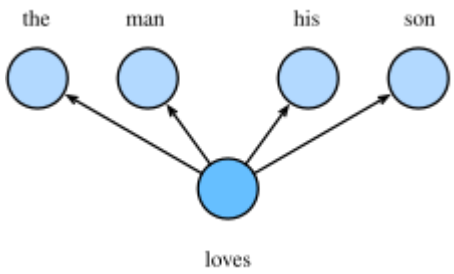
词嵌入基础

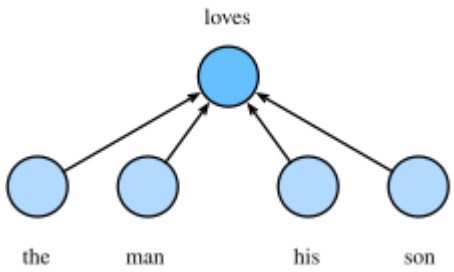
我们在“循环神经网络的从零开始实现” (https://zh.d2l.ai/chapter_recurrent-neural-networks/rnn-scratch.html)一节中使用 one-hot 向量表示单词，虽然它们构造起来很容易，但通常并不是一个好选择。一个主要的原因是，one-hot 词向量无法准确表达不同词之间的相似度，如我们常常使用的余弦相似度。

Word2Vec 词嵌入工具的提出正是为了解决上面这个问题，它将每个词表示成一个定长的向量，并通过在语料库上的预训练使得这些向量能较好地表达不同词之间的相似和类比关系，以引入一定的语义信息。基于两种概率模型的假设，我们可以定义两种 Word2Vec 模型：

1. Skip-Gram 跳字模型 (https://zh.d2l.ai/chapter_natural-language-processing/word2vec.html#%E8%B7%B3%E5%AD%97%E6%A8%A1%E5%9E%8B)：假设背景词由中心词生成，即建模 $P(w_o \mid w_c)$，其中 w_c 为中心词，w_o 为任一背景词；



1. CBOW (continuous bag-of-words) 连续词袋模型 (https://zh.d2l.ai/chapter_natural-language-processing/word2vec.html#%E8%BF%9E%E7%BB%AD%E8%AF%8D%E8%A2%8B%E6%A8%A1%E5%假设中心词由背景词生成，即建模 $P(w_c \mid \mathcal{W}_o)$，其中 $\mathcal{W}_o$ 为背景词的集合。



在这里我们主要介绍 Skip-Gram 模型的实现，CBOW 实现与其类似，读者可之后自己尝试实现。后续的内容将大致从以下四个部分展开：

1. PTB 数据集
2. Skip-Gram 跳字模型
3. 负采样近似
4. 训练模型

In [2]:

```
import collections
import math
import random
import sys
import time
import os
import numpy as np
import torch
from torch import nn
import torch.utils.data as Data
```

PTB 数据集

简单来说，Word2Vec 能从语料中学到如何将离散的词映射为连续空间中的向量，并保留其语义上的相似关系。那么为了训练 Word2Vec 模型，我们就需要一个自然语言语料库，模型将从中学习各个单词间的关系，这里我们使用经典的 PTB 语料库进行训练。PTB (Penn Tree Bank) (https://catalog.ldc.upenn.edu/LDC99T42) 是一个常用的小型语料库，它采样自《华尔街日报》的文章，包括训练集、验证集和测试集。我们将在PTB训练集上训练词嵌入模型。

载入数据集

数据集训练文件 ptb.train.txt 示例：

```
aer banknote berlitz calloway centrust cluett fromstein gitano guterman ...
pierre  N years old will join the board as a nonexecutive director nov. N
mr.  is chairman of  n.v. the dutch publishing group
...
```

