词向量训练实验报告：基于 PyTorch 的中英文词向量训练（CBOW 模型）

陈子扬 人工智能学院 计算机应用技术 202528020629003

一、实验目的

本实验旨在通过使用 **PyTorch 深度学习框架**，实现一个基于 **CBOW（Continuous Bag of Words）模型** 的词向量训练系统。

通过对中文语料（`zh.txt`）和英文语料（`en.txt`）分别进行训练，生成对应的词向量文件（`.pt` 格式），以理解词向量的基本概念、CBOW 的模型结构以及神经网络训练的核心过程。

二、模型原理

1. Word2Vec 概述

Word2Vec 是一种将离散单词表示为连续向量的技术，使得语义相似的词在向量空间中距离更近。

它常用的两种训练结构是：

- **CBOW**：根据上下文预测中心词
- **Skip-gram**：根据中心词预测上下文

本实验采用 **CBOW 模型**。

2. CBOW 模型原理

在 CBOW 模型中：

- **输入**：上下文中左右各 `CONTEXT_SIZE` 个词
- **输出**：中心目标词

模型结构如下：

输入层（上下文词） → 嵌入层（词向量映射） → 平均层 → 全连接层 → 输出层（softmax预测目标词）

目标是最大化在给定上下文的情况下预测正确目标词的概率。

3. 损失函数

本实验使用 **交叉熵损失函数（CrossEntropyLoss）**，用于衡量模型输出与目标词索引之间的差异。

优化器采用 **Adam**，以自动调整学习率并加快收敛速度。

解题思路：

这种方法的思路源于Word2Vec模型中常见的两种训练方式之一（CBOW与Skip-gram）。在任务中，要求我们利用**上下文词**来预测目标词的词向量，正是CBOW（Continuous Bag of Words，连续词袋模型）模型的核心思路。

解题方式的选择

词向量学习的基本目标

- 我们的最终目标是学习一个好的词向量表示，使得**相似的词在向量空间中也能保持接近**。
- 通过使用大规模语料进行训练，模型可以将每个词映射到一个高维向量空间，并通过训练让这些向量捕捉到词语的**语义相似性**（如“狗”和“猫”的词向量应该相近，King-Man+Woman=Queen）。

为什么选择CBOW模型?

- **简单且高效**：CBOW模型在小语料和训练数据较少的情况下，往往能够有效地学习到词向量，因为它的训练过程是基于上下文信息的平均，可以平滑掉一些稀有词的影响。而Skip-gram模型通常需要更大的语料和更多的计算资源，且对于数据量小的语料，可能表现不如CBOW。
- **适合给定的小语料**：任务中的语料规模较小（尤其是中文语料和英文语料），CBOW模型能够较好地处理小语料训练，尤其是在单词频次较低的情况下。它通过上下文对目标词进行预测，减少了对稀有词的需求。
- **上下文窗口的优势**：CBOW通过利用上下文词来预测目标词，非常符合我们在实际任务中需要“上下文感知”的特点。在很多自然语言处理任务中，词语的含义往往

依赖于它的上下文，这使得CBOW的训练方法十分适合。

如何进行解题？

- **步骤一**：我们首先需要读取语料，并进行预处理（如分词、去除低频词等）。然后根据给定的语料构建**词汇表**，并为每个单词分配一个唯一的ID。
- **步骤二**：根据上下文窗口大小（`CONTEXT_SIZE`），从语料中提取上下文词对目标词的训练数据。对于每一个目标词，它的上下文词可以作为输入，而目标词本身作为输出。
- **步骤三**：设计并训练CBOW模型。CBOW模型通过**平均上下文词的词向量**来预测目标词的词向量。
- **步骤四**：训练完成后，我们得到每个词的词向量。这些词向量可以用于后续的文本分析任务，如词义相似度计算、文本分类等。

三、实验设计与实现

代码解析

1. 库和模块导入

```
import torch
import torch.nn as nn
import torch.optim as optim
from collections import Counter
import random
from tqdm import tqdm
```

▼ 模块解析

- **torch**：PyTorch的核心库，用于定义和训练神经网络。
- **torch.nn**：PyTorch中的神经网络模块，包含了许多常用的层、激活函数、损失函数等。
- **torch.optim**：PyTorch中的优化器模块，包含了常用的优化算法，如SGD和Adam等。
- **Counter**：Python的集合模块，用于计算词频。
- **random**：Python的标准库，用于随机操作，如随机洗牌。

- **tqdm**：用于显示进度条的库，特别适合循环迭代时使用。

2. 参数配置

```
CONTEXT_SIZE = 2  # 上下文窗口大小
EMBEDDING_DIM = 128  # 词向量维度
EPOCHS = 200
LR = 0.001
MIN_COUNT = 3
BATCH_SIZE = 128
```

