Word2vec

实现逻辑

[Word2vec ilustrado (tacosdedatos.com)]

在传统NLP中，把单词视为离散符号，hotel, motel, conference，被叫做 localist representation。

单词可以被表示成 one-hot vector，比如 I LOVE MY PARENTS. 中，LOVE = [0 1 0 0] 。但是对一个语言体系而言，总的单词量是相当可观的，这导致了one-hot vector 的维度非常大，这是其中一个问题。

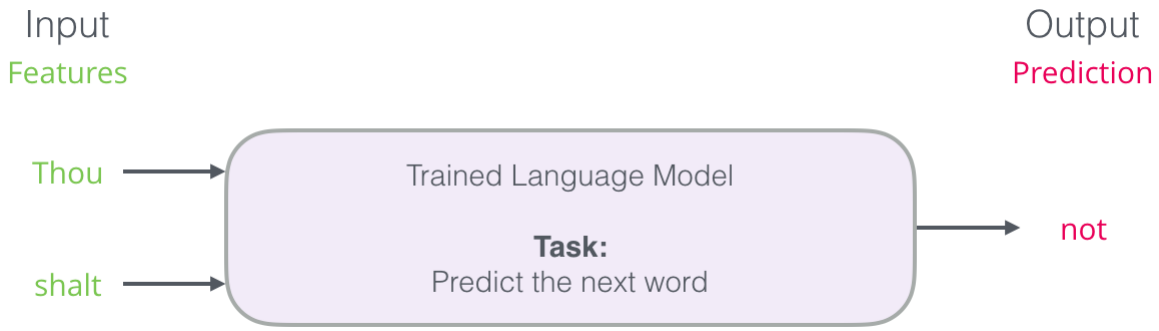
还有个问题就是，每一列都用0和1表达，离散的且一个 one-hot vector 只有一个位置上是 1，是 localist representation，表达能力比较弱。更进一步说这种表达方式很难理解词语之间关系，motel 和 hotel 几乎是一样的，但是从 one-hot vectors 上去看，他们两个的向量是正交的，不相关的，这种表达就会带来不便。

motel = [1 0 0 0 ...]　　hotel = [0 1 0 0 ...]　　motel * hotel = 0

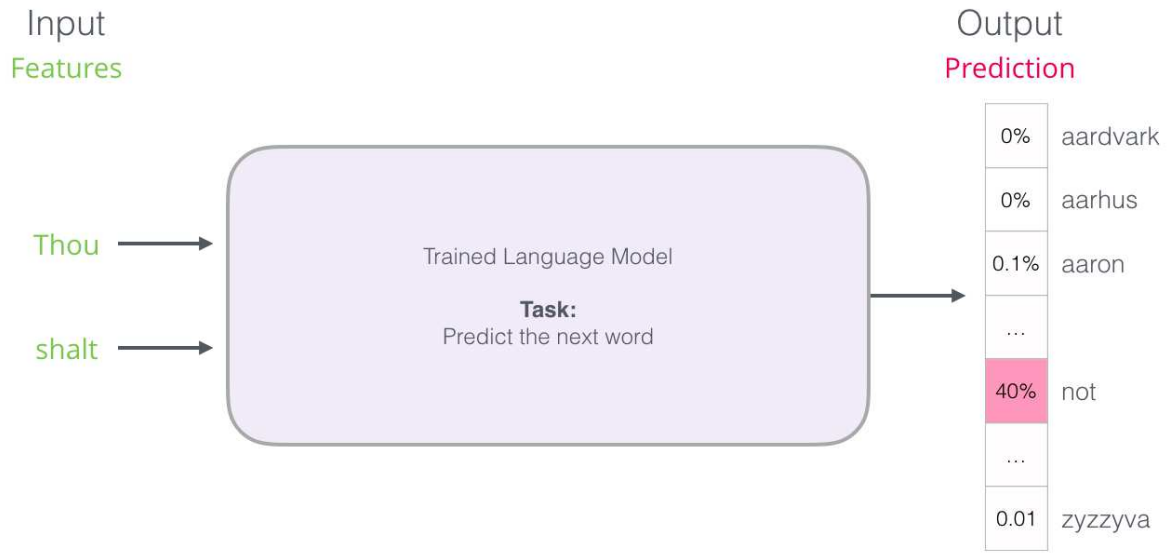
用分布式的词语向量代替 one-hot vectors，被叫做 **distributed representation**。其特点是维度较低 (使用 embedding matrix 降维)、每一个元素是连续的，所以某一个单词的向量不再受到唯一一个元素的限制。