In [4]:

```
with open('/home/kesci/input/ptb_train1020/ptb.train.txt', 'r') as f:
    lines = f.readlines() # 该数据集中句子以换行符为分割
    raw_dataset = [st.split() for st in lines] # st是sentence的缩写，单词以空格为分割
print('# sentences: %d' % len(raw_dataset))

# 对于数据集的前3个句子，打印每个句子的词数和前5个词
# 句尾符为 '' ，生僻词全用 '' 表示，数字则被替换成了 'N'
for st in raw_dataset[:3]:
    print('# tokens:', len(st), st[:5])
```

```
# sentences: 42068
# tokens: 24 ['aer', 'banknote', 'berlitz', 'calloway', 'centrust']
# tokens: 15 ['pierre', '<unk>', 'N', 'years', 'old']
# tokens: 11 ['mr.', '<unk>', 'is', 'chairman', 'of']
```

建立词语索引

In [5]:

```
counter = collections.Counter([tk for st in raw_dataset for tk in st]) # tk是token的缩写
counter = dict(filter(lambda x: x[1] >= 5, counter.items())) # 只保留在数据集中至少出现5次

idx_to_token = [tk for tk, _ in counter.items()]
token_to_idx = {tk: idx for idx, tk in enumerate(idx_to_token)}
dataset = [[token_to_idx[tk] for tk in st if tk in token_to_idx]
           for st in raw_dataset] # raw_dataset中的单词在这一步被转换为对应的idx
num_tokens = sum([len(st) for st in dataset])
'# tokens: %d' % num_tokens
```

Out[5]:

```
'# tokens: 887100'
```

二次采样

文本数据中一般会出现一些高频词，如英文中的“the”“a”和“in”。通常来说，在一个背景窗口中，一个词（如“chip”）和较低频词（如“microprocessor”）同时出现比和较高频词（如“the”）同时出现对训练词嵌入模型更有益。因此，训练词嵌入模型时可以对词进行二次采样。 具体来说，数据集中每个被索引词 w_i 将有一定概率被丢弃，该丢弃概率为

$$P(w_i) = \max(1 - \sqrt{\frac{t}{f(w_i)}}, 0)$$

其中 $f(w_i)$ 是数据集中词 w_i 的个数与总词数之比，常数 t 是一个超参数（实验中设为 10^{-4}）。可见，只有当 $f(w_i) > t$ 时，我们才有可能在二次采样中丢弃词 w_i，并且越高频的词被丢弃的概率越大。具体的代码如下：

In [6]:

```
def discard(idx):
    '''
    @params:
        idx: 单词的下标
    @return: True/False 表示是否丢弃该单词
    '''
    return random.uniform(0, 1) < 1 - math.sqrt(
        1e-4 / counter[idx_to_token[idx]] * num_tokens)

subsampled_dataset = [[tk for tk in st if not discard(tk)] for st in dataset]
print('# tokens: %d' % sum([len(st) for st in subsampled_dataset]))

def compare_counts(token):
    return '# %s: before=%d, after=%d' % (token, sum(
        [st.count(token_to_idx[token]) for st in dataset]), sum(
        [st.count(token_to_idx[token]) for st in subsampled_dataset]))

print(compare_counts('the'))
print(compare_counts('join'))
```

```
# tokens: 375995
# the: before=50770, after=2161
# join: before=45, after=45
```

提取中心词和背景词

In [7]:

```
def get_centers_and_contexts(dataset, max_window_size):
    '''
    @params:
        dataset: 数据集为句子的集合，每个句子则为单词的集合，此时单词已经被转换为相应数字下标
        max_window_size: 背景词的词窗大小的最大值
    @return:
        centers: 中心词的集合
        contexts: 背景词窗的集合，与中心词对应，每个背景词窗则为背景词的集合
    '''
    centers, contexts = [], []
    for st in dataset:
        if len(st) < 2:  # 每个句子至少要有2个词才可能组成一对“中心词-背景词”
            continue
        centers += st
        for center_i in range(len(st)):
            window_size = random.randint(1, max_window_size) # 随机选取背景词窗大小
            indices = list(range(max(0, center_i - window_size),
                                 min(len(st), center_i + 1 + window_size)))
            indices.remove(center_i)  # 将中心词排除在背景词之外
            contexts.append([st[idx] for idx in indices])
    return centers, contexts

all_centers, all_contexts = get_centers_and_contexts(subsampled_dataset, 5)

tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [2, 4]
center 4 has contexts [3, 5]
center 5 has contexts [4, 6]
center 6 has contexts [5]
center 7 has contexts [8]
center 8 has contexts [7, 9]
center 9 has contexts [7, 8]
```