▼ 模块解析

- `CONTEXT_SIZE`：上下文窗口的大小。即在CBOW模型中，每个目标词会用它左右`CONTEXT_SIZE`个词来预测。

  CBOW模型的原理

- `EMBEDDING_DIM`：词向量的维度，也就是神经网络中每个词的向量表示的长度。通常建议词向量维度使用**2的次方数。**

  词向量维度

- `EPOCHS`：训练的轮数。

  早停

- `LR`：学习率。

  学习率LR

- `MIN_COUNT`：词频阈值，低于这个频率的词会被忽略。

  MIN_COUNT：词频阈值

- `BATCH_SIZE`：每次训练时使用的样本数。

  BATCH_SIZE（一次性处理的样本数量）

3. 设备检测

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("=" * 50)
```

```
print(f"使用设备：{device}")
print("=" * 50)
```

▼ 模块解析

检查是否有可用的GPU。如果有，它会选择GPU进行训练（使用CUDA）；否则，会选择CPU。

4. 读取语料

```
# 如果语料没有分割
def read_corpus(path):
    with open(path, "r", encoding="utf-8") as f:
        lines = f.readlines()
    return [line.strip().split() for line in lines if line.strip()]

# 语料已分割，可简化为
def read_corpus(path):
    with open(path, "r", encoding="utf-8") as f:
        # 直接读取每行并去除空行
        return [line.strip() for line in f if line.strip()]
```

▼ 模块解析

该函数用于读取文本文件中的语料。它会逐行读取文件，将每行按空格分割成单词，返回一个包含所有句子的列表，每个句子是由多个单词组成的列表。

5. 构建词典

```
def build_vocab(corpus):
    counter = Counter([w for sent in corpus for w in sent])
    vocab = [w for w, c in counter.items() if c >= MIN_COUNT]
    word2idx = {w: i for i, w in enumerate(vocab)}
    idx2word = {i: w for w, i in word2idx.items()}
    return word2idx, idx2word
```

▼ 模块解析

`counter`是`w`(word)中的次数统计出来；

`counter.item`会把counter的字与次数转换成（次数，字）这样一个二元元组；

`vocab`会提取最终得到的`c >= MIN_COUNT`的字；

`enumerate(vocab)`会给vocab里的字添加索引；

`word2idx`就是再把元组（idx，w）转成'字':idx这种形式。

`idx2word`就是再把'字':idx转成idx:'字'这种形式。

词汇表构建 (1)

6. CBOW数据生成

```
def make_cbow_data(corpus, word2idx):
    data = []
    for sent in corpus:
        idxs = [word2idx[w] for w in sent if w in word2idx]
        for i in range(CONTEXT_SIZE, len(idxs) - CONTEXT_SIZE):
            context = idxs[i - CONTEXT_SIZE:i] + idxs[i + 1:i + CONTEXT_SIZE +
1]
            target = idxs[i]
            data.append((context, target))
    return data
```

- `make_cbow_data`函数用于将语料转换为CBOW模型的训练数据。
- 对每个句子，首先将单词转化为对应的索引。
- 对于每个词，获取它前后的`CONTEXT_SIZE`个词，作为上下文；当前词作为目标词。最后，将上下文和目标词组成一个数据对，添加到`data`列表中。

7. CBOW模型定义

```
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, 128)
        self.activation1 = nn.ReLU()
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, context):
        embeds = self.embeddings(context)
```

```
        avg_embeds = torch.mean(embeds, dim=1)
        out = self.linear1(avg_embeds)
        out = self.activation1(out)
        out = self.linear2(out)
        return out
```

- **CBOW模型**：该模型使用了一个`nn.Embedding`层来为每个单词生成一个词向量。接着，使用一个全连接层（`linear1`）将嵌入向量转换为128维，经过ReLU激活，再通过另一个全连接层（`linear2`）输出词汇大小的维度。
- `forward`方法定义了前向传播过程：
  - `embeds = self.embeddings(context)`：将上下文词转化为词向量。
  - `avg_embeds = torch.mean(embeds, dim=1)`：计算上下文词向量的平均值。
  - 通过全连接层和激活函数，最终输出一个词汇大小的预测。

8. 训练过程

```
def train_word2vec(path, lang="zh"):
    print(f"\n开始训练 {lang} 语料的 Word2Vec 模型")

    corpus = read_corpus(path)
    word2idx, idx2word = build_vocab(corpus)
    data = make_cbow_data(corpus, word2idx)
    vocab_size = len(word2idx)
    print(f"词汇量: {vocab_size}, 训练样本: {len(data)}")

    model = CBOW(vocab_size, EMBEDDING_DIM).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=LR)

    for epoch in range(EPOCHS):
        total_loss = 0
        random.shuffle(data)

        for i in tqdm(range(0, len(data), BATCH_SIZE), desc=f"Epoch {epoch +
1}/{EPOCHS}"):
            batch = data[i:i + BATCH_SIZE]
            context_batch = [c for c, _ in batch]
```

