学习笔记-2：Task2

4 - 文本预处理

建立字典

为了方便模型处理，我们需要将字符串转换为数字. 因此我们需要先构建一个字典(vocabulary)将每个词映射到一个唯一的索引编号.

核心代码注释

```
class Vocab(object):
    def __init__(self, tokens, min_freq=0, use_special_tokens=False):
        # tokens: tokenize函数的返回值，一个2D-list
        # min_freq: 频率下限，低于此频率的词将被忽略
        counter = count_corpus(tokens)  # counter 是一个字典，key-value: 词-词频
        self.token_freqs = list(counter.items())
        self.idx_to_token = []
        if use_special_tokens:
            # padding, begin of sentence, end of sentence, unknown
            # padding: 在矩阵用于补全较短的行(句子)，使各行长度一致
            # unknown: 语料库中尚未收录的词
            self.pad, self.bos, self.eos, self.unk = (0, 1, 2, 3)
            self.idx_to_token += ['<pad>', '<bos>', '<eos>', '<unk>']
        else:
            self.unk = 0
            self.idx_to_token += ['<unk>']
        self.idx_to_token += [token for token, freq in self.token_freqs
                              if freq >= min_freq and token not in self.idx_to_token]  #
一个包含所有词的列表
        self.token_to_idx = dict()  # 建立词的索引到词的映射
        for idx, token in enumerate(self.idx_to_token):
            self.token_to_idx[token] = idx

    def __len__(self):
        return len(self.idx_to_token)

    # 从词到索引的映射
    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):  # 如果tokens既不是list也不是
tuple
            '''
            dict.get(key, default = None)
            key -- 字典中要查找的键
            default -- 如果指定键的值不存在时，返回该默认值
            '''
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    # 给定索引，返回对应的词
    def to_tokens(self, indices):
```

```
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

def count_corpus(sentences):
    tokens = [tk for st in sentences for tk in st]  # 把sentences这个2D-list 展开
    return collections.Counter(tokens)  # 返回一个字典，记录每个词的出现次数
```

vocab[str] 会自动调用 vocab.__getitem__() 函数 (魔法方法), 所以

```
vocab = Vocab(tokens)
vocab['the']
>>> 1
```

5 - 语言模型与数据集

语言模型

给定一个长度为T的词的序列$w_1, w_2, \ldots, w_T$，假设序列中的每个词是依次生成的, 我们有

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$$

$$= P(w_1)P(w_2 \mid w_1)\cdots P(w_T \mid w_1 w_2 \cdots w_{T-1})$$

n元语法

n元语法通过马尔可夫假设简化模型，马尔科夫假设是指一个词的出现只与前面n个词相关，即n阶马尔可夫链 (Markov chain of order n)
基于$n-1$阶马尔可夫链，我们可以将语言模型改写为

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1})$$

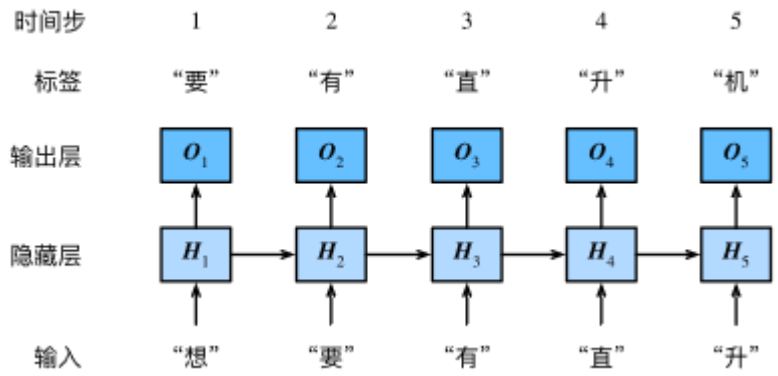
以上也叫n元语法(n-grams), 它是基于$n-1$阶马尔可夫链的概率语言模型

时序数据的采样

在训练中我们需要每次随机读取小批量样本和标签, 时序数据的一个样本通常包含连续的字符, 且样本的标签序列为这些字符分别在训练集中的下一个字符.
如果序列的长度为T, 时间步数为n, 那么一共有$T-n$个合法的样本, 但是这些样本有大量的重合, 我们通常采用更加高效的采样方式. 我们有两种方式对时序数据进行采样, 分别是随机采样和相邻采样.

6 - 循环神经网络

基于当前的输入与过去的输入序列, 预测序列的下一个字符. 循环神经网络引入一个隐藏变量H，用H_t表示H在时间步t的值. H_t的计算基于X_t和H_{t-1}, 可以认为H_t记录了到当前字符为止的序列信息, 利用H_t对序列的下一个字符进行预测.