注：数据批量读取的实现需要依赖负采样近似的实现，故放于负采样近似部分进行讲解。

Skip-Gram 跳字模型

在跳字模型中，每个词被表示成两个 d 维向量，用来计算条件概率。假设这个词在词典中索引为 i ，当它为中心词时向量表示为 $\boldsymbol{v}_i \in \mathbb{R}^d$，而为背景词时向量表示为 $\boldsymbol{u}_i \in \mathbb{R}^d$ 。设中心词 w_c 在词典中索引为 c，背景词 w_o 在词典中索引为 o，我们假设给定中心词生成背景词的条件概率满足下式：

$$P(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)}$$

PyTorch 预置的 Embedding 层

In [8]:

```
embed = nn.Embedding(num_embeddings=10, embedding_dim=4)
print(embed.weight)

x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.long)
print(embed(x))
```

```
Parameter containing:
tensor([[-0.7417, -1.9469, -0.5745,  1.4267],
        [ 1.1483,  1.4781,  0.3064, -0.2893],
        [ 0.6840,  2.4566, -0.1872, -2.2061],
        [ 0.3386,  1.3820, -0.3142,  0.2427],
        [ 0.4802, -0.6375, -0.4730,  1.2114],
        [ 0.7130, -0.9774,  0.5321,  1.4228],
        [-0.6726, -0.5829, -0.4888, -0.3290],
        [ 0.3152, -0.6827,  0.9950, -0.3326],
        [-1.4651,  1.2344,  1.9976, -1.5962],
        [ 0.0872,  0.0130, -2.1396, -0.6361]], requires_grad=True)
tensor([[[ 1.1483,  1.4781,  0.3064, -0.2893],
         [ 0.6840,  2.4566, -0.1872, -2.2061],
         [ 0.3386,  1.3820, -0.3142,  0.2427]],

        [[ 0.4802, -0.6375, -0.4730,  1.2114],
         [ 0.7130, -0.9774,  0.5321,  1.4228],
         [-0.6726, -0.5829, -0.4888, -0.3290]]], grad_fn=<EmbeddingBackward>)
```

PyTorch 预置的批量乘法

In [9]:

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
print(torch.bmm(X, Y).shape)
```

```
torch.Size([2, 1, 6])
```

Skip-Gram 模型的前向计算

In [10]:

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    '''
    @params:
        center: 中心词下标，形状为 (n, 1) 的整数张量
        contexts_and_negatives: 背景词和噪音词下标，形状为 (n, m) 的整数张量
        embed_v: 中心词的 embedding 层
        embed_u: 背景词的 embedding 层
    @return:
        pred: 中心词与背景词（或噪音词）的内积，之后可用于计算概率 p(w_o|w_c)
    '''
    v = embed_v(center) # shape of (n, 1, d)
    u = embed_u(contexts_and_negatives) # shape of (n, m, d)
    pred = torch.bmm(v, u.permute(0, 2, 1)) # bmm((n, 1, d), (n, d, m)) => shape of
    return pred
```

负采样近似

由于 softmax 运算考虑了背景词可能是词典 $\mathcal{V}$ 中的任一词，对于含几十万或上百万词的较大词典，就可能导致计算的开销过大。我们将以 skip-gram 模型为例，介绍负采样 (negative sampling) 的实现来尝试解决这个问题。

