10.3 word2vec的实现

本节是对前两节内容的实践。我们以10.1节（词嵌入word2vec）中的跳字模型和10.2节（近似训练）中的负采样为例，介绍在语料库上训练词嵌入模型的实现。我们还会介绍一些实现中的技巧，如二次采样（subsampling）。

首先导入实验所需的包或模块。

```
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

sys.path.append("..")
import d2lzh_pytorch as d2l
print(torch.__version__)
```

10.3.1 处理数据集

PTB（Penn Tree Bank）是一个常用的小型语料库 [1]。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。该数据集的每一行作为一个句子。句子中的每个词由空格隔开。

确保 `ptb.train.txt` 已经放在了文件夹 `../../data/ptb` 下。

```
assert 'ptb.train.txt' in os.listdir("../../data/ptb")

with open('../../data/ptb/ptb.train.txt', 'r') as f:
    lines = f.readlines()
    # st是sentence的缩写
    raw_dataset = [st.split() for st in lines]
```

```
'# sentences: %d' % len(raw_dataset) # 输出 '# sentences: 42068'
```

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。这个数据集中句尾符为"<eos>"，生僻词全用"<unk>"表示，数字则被替换成了"N"。

```
for st in raw_dataset[:3]:
    print('# tokens:', len(st), st[:5])
```

输出：

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

10.3.1.1 建立词语索引

为了计算简单，我们只保留在数据集中至少出现5次的词。

```
# tk是token的缩写
counter = collections.Counter([tk for st in raw_dataset for tk in st])
counter = dict(filter(lambda x: x[1] >= 5, counter.items()))
```

然后将词映射到整数索引。

```
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]
num_tokens = sum([len(st) for st in dataset])
'# tokens: %d' % num_tokens # 输出 '# tokens: 887100'
```

10.3.1.2 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 [2]。 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

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

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。

```
def discard(idx):
    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]
'# tokens: %d' % sum([len(st) for st in subsampled_dataset]) # '# tokens: 375875'
```

可以看到，二次采样后我们去掉了一半左右的词。下面比较一个词在二次采样前后出现在数据集中的次数。可见高频词“the”的采样率不足1/20。

```
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]))

compare_counts('the') # '# the: before=50770, after=2013'
```

但低频词“join”则完整地保留了下来。

```
compare_counts('join') # '# join: before=45, after=45'
```

10.3.1.3 提取中心词和背景词

我们将与中心词距离不超过背景窗口大小的词作为它的背景词。下面定义函数提取出所有中心词和它们的背景词。它每次在整数1和 `max_window_size` （最大背景窗口）之间随机均匀采样一个整数作为背景窗口大小。

```
def get_centers_and_contexts(dataset, max_window_size):
    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)
            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
```

下面我们创建一个人工数据集，其中含有词数分别为7和3的两个句子。设最大背景窗口为2，打印所有中心词和它们的背景词。

```
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)
```

输出：

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

```
center 8 has contexts [7, 9]
center 9 has contexts [7, 8]
```

实验中，我们设最大背景窗口大小为5。下面提取数据集中所有的中心词及其背景词。

```
all_centers, all_contexts = get_centers_and_contexts(subsampled_dataset, 5)
```

10.3.2 负采样

我们使用负采样来进行近似训练。对于一对中心词和背景词，我们随机采样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 [2]。

```
def get_negatives(all_contexts, sampling_weights, K):
    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)
```

10.3.3 读取数据

我们从数据集中提取所有中心词 `all_centers` ，以及每个中心词对应的背景词 `all_contexts` 和噪声词 `all_negatives` 。我们先定义一个 `Dataset` 类。

```
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)
```

我们将通过随机小批量来读取它们。在一个小批量数据中，第i个样本包括一个中心词以及它所对应的n_i个背景词和m_i个噪声词。由于每个样本的背景窗口大小可能不一样，其中背景词与噪声词个数之和$n_i + m_i$也会不同。在构造小批量时，我们将每个样本的背景词和噪声词连结在一起，并添加填充项0直至连结后的长度相同，即长度均为$\max_i n_i + m_i$（ `max_len` 变量）。为了避免填充项对损失函数计算的影响，我们构造了掩码变量 `masks` ，其每一个元素分别与连结后的背景词和噪声词 `contexts_negatives` 中的元素一一对应。当 `contexts_negatives` 变量中的某个元素为填充项时，相同位置的掩码变量 `masks` 中的元素取0，否则取1。为了区分正类和负类，我们还需要将 `contexts_negatives` 变量中的背景词和噪声词区分开来。依据掩码变量的构造思路，我们只需创建与 `contexts_negatives` 变量形状相同的标签变量 `labels` ，并将与背景词（正类）对应的元素设1，其余清0。

