## 词向量

### 离散表示 one-hot

将一段文本的不重复的词汇进行one-hot编码。  
但是使用one-hot并非一个很好的选择 主要原因是：1、如果词汇数量很多，那么one-hot的维度就会很大。2、数据稀疏。3、无法准确的表示不同词之间的相似度。

### Skip-Gram（跳字）模型

跳字模型假设基于某个词开生成它在文本序列周围的词。

这以《动身学习深度学习》书籍中的例子，"The man loves his son"。以"love"为中心词，背景窗口大小为2，跳字模型关心的是给中心词"loves" 生成与它距离不超过2个词的背景词"The" "man" "his" "son"的条件概率。

假设给定的中心词的情况下，背景词的生成是相互独立的，那么上式可以改写为

在跳字模型中，每个词被表示为两个d维向量，（d 维 为不重复词汇的数量）用来计算条件概率。

假设这个词在词典中的索引为i，当它为中心词向量表示为背景词表示为.设中心词在词典中的索引为c。背景词的索引为 给定中心词生成背景词的条件概率可以通过对向量内积做softmax运算得到。

其中词典索引集 假设给定一个长度为T的文本序列，设时间步t的词为 假设给定中性词的情况下背景词的生成相互独立，当背景窗口大小为m时，跳字模型的似然函数即给定中心词所生成的所有背景词的概率

需要求上式的最大值，等价于求下式的最小值

也就是求上式梯度即可

根据定义可以求出

求导得到结果为

$$\frac{\partial{logP(w^{t+j}|w^t})}{\part{v\_c}}=u\_o-{\sum\_{j\in{\mathcal{V}}}P(w\_j|w\_c)}u\_j$$

从推导结果可以看出他的计算需要词典中所有词以为中心词的条件概率。

### 连续词袋模型

连续词袋模型与跳字模型类似，概念相反，跳字模型是使用中心词预测周围词，连续词袋模型利用周围词预测中性词。

也就是

因为连续词袋模型的背景词有多个，将这些背景词向量取平均，然后使用和跳字模型一样的方法来计算条件概率。设和分别表示词典中索引为i的词作为背景词和中心词的向量。设中心词在词典中的索引为c，背景词 在词典中的索引为，那么给定背景词生成中心词的条件概率为

给出一个长度为T的文本序列，设时间步t的词为,背景窗口大小为m。连续词袋模型的似然函数是由背景词生成任一中心词的概率为

求导得到

$$\frac{\part{logP(w\_c|\mathcal{W\_0})}}{\part{v\_{o\_i}}}=\frac{1}{2m}(u\_c-\sum\_{j\in{\mathcal{V}}}P(w\_j|{\mathcal{W\_o}})u\_j)
\\
\\
\\
\mathcal{W}=\{w\_{o\_1},....w\_{o\_{2m}}\}$$

从上式也可以看出需要对词典中的所有词进行计算。

## 进行近似训练

以跳字模型为例，使用softmax运算得到给定中性词生成背景词的条件概率

损失函数为

由于softmax考虑到了背景词可能是词典V中任意一个词，上面的损失函数包含了词典大小数目项的累加。计算代价过大，因此采用近似训练法“负采样”或者“层序softmax”进行训练。

### 负采样

负采样修改了原来的目标函数。给定中心词的一个背景窗口，我们把背景词出现在该背景窗口看作一个事件，并将该事件的概率计算为

其中函数与sigmoid激活函数的定义相同

def forward\_noise(self, size, N\_SAMPLES):  
 noise\_dist = self.noise\_dist  
 #从词汇分布中采样负样本  
 noise\_words = torch.multinomial(noise\_dist,  
 size \* N\_SAMPLES,  
 replacement=True)  
 noise\_vectors = self.out\_embed(noise\_words).view(  
 size, N\_SAMPLES, self.n\_embed)  
 return noise\_vectors

我们先考虑最大化文本序列中所有该事件的联合概率来训练词向量。具体来说，给定一个长度为T的文本序列，设时间步t的词为且背景窗口大小为考虑最大化联合概率

然而，以上模型中包含的事件仅考虑了正样本，则就导致了当所有词向量相等且为无穷大时以上的联合概率才能最大化为1。负采样通过采样添加负样本使目标函数变得更加有意义。设背景词出现在中心词的一个背景窗口为事件P，我们根据分布P(w)，采样K个未出现在该背景窗口中的词，也就是噪声词。设噪声词不出现在中心词的背景为事件.假设同时含有正样本和负样本的事件为P,相互独立，负采样将以上需要最大化的仅考虑正样本的联合概率改写为