两个词近似应该从这两个词的上下文去考虑，上下文接近，这两个词就比较接近。反过来说，两个词语义接近，那么这两个词的上下文也比较接近。通常维度越高（谷歌提供的是 300 维），提供的信息就越多，计算结果的可信度也就越高。

如果要训练一个词向量模型，中间过程交给神经网络，输入输出比较关键。我们希望让神经网络学习到上下文的前后关系。比如在“在词向量模型中输入和输出是什么？”这句话中，我们期望输入“词向量”时可以输出它的下文“模型”。

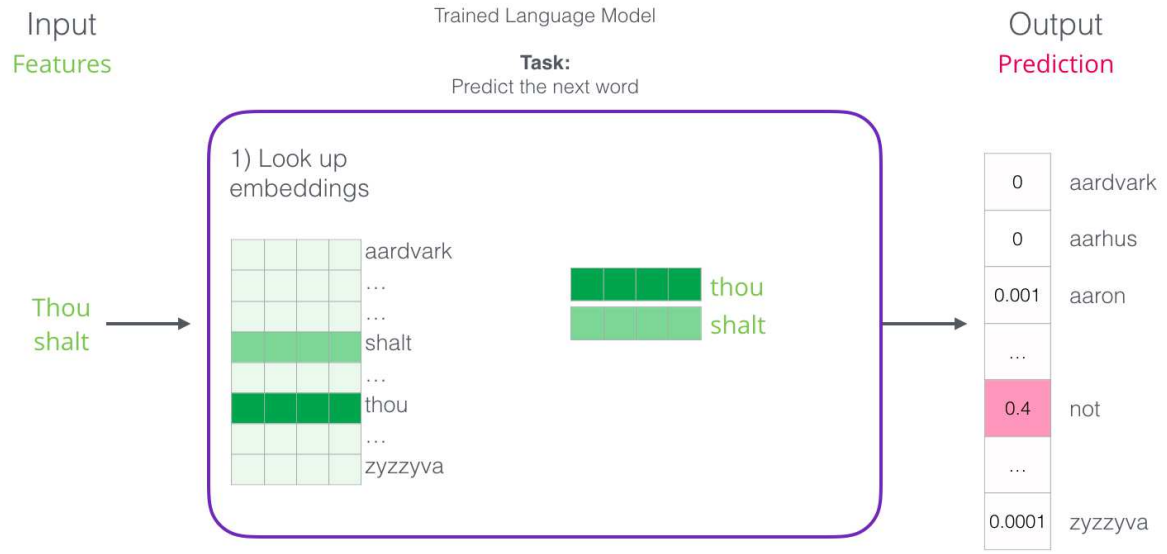


那么，我们可以输出一个预测单词的概率分布，类似图像识别的softmax分类。



显然，神经网络的输入不会是简单的词，而是在库中找到它们的词向量，关键的函数是**tf.nn.embedding_lookup**, 它把索引转换成向量矩阵，这个矩阵一开始是随机初始化的，之后的神经网络在后向传播时，不仅会更新权重还会更新这个矩阵里的向量值。**即要做的是：词向量矩阵应该如何表达才能使神经网络预测出的词更加准确。**

