NLP-Beginner的语言模型任务

> 链接：https://github.com/FudanNLP/nlp-beginner

1.任务介绍

- 训练过程中如同翻译模型中的teaching forcing，输入是$[< start >, x1, x2, \dots xn]$，期待的输出是$[x1, x2, x3, \dots, xn, < end >]$
- 训练完成后进行测试时也就是古诗生成时有两种可选模式
  - 一种是将输入的文本作为古诗的开头，将输入的x输入到GRU中，将最后一个time step的隐状态作为解码开始的隐状态，解码过程就是将上一步解码得到的输出作为下一个time step的输入，直到出现或者达到最大长度。
  - 一种是将输入的句子的每个字作为每个诗句的开头（也就是生成藏头诗），在每个句子的开头输入指定的单词，然后进行解码。
- 数据集太小只有164首，生成效果惨不忍睹，也没再花更多的时间

2.语言模型

语言模型：计算一个句子概率的模型，形式化表达如下，其中S表示句子，w_i表示句子中相应的词语
$S = w_1, w_2, \dots, w_n, P(S) = P(w_1, w_2, \dots, w_n) = P(w_1)P(w_2|w_1)\dots P(w_n|w_{n-1}\dots w_1)$

但是$P(w_n|w_{n-1}\dots w_1)$的计算比较困难

- 对应词的组合太多会导致参数空间太大
- 词较多的时候词的组合比较稀疏（大多数的组合不会出现），也就学不到有用的知识

解决这一问题的主流思路是马尔科夫假设:一个词出现的概率只与它前面出现的一个或几个词有关

- 如果一个词出现的概率与它周围的词无关，我们称之为一元语言模型/unigram
  $P(S) \approx P(w_1)P(w_2)\dots P(w_n)$
- 如果一个词出现的概率只依赖于它前面的一个词，称之为二元语言模型/bigram
  $P(S) \approx P(w_1)P(w_2|w_1)\dots P(w_n|w_{n-1})$
- 如果一个词出现的概率只依赖于它前面的两个词，称之为三元语言模型/trigram
  $P(S) \approx P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)\dots P(w_n|w_{n-1}, w_{n-2})$

一般来说，N元语言模型就是当前词的概率只与它前面的N-1个词有关，相应的概率参数通过大量的语料库进行计算求得。

一般使用bigram和trigram，因为4元以上需要更多的语料数据且精度没有明显提升

一个小例子：

- Unigram: 前提假设是每个词的概率与其他词无关，$P(w_1), P(w_2)\dots$的计算采取极大似然估计：

```
S1=我，喜欢，你，我，认真，的
S2=我，开心
语料库总size=8，词典size=6
P(我)=3/8, P(喜欢)=1/8, P(你)=1/8,
P(认真)=1/8, P(的)=1/8, P(开心)=1/8
```

若要计算新句子的概率，直接对相应word的概率相乘即可。

- Bigram:一般会添加开始/结束标记：
  $P(S) \approx P(w_1|<s>)P(w_2|w_1)...P(w_n|w_{n-1})P(</s>|w_n)$，每一项计算如下：
  $p(w_i|w_{i-1}) = \frac{c(w_{i-1},w_i)}{w_{i-1}}$

  通常情况下，Bigram 比unigram能更好地考虑上下文

```
p(he eats pizza) = p(he|<s>)p(eats|he)*p(pizza|eats)*p(</s>|pizza)
p(he drinks pizza) = p(he|<s>)p(drinks|he)*p(pizza|drinks)*p(</s>|pizza)
一般来说，p(pizza|eats) > p(pizza|drinks)，则说明eats搭配pizza更合理。
```

3.困惑度(perplexity)与交叉熵的关系

3.1 困惑度

困惑度用来评价语言模型好坏，基本思想：给测试集的句子赋予较高概率值的语言模型较好，当语言模型训练完之后，测试集中的句子都是正常句子，那么训练好的模型在测试集上的概率是越高越好。形式化表达如下：

$PP(S) = P(w_1, w_2...w_n)^{-\frac{1}{n}} = \prod_{i=1}^{n} P(w_i|w_{i-1}...w_i)^{-\frac{1}{n}}$，由公式可得句子的概率越大，语言模型越好，困惑度越小，两边取对数得:

