算。

```
def bleu(pred_tokens, label_tokens, k):
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[''.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[''.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[''.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

接下来，定义一个辅助打印函数。

```
def score(input_seq, label_seq, k):
    pred_tokens = translate(encoder, decoder, input_seq, max_seq_len)
    label_tokens = label_seq.split(' ')
```

```
        print('bleu %.3f, predict: %s' % (bleu(pred_tokens, label_tokens, k),
                                          ' '.join(pred_tokens)))
```

预测正确则分数为1。

```
score('ils regardent .', 'they are watching .', k=2)
```

输出：

```
bleu 1.000, predict: they are watching .
```

测试一个不在训练集中的样本。

```
score('ils sont canadienne .', 'they are canadian .', k=2)
```

输出：

```
bleu 0.658, predict: they are russian .
```

小结

- 可以将编码器—解码器和注意力机制应用于机器翻译中。
- BLEU可以用来评价翻译结果。