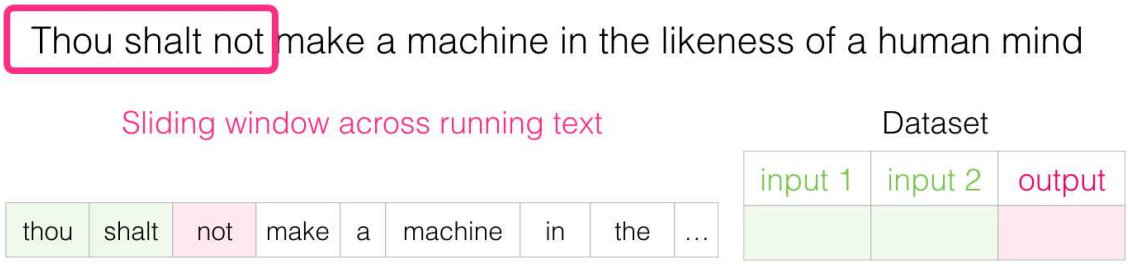
$$\frac{\partial E}{\partial w} \quad and \quad \frac{\partial E}{\partial x}$$



词向量矩阵又叫做词嵌入 (Word Embedding) 矩阵，为了存储词嵌入，我们需要一个 ***V*D*** 矩阵，其中 ***V*** 是词汇量大小，***D*** 是词嵌入的维度（即向量中表示单个单词的元素数量）。***D*** 是用户定义的超参数，***D*** 越大，学习到的词嵌入表达力越强。该矩阵将称为嵌入空间或嵌入层。

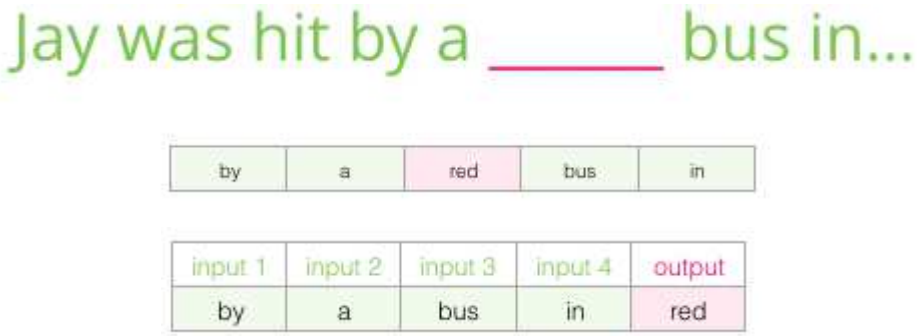
对于一个包含10万个单词的库，一个单词的表达维度是 [1, 10万]，如果用一个 [10万， 128] 的矩阵与其相乘，就可以**降维**到 [1, 128]。

数据样本可以来源于任何符合人类语言规范的文本，在数据样本的构建时，使用滑动窗口，可以使前一次的输出变成下一次的输入。第一次的“Thou”和“shalt”是输入，“not”是输出，在下一次滑动的时候，“shalt”和“not”就是输入了，“make”变成了输出。

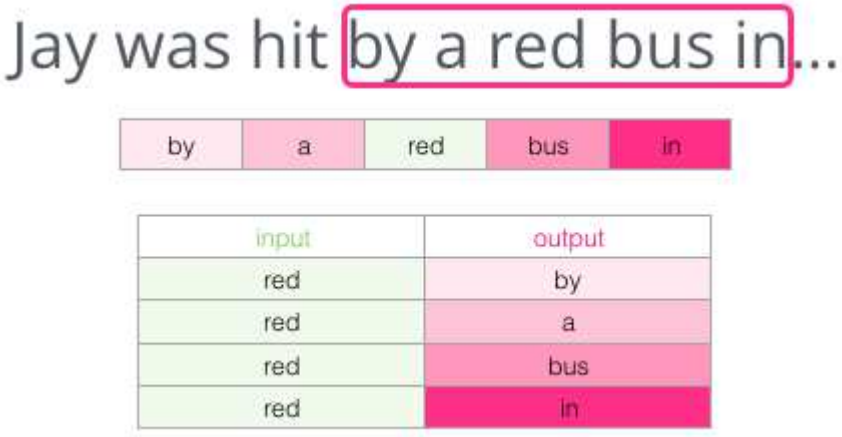


不同的 Word2vec 模型，CBOW和Skipgram，CBOW的输入是上下文，输出是中间的词。Skipgram输入的是中间词，输出的是上下文，不过构建的是多组数据。

