N-Gram 模型

下面我们直接用代码进行说明

```
CONTEXT_SIZE = 2 # 依据的单词数
EMBEDDING_DIM = 10 # 词向量的维度
# 我们使用莎士比亚的诗
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""".split()
```

这里的 `CONTEXT_SIZE` 表示我们希望由前面几个单词来预测这个单词，这里使用两个单词， `EMBEDDING_DIM` 表示词嵌入的维度。

接着我们建立训练集，便利整个语料库，将单词三个分组，前面两个作为输入，最后一个作为预测的结果。

```
 trigram = [((test_sentence[i], test_sentence[i+1]), test_sentence[i+2])
            for i in range(len(test_sentence)-2)]
```

```
# 总的数据量
len(trigram)
```

```
113
```

```
# 取出第一个数据看看
trigram[0]
```

```
(('When', 'forty'), 'winters')
```

```
# 建立每个词与数字的编码，据此构建词嵌入
vocb = set(test_sentence) # 使用 set 将重复的元素去掉
word_to_idx = {word: i for i, word in enumerate(vocb)}
idx_to_word = {word_to_idx[word]: word for word in word_to_idx}
```

```
word_to_idx
```

```
{"'This": 94,
 'And': 71,
 'How': 18,
 'If': 49,
 'Proving': 78,
 'Shall': 48,
 'Then': 33,
 'This': 68,
 'Thy': 75,
 'To': 81,
 'Were': 61,
 'When': 14,
 'Where': 95,
 'Will': 27,
 'a': 21,
 'all': 53,
 'all-eating': 3,
 'an': 15,
 'and': 23,
 'answer': 80,
 'art': 70,
 'asked,': 69,
 'be': 29,
 'beauty': 16,
 "beauty's": 40,
 'being': 79,
 'besiege': 55,
 'blood': 11,
 'brow,': 1,
 'by': 59,
 'child': 8,
 'cold.': 32,
 'couldst': 26,
 'count,': 77,
 'days;': 43,
 'deep': 62,
 "deserv'd": 41,
 'dig': 64,
 "excuse,'": 86,
 'eyes,': 84,
 'fair': 56,
 "feel'st": 44,
 'field,': 9,
 'forty': 46,
 'gazed': 93,
 'held:': 12,
 'his': 89,
 'in': 45,
 'it': 34,
 'lies,': 57,
 'livery': 28,
 'lusty': 65,
 'made': 54,
 'make': 42,
 'mine': 13,
 'more': 83,
 'much': 30,
 'my': 50,
 'new': 92,
 'now,': 25,
 'of': 47,
 'old': 22,
 'old,': 19,
 'on': 74,
 'own': 20,
 'praise': 38,
 'praise.': 96,
 'proud': 5,
 'say,': 63,
 'see': 58,
 'shall': 87,
 'shame,': 90,
 'small': 31,
 'so': 67,
 'succession': 36,
 'sum': 10,
 'sunken': 60,
 'the': 73,
 'thine': 24,
 'thine!': 0,
 'thou': 51,
 'thriftless': 72,
 'thy': 76,
 'to': 85,
 "totter'd": 2,
 'treasure': 17,
 'trenches': 39,
 'use,': 35,
 'warm': 66,
 'weed': 91,
 'were': 82,
 'when': 7,
 'where': 37,
 'winters': 88,
 'within': 4,
 'worth': 52,
 "youth's": 6}
```

从上面可以看到每个词都对应一个数字，且这里的单词都各不相同

接着我们定义模型，模型的输入就是前面的两个词，输出就是预测单词的概率

```
import torch
from torch import nn
import torch.nn.functional as F
from torch.autograd import Variable
```

```
# 定义模型
class n_gram(nn.Module):
    def __init__(self, vocab_size, context_size=CONTEXT_SIZE, n_dim=EMBEDDING_DIM):
        super(n_gram, self).__init__()

        self.embed = nn.Embedding(vocab_size, n_dim)
        self.classify = nn.Sequential(
            nn.Linear(context_size * n_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, vocab_size)
        )

    def forward(self, x):
        voc_embed = self.embed(x) # 得到词嵌入
        voc_embed = voc_embed.view(1, -1) # 将两个词向量拼在一起
        out = self.classify(voc_embed)
        return out
```

最后我们输出就是条件概率，相当于是一个分类问题，我们可以使用交叉熵来方便地衡量误差

```
net = n_gram(len(word_to_idx))

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=1e-2, weight_decay=1e-5)
```

```
for e in range(100):
    train_loss = 0
    for word, label in trigram: # 使用前 100 个作为训练集
        word = Variable(torch.LongTensor([word_to_idx[i] for i in word])) # 将两个词作为输入
        label = Variable(torch.LongTensor([word_to_idx[label]]))
        # 前向传播
        out = net(word)
        loss = criterion(out, label)
        train_loss += loss.data[0]
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if (e + 1) % 20 == 0:
        print('epoch: {}, Loss: {:.6f}'.format(e + 1, train_loss / len(trigram)))
```

```
epoch: 20, Loss: 0.088273
epoch: 40, Loss: 0.065301
epoch: 60, Loss: 0.057113
epoch: 80, Loss: 0.052442
epoch: 100, Loss: 0.049236
```

最后我们可以测试一下结果

```
net = net.eval()
```

```
# 测试一下结果
word, label = trigram[19]
print('input: {}'.format(word))
print('label: {}'.format(label))
print()
word = Variable(torch.LongTensor([word_to_idx[i] for i in word]))
out = net(word)
pred_label_idx = out.max(1)[1].data[0]
predict_word = idx_to_word[pred_label_idx]
print('real word is {}, predicted word is {}'.format(label, predict_word))
```

```
input: ('so', 'gazed')
label: on

real word is on, predicted word is on
```

```
word, label = trigram[75]
print('input: {}'.format(word))
print('label: {}'.format(label))
print()
word = Variable(torch.LongTensor([word_to_idx[i] for i in word]))
out = net(word)
pred_label_idx = out.max(1)[1].data[0]
predict_word = idx_to_word[pred_label_idx]
print('real word is {}, predicted word is {}'.format(label, predict_word))
```

```
input: ("'This", 'fair')
label: child

real word is child, predicted word is child
```

可以看到网络在训练集上基本能够预测准确，不过这里样本太少，特别容易过拟合。

下一次课我们会讲一讲 RNN 如何应用在自然语言处理中