下面我们实现这个小批量读取函数 `batchify` 。它的小批量输入 `data` 是一个长度为批量大小的列表，其中每个元素分别包含中心词 `center` 、背景词 `context` 和噪声词 `negative` 。该函数返回的小批量数据符合我们需要的格式，例如，包含了掩码变量。

```
def batchify(data):
    """用作DataLoader的参数collate_fn: 输入是个长为batchsize的list,
    list中的每个元素都是Dataset类调用__getitem__得到的结果
    """
    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)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (torch.tensor(centers).view(-1, 1), torch.tensor(contexts_negatives),
            torch.tensor(masks), torch.tensor(labels))
```

我们用刚刚定义的 `batchify` 函数指定 `DataLoader` 实例中小批量的读取方式，然后打印读取的第一个批量中各个变量的形状。

```
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
```

输出：

```
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])
```

10.3.4 跳字模型

我们将通过使用嵌入层和小批量乘法来实现跳字模型。它们也常常用于实现其他自然语言处理的应用。

10.3.4.1 嵌入层

获取词嵌入的层称为嵌入层，在PyTorch中可以通过创建 `nn.Embedding` 实例得到。嵌入层的权重是一个矩阵，其行数为词典大小（ `num_embeddings` ），列数为每个词向量的维度（ `embedding_dim` ）。我们设词典大小为20，词向量的维度为4。

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
embed.weight
```

输出：

```
Parameter containing:
tensor([[-0.4689,  0.2420,  0.9826, -1.3280],
        [-0.6690,  1.2385, -1.7482,  0.2986],
        [ 0.1193,  0.1554,  0.5038, -0.3619],
        [-0.0347, -0.2806,  0.3854, -0.8600],
        [-0.6479, -1.1424, -1.1920,  0.3922],
        [ 0.6334, -0.0703,  0.0830, -0.4782],
        [ 0.1712,  0.8098, -1.2208,  0.4169],
        [-0.9925,  0.9383, -0.3808, -0.1242],
        [-0.3762,  1.9276,  0.6279, -0.6391],
        [-0.8518,  2.0105,  1.8484, -0.5646],
        [-1.0699, -1.0822, -0.6945, -0.7321],
        [ 0.4806, -0.5945,  1.0795,  0.1062],
        [-1.5377,  1.0420,  0.4325,  0.1098],
        [-0.8438, -1.4104, -0.9700, -0.4889],
        [-1.9745, -0.3092,  0.6398, -0.4368],
        [ 0.0484, -0.8516, -0.4955, -0.1363],
        [-2.6301, -0.7091,  2.2116, -0.1363],
        [-0.2025,  0.8037,  0.4906,  1.5929],
        [-0.6745, -0.8791, -0.9220, -0.8125],
        [ 0.2450,  1.9456,  0.1257, -0.3728]], requires_grad=True)
```

嵌入层的输入为词的索引。输入一个词的索引i，嵌入层返回权重矩阵的第i行作为它的词向量。下面我们将形状为(2, 3)的索引输入进嵌入层，由于词向量的维度为4，我们得到形状为(2, 3, 4)的词向量。

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

输出：

```
tensor([[[-0.6690,  1.2385, -1.7482,  0.2986],
         [ 0.1193,  0.1554,  0.5038, -0.3619],
         [-0.0347, -0.2806,  0.3854, -0.8600]],

        [[-0.6479, -1.1424, -1.1920,  0.3922],
         [ 0.6334, -0.0703,  0.0830, -0.4782],
         [ 0.1712,  0.8098, -1.2208,  0.4169]]], grad_fn=<EmbeddingBackward>)
```

10.3.4.2 小批量乘法

我们可以使用小批量乘法运算 `bmm` 对两个小批量中的矩阵一一做乘法。假设第一个小批量中包含n个形状为$a \times b$的矩阵$\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$，第二个小批量中包含$n$个形状为$b \times c$的矩阵$\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$。这两个小批量的矩阵乘法输出为$n$个形状为$a \times c$的矩阵$\boldsymbol{X}_1\boldsymbol{Y}_1, \ldots, \boldsymbol{X}_n\boldsymbol{Y}_n$。因此，给定两个形状分别为$(n, a, b)$和$(n, b, c)$的 `Tensor` ，小批量乘法输出的形状为$(n, a, c)$。

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

输出：

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

10.3.4.3 跳字模型前向计算

在前向计算中，跳字模型的输入包含中心词索引 `center` 以及连结的背景词与噪声词索引 `contexts_and_negatives` 。其中 `center` 变量的形状为(批量大小, 1)，而 `contexts_and_negatives` 变量的形状为(批量大小， `max_len`)。这两个变量先通过词嵌入层分别由词索引变换为词向量，再通过小批量乘法得到形状为(批量大小, 1， `max_len`)的输出。输出中的每个元素是中心词向量与背景词向量或噪声词向量的内积。

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

10.3.5 训练模型

在训练词嵌入模型之前，我们需要定义模型的损失函数。

10.3.5.1 二元交叉熵损失函数

根据负采样中损失函数的定义，我们可以使用二元交叉熵损失函数,下面定义 `SigmoidBinaryCrossEntropyLoss` 。