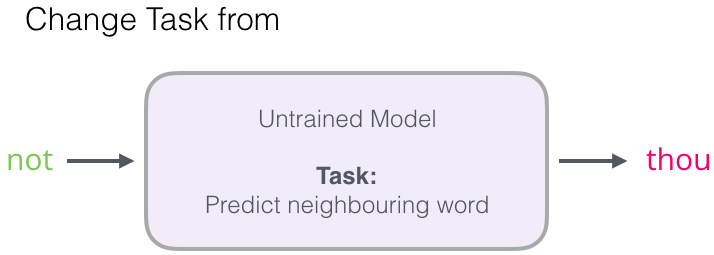
CBOW



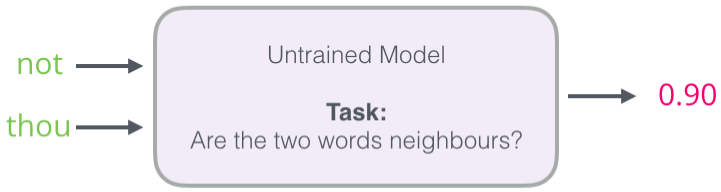
Skipgram



也有其他的输入输出方案，之前是输入前文，输出下文，也可以输入上文和下文，输出的是预测上文的下文是输入的概率，越接近 1 越好。但是这样做的话在构建数据集的时候 target 就全部是 1 了，非常类似我之前工作的逆向设计问题，非唯一解！无法收敛！其改进方案是在语料库中人为的增加一些**负采样样本**，这些词之前没有在语料库中出现过，那么它的标签值就是 0 。这样做的好处是把之前的万量级的分类任务转换成了二分类任务。现阶段的 word2vec 任务全是用这种方式训练的。一般一个单词附加5个负采样样本。



To:



input word	output word	target
not	thou	1
not		0
not		0
not	shalt	1
not	make	1

Negative examples

Skipgram

shalt	not	make	a	machine

input	output
make	shalt
make	not
make	a
make	machine

Negative Sampling

input word	output word	target
make	shalt	1
make	aaron	0
make	taco	0

数学原理

https://wmathor.com/index.php/archives/1430/

跳字模型 （skip-gram）

在跳字模型中，使用一个词来预测上下文的词，假如文本序列是 “I", "love ”, "all", "of", "you" ，设定背景窗口大小是 2，跳字模型关注的是，当选中中心词时，预测为上下文的概率，即

$$P(“I\ ”,”\ love”,”\ of\ ”,”\ you\ ”\ |\ ”\ all\ ”)$$

假定背景词的生成又是独立的，上式可以改写成，

$$P(“I\ ”\ |“all”)\cdot P(“love\ ”\ |“all”)\cdot P(”\ of\ ”\ |“all”)\cdot P(“you\ ”\ |“all”)$$

假定词典大小为 |V|，将词典中的每一个词与 0 到 |V| - 1 的整数一一对应，构成词在词典中的位置索引。给定一个长度为 T 的文本序列，t 时刻的词为 $\omega^{(t)}$ 。当时间窗口大小为 m 时，**跳字模型需要最大化给定任意中间词生成背景词的概率**：

$$\prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} P(\omega^{(t+j)} | \omega^{(t)})$$

上式得最大似然估计与**最小化**以下损失函数等价，

$$-\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} log P(\omega^{(t+j)} | \omega^{(t)})$$