负采样方法用以下公式来近似条件概率 $P(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)}$：

$$P(w_o \mid w_c) = P(D=1 \mid w_c, w_o) \prod_{k=1, w_k \sim P(w)}^{K} P(D=0 \mid w_c, w_k)$$

其中 $P(D=1 \mid w_c, w_o) = \sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c)$，$\sigma(\cdot)$ 为 sigmoid 函数。对于一对中心词和背景词，我们从词典中随机采样 K 个噪声词（实验中设 $K=5$）。根据 Word2Vec 论文的建议，噪声词采样概率 $P(w)$ 设为 w 词频与总词频之比的 0.75 次方。

In [11]:

```
def get_negatives(all_contexts, sampling_weights, K):
    '''
    @params:
        all_contexts: [[w_o1, w_o2, ...], [...], ... ]
        sampling_weights: 每个单词的噪声词采样概率
        K: 随机采样个数
    @return:
        all_negatives: [[w_n1, w_n2, ...], [...], ...]
    '''
    all_negatives, neg_candidates, i = [], [], 0
    population = list(range(len(sampling_weights)))
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            if i == len(neg_candidates):
                # 根据每个词的权重（sampling_weights）随机生成k个词的索引作为噪声词。
                # 为了高效计算，可以将k设得稍大一点
                i, neg_candidates = 0, random.choices(
                    population, sampling_weights, k=int(1e5))
            neg, i = neg_candidates[i], i + 1
            # 噪声词不能是背景词
            if neg not in set(contexts):
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

sampling_weights = [counter[w]**0.75 for w in idx_to_token]
all_negatives = get_negatives(all_contexts, sampling_weights, 5)
```

注：除负采样方法外，还有层序 softmax (hiererarchical softmax) 方法也可以用来解决计算量过大的问题，请参考原书10.2.2 节 (https://zh.d2l.ai/chapter_natural-language-processing/approx-training.html#%E5%B1%82%E5%BA%8Fsoftmax)。

批量读取数据

In [12]:

```
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, centers, contexts, negatives):
        assert len(centers) == len(contexts) == len(negatives)
        self.centers = centers
        self.contexts = contexts
        self.negatives = negatives

    def __getitem__(self, index):
        return (self.centers[index], self.contexts[index], self.negatives[index])

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

def batchify(data):
    '''
    用作DataLoader的参数collate_fn
    @params:
        data: 长为batch_size的列表，列表中的每个元素都是__getitem__得到的结果
    @outputs:
        batch: 批量化后得到 (centers, contexts_negatives, masks, labels) 元组
            centers: 中心词下标，形状为 (n, 1) 的整数张量
            contexts_negatives: 背景词和噪声词的下标，形状为 (n, m) 的整数张量
            masks: 与补齐相对应的掩码，形状为 (n, m) 的0/1整数张量
            labels: 指示中心词的标签，形状为 (n, m) 的0/1整数张量
    '''
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)] # 使用掩码变量mask来避免填
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
        batch = (torch.tensor(centers).view(-1, 1), torch.tensor(contexts_negatives
            torch.tensor(masks), torch.tensor(labels))
    return batch

batch_size = 512
num_workers = 0 if sys.platform.startswith('win32') else 4

dataset = MyDataset(all_centers, all_contexts, all_negatives)
data_iter = Data.DataLoader(dataset, batch_size, shuffle=True,
                            collate_fn=batchify,
                            num_workers=num_workers)
for batch in data_iter:
    for name, data in zip(['centers', 'contexts_negatives', 'masks',
                           'labels'], batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: torch.Size([512, 1])
contexts_negatives shape: torch.Size([512, 60])
masks shape: torch.Size([512, 60])
labels shape: torch.Size([512, 60])
```

训练模型

损失函数

应用负采样方法后，我们可利用最大似然估计的对数等价形式将损失函数定义为如下

$$\sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} [-\log P(D=1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim P(w)^K} \log P(D=0 \mid w^{(t)}, w_k)]$$