循环神经网络的构造

假设$\boldsymbol{X_t} \in \mathbb{R}^{n\times d}$是时间步$t$的小批量输入, $\boldsymbol{H_t} \in \mathbb{R}^{n\times h}$是该时间步的隐藏变量, 则

$$\boldsymbol{H_t} = \phi(\boldsymbol{X_t}\boldsymbol{W_{xh}} + \boldsymbol{H_{t-1}}\boldsymbol{W_{hh}} + \boldsymbol{b_h})$$

其中, $\boldsymbol{W_{xh}} \in \mathbb{R}^{d\times h}$, $\boldsymbol{W_{hh}} \in \mathbb{R}^{h\times h}$, $\boldsymbol{b_h} \in \mathbb{R}^{1\times h}$, ϕ函数是非线性激活函数

在时间步t, 输出层的输出为

$$\boldsymbol{O_t} = \boldsymbol{H_t}\boldsymbol{W_{hq}} + \boldsymbol{b_q}$$

其中$\boldsymbol{W_{hq}} \in \mathbb{R}^{h\times q}$, $\boldsymbol{b_q} \in \mathbb{R}^{1\times q}$

one-hot向量

假设词典大小是N, 每个字符对应一个从0到$N-1$的唯一的索引, 则该字符的向量是一个长度为N的向量, 若字符的索引是i, 则该向量的索引是i的元素为1, 其他位置为0.

```
def one_hot(x, n_class, dtype = torch.float32):
    """
    x: 一维向量，每个元素都是一个字符的索引
    n_class: 字典的大小
    """
    result = torch.zeros(x.shape[0], n_class, dtype = dtype, device = x.device)  #
shape: (n, n_class)
    result.scatter_(1, x.long().view(-1, 1), 1)  # result[i, x[i, 0]] = 1
    return result

# an example
x = torch.tensor([0, 2])
x_one_hot = one_hot(x, vocab_size)

print(x_one_hot)
>>> tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.]])

print(x_one_hot.shape)
>>> torch.Size([2, 1027])
```

```
print(x_one_hot.sum(axis = 1))
>>> tensor([1., 1.])
```

每次采样的小批量的形状是批量大小×时间步数. 下面的函数将这样的小批量变换成数个形状为批量大小×词典大小的矩阵, 矩阵个数等于时间步数.
也就是说, 时间步t的输入为$\boldsymbol{X}_t \in \mathbb{R}^{n\times d}$, 其中$n$为批量大小, d为词向量大小, 即one-hot向量长度

```
def to_onehot(X, n_class):
    return [one_hot(X[:, i], n_class) for i in range(X.shape[1])]  # 取出 X 的每一列，变成一个 one-hot 矩阵 (n×d)

X = torch.arange(10).view(2, 5)  # 2是batch size, 5是时间步数
inputs = to_onehot(X, vocab_size)  # inputs 中有5个矩阵，每个的大小都是 2×1027
print(len(inputs), inputs[0].shape)
>>> 5 torch.Size([2, 1027])
```

裁剪梯度

循环神经网络中较容易出现梯度衰减或梯度爆炸, 这会导致网络几乎无法训练. 裁剪梯度(clip gradient)是一种应对梯度爆炸的方法. 假设我们把所有模型参数的梯度拼接成一个向量 $\boldsymbol{g}$，并设裁剪的阈值是θ. 裁剪后的梯度

$$\min\left(\frac{\theta}{|\boldsymbol{g}|}, 1\right)\boldsymbol{g}$$

的L_2范数不超过θ

定义预测函数

以下函数基于前缀prefix(含有数个字符的字符串)来预测接下来的num_chars个字符

```
def predict_rnn(prefix, num_chars, rnn, params, init_rnn_state,
                num_hiddens, vocab_size, device, idx_to_char, char_to_idx):
    state = init_rnn_state(1, num_hiddens, device)
    # output记录prefix加上预测的num_chars个字符, 记录的是字符的索引
    output = [char_to_idx[prefix[0]]]
    for t in range(num_chars + len(prefix) - 1):
        # 将上一时间步的输出作为当前时间步的输入
        # 下面的torch.tensor是1×1的tensor, 代表batch_size=1, 1个时间步
        X = to_onehot(torch.tensor([[output[-1]]], device=device), vocab_size)
        # 计算输出和更新隐藏状态
        (Y, state) = rnn(X, state, params)
        # 下一个时间步的输入是prefix里的字符或者当前的最佳预测字符
        if t < len(prefix) - 1:
            # 还在处理prefix里的字符, 无需预测
            output.append(char_to_idx[prefix[t + 1]])
        else:
            # 对之后的字符进行预测
            output.append(Y[0].argmax(dim=1).item())
```

```
    return ''.join([idx_to_char[i] for i in output])  # 把ouput中的索引转换为字符并
连起来
```