可以用 v 和 u 分别代表中心词的词向量和背景词的词向量，那么对于词典中一个索引为 i 的词，它本身有两个向量 v_i 和 u_i （输入矩阵*词嵌入矩阵）进行表示，**词典中的每一个词的这两个向量就是模型所需要学习的参数。**为了把模型参数植入损失函数，需要使用模型参数来表达损失函数里中心词生成背景词的概率。假设中心词的概率是相互独立，中心词 ω_c 在词典中的索引位置是 c , 背景词 ω_o 在词典中的索引为 o , 损失函数里中心词生成背景词的概率可以用 softmax 函数进行定义：

$$P(\omega_o | \omega_c) = \frac{exp(u_o^T v_c)}{\sum_{i \in V} exp(u_i^T v_c)}$$

当序列长度 T 比较大时，可以随机采样一个较小的序列来计算损失函数，并使用随机梯度下降 SGD(stochastic gradient descent, 随机也就是说用样本中的一个例子来近似我所有的样本，来调整 θ) 优化该损失函数。通过求导，可以计算出上式生成概率的对数关于中心词向量 v_c 的梯度为：（为何 i 从 1 开始？）

$$\begin{aligned} \frac{\partial \, log \, P(\omega_o | \omega_c)}{\partial v_c} &= \frac{\partial}{\partial v_c} log \frac{exp(u_o^T v_c)}{\sum_{i=1}^{|V|} exp(u_i^T v_c)} \\ &= \frac{\partial}{\partial v_c} log \, exp(u_o^T v_c) - \frac{\partial}{\partial v_c} log \sum_{i=1}^{|V|} exp(u_o^T v_c) \end{aligned}$$

第一部分的推导：

$$\frac{\partial}{\partial v_c} log \, exp(u_o^T v_c) = \frac{\partial}{\partial v_c} u_o^T v_c = u_o$$

第二部分的推导：

$$\begin{aligned} \frac{\partial}{\partial v_c} log \sum_{i=1}^{|V|} exp(u_i^T v_c) &= \frac{1}{\sum_{i=1}^{|V|} exp(u_x^T v_c)} \cdot \frac{\partial}{\partial v_c} \sum_{x=1}^{|V|} exp(u_x^T v_c) \\ &= \frac{1}{A} \cdot \sum_{x=1}^{|V|} \frac{\partial}{\partial v_c} exp(u_x^T v_c) \\ &= \frac{1}{A} \cdot \sum_{x=1}^{|V|} exp(u_x^T v_c) \frac{\partial}{\partial v_c} u_x^T v_c \\ &= \frac{1}{\sum_{i=1}^{|V|} exp(u_x^T v_c)} \cdot \sum_{x=1}^{|V|} exp(u_x^T v_c) u_x \\ &= \sum_{x=1}^{|V|} u_x \cdot \frac{exp(u_x^T v_c)}{\sum_{i=1}^{|V|} exp(u_i^T v_c)} \\ &= \sum_{x=1}^{|V|} P(\omega_x | \omega_c) u_x \end{aligned}$$

综上所述

$$\frac{\partial log\ P(\omega_)}{\partial v_c} = u_o - \sum_{j \in V} P(\omega_j|\omega_c)u_j$$

通过上面的梯度计算后，就可以使用随机梯度下降不断迭代更新模型参数 v_c，u_o 也可以通过这种方式更新。最终对于词典中的任一索引为 i 的词，均可以得到以该词作为中心词或者背景词的两组向量 v_i 和 u_i 。

近似训练方法

由梯度更新的公式可以看到，每计算一个参数的梯度都要遍历整个字典，与 V 的大小成正比，为了减少训练的开销，可以使用近似方法来节省，包括负采样和层序 softmax。

负采样

“I want a glass of orange juice to drink” 窗口为 “glass of orange juice to ”