根据这个损失函数的定义，我们可以直接使用二元交叉熵损失函数进行计算：

In [ ]:

```
class SigmoidBinaryCrossEntropyLoss(nn.Module):
    def __init__(self):
        super(SigmoidBinaryCrossEntropyLoss, self).__init__()
    def forward(self, inputs, targets, mask=None):
        '''
        @params:
            inputs: 经过sigmoid层后为预测D=1的概率
            targets: 0/1向量，1代表背景词，0代表噪音词
        @return:
            res: 平均到每个label的loss
        '''
        inputs, targets, mask = inputs.float(), targets.float(), mask.float()
        res = nn.functional.binary_cross_entropy_with_logits(inputs, targets, reduc
        res = res.sum(dim=1) / mask.float().sum(dim=1)
        return res

loss = SigmoidBinaryCrossEntropyLoss()

pred = torch.tensor([[1.5, 0.3, -1, 2], [1.1, -0.6, 2.2, 0.4]])
label = torch.tensor([[1, 0, 0, 0], [1, 1, 0, 0]]) # 标签变量label中的1和0分别代表背景词
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 1, 0]])  # 掩码变量
print(loss(pred, label, mask))

def sigmd(x):
    return - math.log(1 / (1 + math.exp(-x)))
print('%.4f' % ((sigmd(1.5) + sigmd(-0.3) + sigmd(1) + sigmd(-2)) / 4)) # 注意1-sigm
print('%.4f' % ((sigmd(1.1) + sigmd(-0.6) + sigmd(-2.2)) / 3))
```

模型初始化

In [ ]:

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(idx_to_token), embedding_dim=em
                    nn.Embedding(num_embeddings=len(idx_to_token), embedding_dim=em
```

训练模型

In [ ]:

```
def train(net, lr, num_epochs):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("train on", device)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    for epoch in range(num_epochs):
        start, l_sum, n = time.time(), 0.0, 0
        for batch in data_iter:
            center, context_negative, mask, label = [d.to(device) for d in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])

            l = loss(pred.view(label.shape), label, mask).mean() # 一个batch的平均los
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
            l_sum += l.cpu().item()
            n += 1
        print('epoch %d, loss %.2f, time %.2fs'
              % (epoch + 1, l_sum / n, time.time() - start))

train(net, 0.01, 5)
```

```
train on cpu
epoch 1, loss 0.61, time 221.30s
epoch 2, loss 0.42, time 227.70s
epoch 3, loss 0.38, time 240.50s
epoch 4, loss 0.36, time 253.79s
epoch 5, loss 0.34, time 238.51s
```

注：由于本地CPU上训练时间过长，故只截取了运行的结果，后同。大家可以自行在网站上训练。

测试模型

In [ ]:

```
def get_similar_tokens(query_token, k, embed):
    '''
    @params:
        query_token: 给定的词语
        k: 近义词的个数
        embed: 预训练词向量
    '''
    W = embed.weight.data
    x = W[token_to_idx[query_token]]
    # 添加的1e-9是为了数值稳定性
    cos = torch.matmul(W, x) / (torch.sum(W * W, dim=1) * torch.sum(x * x) + 1e-9).
    _, topk = torch.topk(cos, k=k+1)
    topk = topk.cpu().numpy()
    for i in topk[1:]:  # 除去输入词
        print('cosine sim=%.3f: %s' % (cos[i], (idx_to_token[i])))

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.446: intel
cosine sim=0.427: computer
cosine sim=0.427: computers
```

参考

- Dive into Deep Learning (https://d2l.ai/chapter_natural-language-processing/word2vec.html). Ch14.1-14.4.
- 动手学深度学习 (http://zh.gluon.ai/chapter_natural-language-processing/word2vec.html). Ch10.1-10.3.
- Dive-into-DL-PyTorch on GitHub (https://github.com/ShusenTang/Dive-into-DL-PyTorch/blob/master/code/chapter10_natural-language-processing/10.3_word2vec-pytorch.ipynb)