其中

对于损失函数：

#定义损失函数  
class NegativeSamplingLoss(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def forward(self, input\_vectors, output\_vectors, noise\_vectors):  
 BATCH\_SIZE, embed\_size = input\_vectors.shape  
 #将输入词向量与目标词向量作维度转化处理  
 input\_vectors = input\_vectors.view(BATCH\_SIZE, embed\_size, 1)  
 output\_vectors = output\_vectors.view(BATCH\_SIZE, 1, embed\_size)  
 #目标词损失  
 test = torch.bmm(output\_vectors, input\_vectors)  
 out\_loss = torch.bmm(output\_vectors, input\_vectors).sigmoid().log()  
 out\_loss = out\_loss.squeeze()  
 #负样本损失  
 noise\_loss = torch.bmm(noise\_vectors.neg(),  
 input\_vectors).sigmoid().log()  
 noise\_loss = noise\_loss.squeeze().sum(1)  
 #综合计算两类损失  
 return -(out\_loss + noise\_loss).mean()

这时的损失函数已经与词典的大小没有了关系，只与K具有线性关系。

使用Pytorch实现skip-gram

数据处理

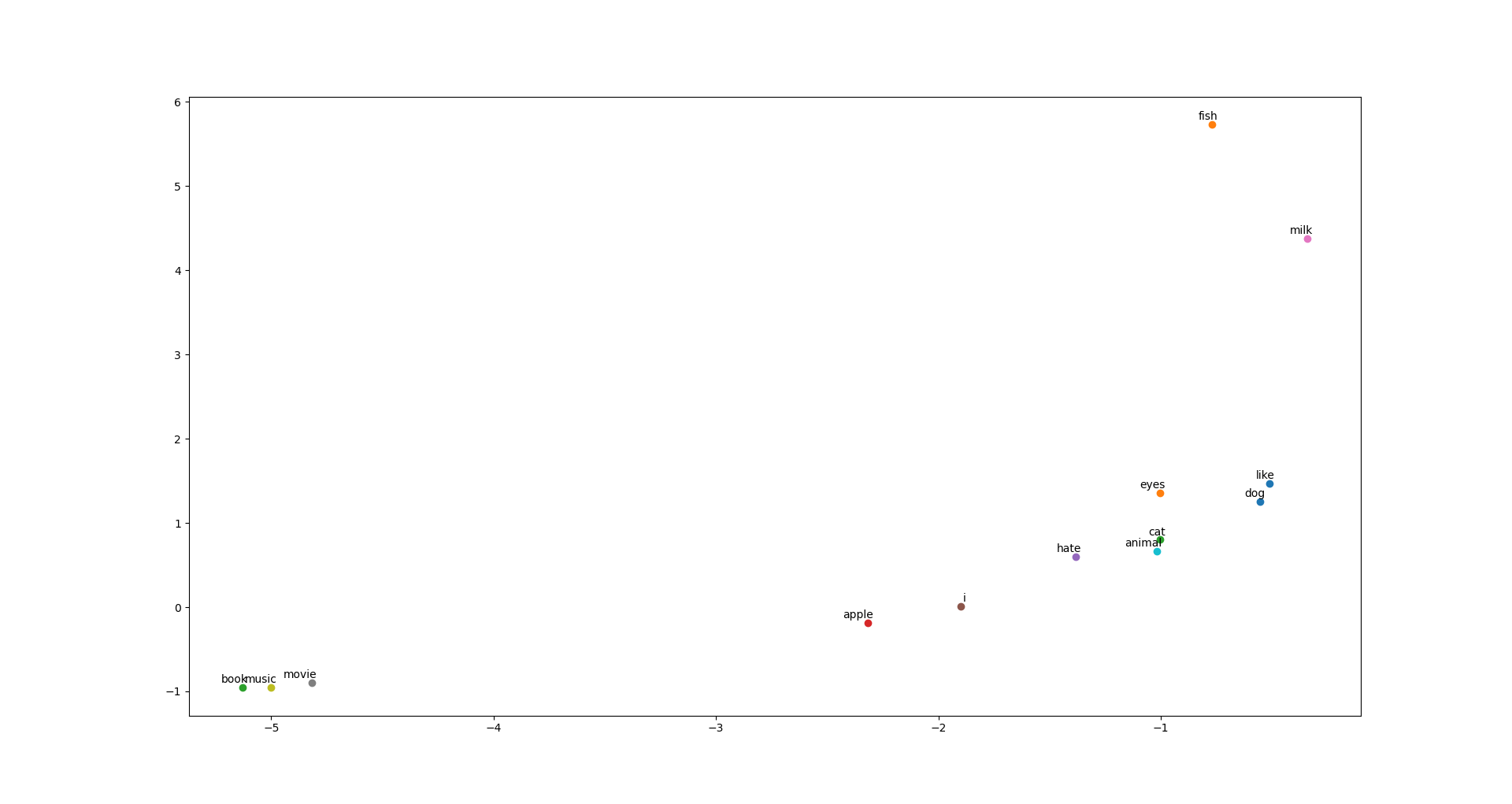
def process(text: str, appearance\_rate: int):  
 """  
 处理文本 将文本转换成为单词列表 并去除低于appearance\_rate的单词  
 """  
 # 将所有的词汇转换成小写  
 text = text.lower()  
 # 将句子转换成词汇  
 words = text.split()  
 # 统计词汇出现的次数  
 word\_counts = Counter(words)  
 # 去除低于appearance\_rate的词汇  
 trimmed\_words = [  
 word for word in words if word\_counts[word] > appearance\_rate  
 ]  
 return trimmed\_words