中心词是 orange，target 是 juice

从字典中随机抽取几个噪声词，从 V 分类变成 2（正负样本） 分类

context	word	label	
orange	juice	1	(正样本，希望 P(D=1| orange, juice) 越大越好*)
orange	king	0	(负样本)
orange	book	0	(负样本)
orange	the	0	(负样本)
orange	of	0*	(负样本，希望 P(D=0| orange, of) 越大越好*)

- P(D=1| orange, juice)：orange 和 juice 出现在正样本里的概率
- 尽管 “of” 在 window 里，但是不是这个样本的 target
- P(D=0| orange, of)：orange 和 of 出现在负样本里的概率

以跳字模型为例，之所以 V 会出现在目标函数中是因为中心词 w_c 生成背景词 w_o 的概率 $P(\omega_o|\omega_c)$ 使用了 softmax，m个target会产生m*(V-1)个非target。可以换个角度考虑，假设中心词 w_c 生成背景词 w_o 由以下两个互相独立的联合事件组成来近似：

1. 中心词 w_c 和背景词 w_o 同时出现在该训练数据窗口
2. 中心词 w_c 和噪声词不同时出现在该训练数据窗口

   中心词 w_c 和第一个噪声词 w_1 不同时出现在该训练数据窗口

   .....

   中心词 w_c 和第一个噪声词 w_K 不同时出现在该训练数据窗口

   （噪声词按照噪声词分布 $P(\omega)$ 随机生成）

   使用 sigmoid函数， $\sigma(x) = \frac{1}{1+exp(-x)}$ 来表达中心词 w_c 和背景词 ω_o 同时出现在训练数据窗口的概率：

$$P(D=1|\omega_0, \omega_c) = \sigma(u_o^T v_c)$$

中心词和噪声词不同时出现在训练数据窗口（出现在负样本）的概率：

$$P(D=0|\omega_0, \omega_k) = \sigma(u_o^T v_k) = 1 - P(D=1|\omega_0, \omega_c)$$

那么中心词 w_c 生成背景词 w_o 对数概率可以近似为

$$log\ P(\omega_o|\omega_c) = log\ \frac{1}{1+exp(-u_o^T v_c)} + \sum_{k=1,\omega_k\ P(\omega)}^{K} log[1 - \frac{1}{1+exp(-u_{i_k}^T v_c)}]$$

因此，有关中心词 ω_c 生成背景词 ω_o 的损失函数是

$$-log\ P(\omega_o|\omega_c) = -log\ \frac{1}{1+exp(-u_o^T v_c)} - \sum_{k=1,\omega_k\ P(\omega)}^{K} log[\frac{1}{1+exp(u_{i_k}^T v_c)}]$$

现在，训练中每一步的梯度计算开销不再与词典大小相关，而与 K 线性相关。当 K 取较小的常数时，负采样的每一步梯度计算开销也较小。

代码

https://www.bilibili.com/video/BV14z4y19777

1. 文本预处理

```
sentences = ["jack like dog", "jack like cat", "jack like animal",
  "dog cat animal", "banana apple cat dog like", "dog fish milk like",
  "dog cat animal like", "jack like apple", "apple like", "jack like banana",
  "apple banana jack movie book music like", "cat dog hate", "cat dog like"]
word_sequence = " ".join(sentences).split()
vocab = list(set(word_sequence)) # 构建单词表
vocab_size = len(vocab)
word2idx = {w: i for i, w in enumerate(vocab)} # 构建单词和索引位置的字典
```

其中 join() 方法返回通过指定字符连接序列中元素后生成的新字符串

```
str = "-";
seq = ("a", "b", "c"); # 字符串序列
print str.join( seq );
```

set() 方法把word_sentence中的重复元素筛除

2. 模型参数

```
C = 2 # 滑动窗口的大小（一般长度）
batch_size = 8
m = 2 # 嵌入词矩阵维度
```

3. 数据预处理

```
单词表 ['jack', 'like', 'dog', 'jack', 'like', 'cat', 'jack',...]
索引值 [  10,     11,     7,     10,     11,     3,     10 ,...]
```

那么中心词 center 的索引位置是 [7, 10, 11, 3, 10, ...]