$log(PP(S)) = -\frac{1}{n}\sum_{i=1}^{n} logP(w_i|w_{i-1}...w_i)$

3.2PPL与交叉熵的关系

困惑度是交叉熵的指数形式

- 交叉熵：交叉熵描述了两个概率分布之间的一种距离

  假设x是一个随机变量，$u(x), v(x)$是两个与x相关的概率分布，那么u, v之间的交叉熵定义为分布在u下$-log(v(x))$的期望值
  $H(u,v) = E_u[-log(v(x))] = -\sum_x u(x)log(v(x))$
- 我们把x看作句子中的单词，u(x)表示每个位置上单词的真实分布,v(x)是模型的预测分布
  $P(w_i|w_{i-1}...w_i)$

  $u(x|w_1, w_2...w_{i-1}) = \begin{cases} 1, x = w_i \\ 0, x \neq w_i \end{cases}$

  则该句的交叉熵：

$H(u,v) = -\sum_x u(x)log(v(x))$
$= -\frac{1}{n}\sum_{i=1}^{n}(\sum_x u(x|w_1,w_2\dots w_{i-1})logP(w_i|w_{i-1}\dots w_i)$
$= -\frac{1}{n}\sum_{i=1}^{n}(1\times \log P(w_i|w_{i-1}\dots w_i) + \sum_{x\neq w_i} 0\times \log P(w_i|w_{i-1}\dots w_i))$
$= -\frac{1}{n}\sum_{i=1}^{n}\log P(w_i|w_{i-1}\dots w_i)$

因此log perplexity和交叉熵是等价的

关于 perplexity，更常见的表述如下，所有表述方式实质上是等价的：

> 假设测试语料共包括 m 个句子，第 i 个句子有 n_i 个字，则有 $M=\sum_{i=1}^{m} n_i$ 对测试语料中的每一个句子 $x^{(i)}$，若 $\prod_{i=1}^{m} p(x^{(i)})$ 越大，则说明语言模型在测试语料上的表现越好。即相当于下式 $l=\frac{1}{M}\sum_{i=1}^{m}\log_2 p(x^{(i)})$ 而 perplexity 定义为 $PP=2^{-l}$

4.TorchText

`TorchText`：pytorch中文本数据处理的库（图像处理库`torchvision`），可以将文本直接转化为Batch，在模型训练时使用。

其数据预处理流程如下：

1. 定义数据处理的方式，使用`torchtext.data.Field`，创建Field对象

```
SENT = data.Field(sequential=True, tokenize=tokenizer, lower=False,
init_token='<START>', eos_token='<END>', batch_first=True)
LABEL = data.Field(sequential=False, use_vocab=False)
```

`sequential`表示是否切分数据，如果数据已经是序列化的了而且是数字类型的，则为False且参数`use_vocab = False`
`tokenize` 传入一个函数，表示如何将文本str变成token，默认`str.split`
`init_token`每一条数据的起始字符
`eos_token`每一条数据的结束字符
`batch_first`batch dimension是否在第0维
`pad_token`、`unk_token`：默认、

2. 加载corpus，即构建Dataset。TorchText中有各种内置Dataset，用于处理常见的数据格式，TabularDataset可以处理CSV/TSV/JSON格式的文件，比较常用。通过·`torchtext.data.Dataset`类的方法`split`加载语料库

```
train_dataset, valid_dataset = data.TabularDataset.splits(path='',
train=target_path, validation=target_path, format='csv',skip_header=True,
fields=[('sent', SENT)])
```

`path`数据存放的根目录
`skip_header`跳过第一行
`fileds`是一个列表，每个元素是一个元组(name, field)，name表示源文件的列名，field是Field对象，field必须与列的顺序相同，不使用的列可传入None

```
tst_datafields = [("id", None), # 我们不会需要id，所以我们传入的filed是None
                  ("comment_text", TEXT)]
```

3. 建立词表

```
SENT.build_vocab(train_dataset,vectors="glove.6B.100d"))
```

`SENT.vocab`即我们需要得到的词典类
`SENT.vocab.stoi`包含word到id的映射,按词频由高到底进行排序
`SENR.vocab.itos`包含id到word的映射

可以使用预训练的word_vectors，直接传入string，后台会自动下载。也可以通过vocab.vectors使用自定义的vectors

4. 构造迭代器

```
train_iter = data.BucketIterator(train_dataset, batch_size=batch_size,
sort_key=lambda x:len(x.sent), shuffle=False, device=device)
```

`sort_key`排序的key，这样的话相近长度的样本就会杯batch到一起，减少pad
`sort_with_batch`batch内部是否进行排序，设置为True可以方便RNN进行pack和pad

然后就可以直接输进模型啦

```
for i, input in enumerate(train_iter):
   sent = input.sent
   label = input.label
   pass
```