模型定义

def get\_train\_words(word\_freqs, t=0.1):  
 assert t > 0  
 train\_words = [  
 word for word, rate in zip(word\_freqs.keys(), word\_freqs.values())  
 if random.uniform(0, 1) > (1 - math.sqrt(t / rate))  
 ]  
 return train\_words  
  
  
print(get\_train\_words(word\_freqs, t=0.2))  
  
train\_words = get\_train\_words(word\_freqs, t=0.2)  
  
word\_freqs = np.array(list(word\_freqs.values()))  
print(word\_freqs)  
unigram\_dist = word\_freqs / word\_freqs.sum()  
noise\_dist = torch.from\_numpy(unigram\_dist\*\*(0.75) /  
 np.sum(unigram\_dist\*\*(0.75))) # 论文中为3/4  
  
print(noise\_dist)

定义网络模型

class SKipGramNet(nn.Module):  
 def \_\_init\_\_(self, n\_vocab, n\_embed, noise\_dist):  
 super().\_\_init\_\_()  
 self.n\_vocab = n\_vocab  
 self.n\_embed = n\_embed  
 self.noise\_dist = noise\_dist  
 #定义词向量层  
 self.in\_embed = nn.Embedding(n\_vocab, n\_embed)  
 self.out\_embed = nn.Embedding(n\_vocab, n\_embed)  
 #词向量层参数初始化  
 self.in\_embed.weight.data.uniform\_(-1, 1)  
 self.out\_embed.weight.data.uniform\_(-1, 1)  
  
 #输入词的前向过程  
 def forward\_input(self, input\_words):  
 input\_vectors = self.in\_embed(input\_words)  
 return input\_vectors  
  
 #目标词的前向过程  
 def forward\_output(self, output\_words):  
 output\_vectors = self.out\_embed(output\_words)  
 return output\_vectors  
 #负样本词的前向过程  
 def forward\_noise(self, size, N\_SAMPLES):  
 noise\_dist = self.noise\_dist  
 #从词汇分布中采样负样本  
 noise\_words = torch.multinomial(noise\_dist,  
 size \* N\_SAMPLES,  
 replacement=True)  
 noise\_vectors = self.out\_embed(noise\_words).view(  
 size, N\_SAMPLES, self.n\_embed)  
 return noise\_vectors

训练可视化结果

可以看出 "book","music","movie" 距离很近 " animal","cat","like","hate","dog" 距离很近 从训练的文本中也可以看出词汇之间的关系训练的相对还是不错的。