对应的背景词 context 位置是 [[10, 11, 10, 11],

[11, 7, 11, 3],

[7, 10, 3, 10], ...]

```
skip_grams = []
for idx in range(C, len(sentences_list) - C):
  center = word2idx[sentences_list[idx]]
  # 在原始文本中找到每一个单词作为中心词在单词表中的索引
  context_idx = list(range(idx-C, idx)) + list(range(idx+1, idx+1+C))
  # 背景词在原始文本中的索引
  context = [word2idx[sentences_list[i]]  for i in context_idx]
```

找到背景词后，组合成输入就有CBOW和skip_gram两种方式，在skip_gram中，把中心词分别和背景词一一组合

[[7, 10],

[7, 11],

[7, 10],

[7, 11],

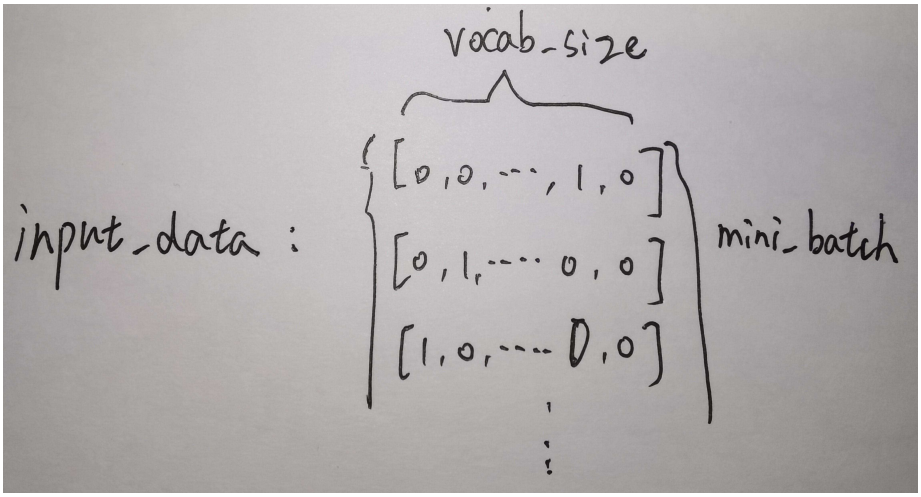
[10, 11],

[10, 7], ...]

```
skip_grams = []
for idx in range(C, len(sentences_list) - C):
  center = word2idx[sentences_list[idx]]
  # 在原始文本中找到每一个单词作为中心词在单词表中的索引
  context_idx = list(range(idx-C, idx)) + list(range(idx+1, idx+1+C))
  # 背景词在原始文本中的索引
  context = [word2idx[sentences_list[i]]  for i in context_idx]

  for w in context:
    skip_grams.append([center, w])
```

这时候的 skip_grams 包含了168个（正好是4的倍数）二维数据

在一个batch里，原始的输入是 one-hot 向量



one-hot向量可以使用单位对角阵来生成，只需要把索引值取到对应行就行

```
def make_data(skip_grams):
  input_data = []
  output_data = []
  for a, b in skip_grams:
    input_data.append(np.eye(vocab_size)[a])
    output_data.append(b)
    return input_data, output_data
```

构建数据集

```
input_data, output_data = make_data(skip_grams)
input_data, output_data = torch.Tensor(input_data),
torch.LongTensor(output_data)
dataset = Data.TensorDataset(input_data, output_data)
loader = Data.DataLoader(dataset, batch_size, shuffle=True)
```