```
class SigmoidBinaryCrossEntropyLoss(nn.Module):
    def __init__(self): # none mean sum
        super(SigmoidBinaryCrossEntropyLoss, self).__init__()
    def forward(self, inputs, targets, mask=None):
        """
        input – Tensor shape: (batch_size, len)
        target – Tensor of the same shape as input
        """
        inputs, targets, mask = inputs.float(), targets.float(), mask.float()
        res = nn.functional.binary_cross_entropy_with_logits(inputs, targets, reducti
        return res.mean(dim=1)

loss = SigmoidBinaryCrossEntropyLoss()
```

值得一提的是，我们可以通过掩码变量指定小批量中参与损失函数计算的部分预测值和标签：当掩码为1时，相应位置的预测值和标签将参与损失函数的计算；当掩码为0时，相应位置的预测值和标签则不参与损失函数的计算。我们之前提到，掩码变量可用于避免填充项对损失函数计算的影响。

```
pred = torch.tensor([[1.5, 0.3, -1, 2], [1.1, -0.6, 2.2, 0.4]])
# 标签变量label中的1和0分别代表背景词和噪声词
```

```
label = torch.tensor([[1, 0, 0, 0], [1, 1, 0, 0]])
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 1, 0]])  # 掩码变量
loss(pred, label, mask) * mask.shape[1] / mask.float().sum(dim=1)
```

输出：

```
tensor([0.8740, 1.2100])
```

作为比较，下面将从零开始实现二元交叉熵损失函数的计算，并根据掩码变量 `mask` 计算掩码为1的预测值和标签的损失。

```
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-sigmo
print('%.4f' % ((sigmd(1.1) + sigmd(-0.6) + sigmd(-2.2)) / 3))
```

输出：

```
0.8740
1.2100
```

10.3.5.2 初始化模型参数

我们分别构造中心词和背景词的嵌入层，并将超参数词向量维度 `embed_size` 设置成100。

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

10.3.5.3 定义训练函数

下面定义训练函数。由于填充项的存在，与之前的训练函数相比，损失函数的计算稍有不同。

```
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])

            # 使用掩码变量mask来避免填充项对损失函数计算的影响
            l = (loss(pred.view(label.shape), label, mask) *
                 mask.shape[1] / mask.float().sum(dim=1)).mean() # 一个batch的平均loss
            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))
```

现在我们就可以使用负采样训练跳字模型了。

```
train(net, 0.01, 10)
```

输出：

```
train on cpu
epoch 1, loss 1.97, time 74.53s
epoch 2, loss 0.62, time 81.85s
epoch 3, loss 0.45, time 74.49s
epoch 4, loss 0.39, time 72.04s
```

```
epoch 5, loss 0.37, time 72.21s
epoch 6, loss 0.35, time 71.81s
epoch 7, loss 0.34, time 72.00s
epoch 8, loss 0.33, time 74.45s
epoch 9, loss 0.32, time 72.08s
epoch 10, loss 0.32, time 72.05s
```

10.3.6 应用词嵌入模型

训练好词嵌入模型之后，我们可以根据两个词向量的余弦相似度表示词与词之间在语义上的相似度。可以看到，使用训练得到的词嵌入模型时，与词“chip”语义最接近的词大多与芯片有关。

```
def get_similar_tokens(query_token, k, embed):
    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).sc
    _, 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])
```

输出：

```
cosine sim=0.478: hard-disk
cosine sim=0.446: intel
cosine sim=0.440: drives
```

小结

- 可以使用PyTorch通过负采样训练跳字模型。
- 二次采样试图尽可能减轻高频词对训练词嵌入模型的影响。

- 可以将长度不同的样本填充至长度相同的小批量，并通过掩码变量区分非填充和填充，然后只令非填充参与损失函数的计算。