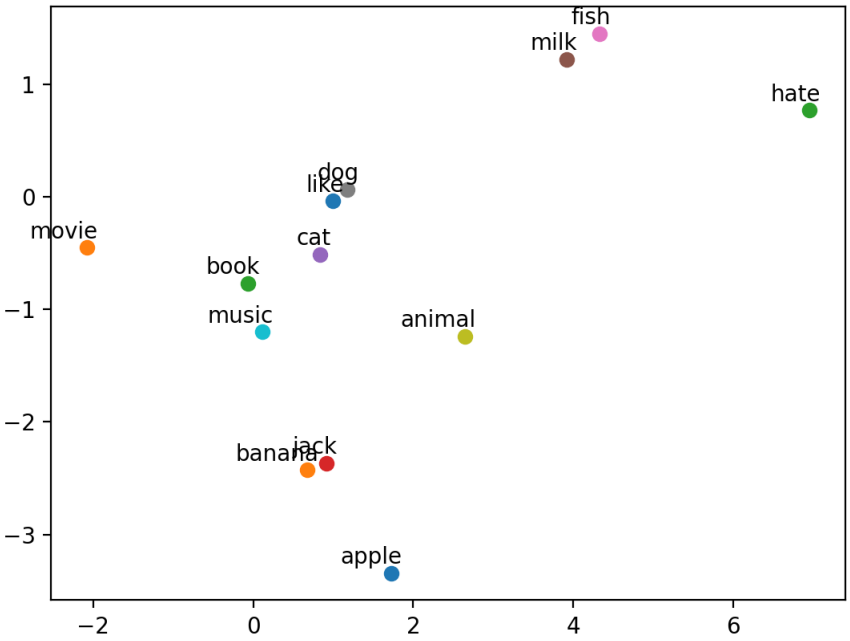
训练网络

```
# 1.
class word2vec(nn.Module):
  def __init__(self):
    super(word2vec, self).__init__()
    self.W = nn.Parameter(torch.randn(vocab_size, m).type(dtype))
    self.V = nn.Parameter(torch.randn(m, vocab_size).type(dtype))

  def forward(self, X):
    # X [batch_size, vocab_size]
    hidden = torch.mm(X, self.W)  # [batch_size, m]
    output = torch.mm(hidden, self.V) # [batch_size, vocab_size] 分类问题
    return output

# 2.
model = word2vec().to(device)
loss_fn = nn.CrossEntropyLoss().to(device)
optim = optim.Adam(model.parameters(), lr=1e-3)

# 3.
for epoch in range(2000):
  for i, (batch_x, batch_y) in enumerate(loader):
    batch_x = batch_x.to(device)
    batch_y = batch_y.to(device)
    pred = model(batch_x)
    loss = loss_fn(pred, batch_y)
    if (epoch + 1) % 100 == 0:
      print(epoch + 1, i, loss.item())

    optim.zero_grad()
    loss.backward()
    optim.step()
```

画图

```
for i, label in enumerate(vocab):
  W, WT = model.parameters()
  x,y = float(w[i][0]), float(w[i][1])
  plt.scatter(x, y)
  plt.annotate(label, xy=(x, y), xytext=(5, 2),
               textcoords='offset points', ha='right', va='bottom')
plt.show()
```

结果



疑惑与感悟

问：经过对网络模型数学原理的学习再结合之前调包调参的经验，好像在实际工程里并不是很要求数学过程的推导能力，但相关的面试中却经常问到网络的推导。那么在实际工作中对网络的数学理解是否完全必要呢?

答：取决于你想成为什么！可能工程更适合我，以后就不太会更数学原理啦。

——

我想之所以人与人之间能够相互沟通交流，不仅是因为各国人所用的语言体系一致，而且是因为构成整个世界语言体系的要素是一样的，我们有着十分类似的生活经历，使用着属性一致的生活用品，保留着从先人遗传下来的对世界的认知，即便语言种类不同，仍然可以进行一些浅层次的交流，甚至情感上的互动。从文本语言构成要素的角度去理解世界是人类智慧的体现。