```
            target_batch = [t for _, t in batch]

            context_batch = torch.tensor(context_batch, dtype=torch.long).to(d
evice)
            target_batch = torch.tensor(target_batch, dtype=torch.long).to(devi
ce)

            outputs = model(context_batch)
            loss = criterion(outputs, target_batch)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / (len(data) / BATCH_SIZE)
        print(f"Epoch {epoch + 1}/{EPOCHS} 平均Loss: {avg_loss:.4f}")

    torch.save(model.embeddings.weight.data.cpu(), f"word2vec_{lang}.pt")
    print(f"{lang} 模型训练完成，词向量已保存至 word2vec_{lang}.pt\n")
```

- **训练函数**：该函数负责训练Word2Vec模型。
- 读取语料并构建词汇表，生成训练数据。
- 创建模型、损失函数和优化器。
- 在每个epoch中，随机打乱数据，然后按批次进行训练。计算每个批次的损失，并进行反向传播。
- 训练完成后，保存模型的词向量。

9. 主函数

```
if __name__ == "__main__":
    train_word2vec("data/zh.txt", "zh")
    train_word2vec("data/en.txt", "en")
```

- 这里的主函数调用了`train_word2vec`函数，分别训练中文（`zh.txt`）和英文（`en.txt`）的词向量模型。

结构说明：

- **Embedding层**：将词ID映射为128维向量
- **Linear层1**：将平均后的上下文向量映射到128维特征空间
- **ReLU激活**：引入非线性
- **Linear层2**：输出词汇表大小的向量（每个词的预测得分）

---

3. 数据处理流程

1. 读取语料并分词（语料已预分词）
2. 构建词典，过滤低频词（`MIN_COUNT=1`）
3. 生成训练样本 `(context, target)`
4. 按 `BATCH_SIZE` 分批训练，计算损失并反向传播
5. 每轮输出平均损失 `avg_loss`

---

4. 输出文件

程序运行后会在根目录下生成以下文件：

```
word2vec_{EPOCHS}_zh.pt   # 中文词向量
word2vec_{EPOCHS}_en.pt   # 英文词向量
```

四、实验结果与分析

1. 训练过程

在训练过程中，**Loss 从较高值逐步下降**，说明模型在不断学习上下文和目标词之间的关系。

中文和英文语料由于规模不同，Loss 收敛速度略有差异。

示例输出（节选）：

```
(homework) czy25@yons-Z690-UD-DDR4:~/NLP_word2vec$ CUDA_VISIBLE_DEVICES=0,1 python train_word2vec.py
==================================================
使用设备：cuda
==================================================

开始训练 zh 语料的 Word2Vec 模型
词汇量: 13286, 训练样本: 126004
Epoch 1/256: 100%|██████████| 985/985 [00:00<00:00, 1352.07it/s]
Epoch 1/256 平均Loss: 6.9939
Epoch 2/256: 100%|██████████| 985/985 [00:00<00:00, 2195.15it/s]
```

```
Epoch 255/256 平均Loss: 0.0632
Epoch 256/256: 100%|██████████| 985/985 [00:00<00:00, 2177.64it/s]
Epoch 256/256 平均Loss: 0.0625
zh 模型训练完成，词向量已保存至 word2vec_256_zh.pt
```

```
开始训练 en 语料的 Word2Vec 模型
词汇量: 11869, 训练样本: 160544
Epoch 1/256: 100%|██████████| 1255/1255 [00:00<00:00, 2225.81it/s]
Epoch 1/256 平均Loss: 6.4939
Epoch 2/256: 100%|██████████| 1255/1255 [00:00<00:00, 2245.14it/s]
Epoch 2/256 平均Loss: 5.6310
```

```
Epoch 255/256: 100%|██████████| 1255/1255 [00:00<00:00, 2237.04it/s]
Epoch 255/256 平均Loss: 0.1241
Epoch 256/256: 100%|██████████| 1255/1255 [00:00<00:00, 2238.19it/s]
Epoch 256/256 平均Loss: 0.1233
en 模型训练完成，词向量已保存至 word2vec_256_en.pt
```

2. 结果分析

- 训练得到的词向量能在语义上反映词与词之间的关系；
- 加载 `.pt` 文件并查看相似度（如用余弦相似度计算）时，语义接近的词距离较近；
- 中文语料由于词汇量更大，Loss 收敛速度略慢于英文。

五、问题与思考

在这次实验中我也遇到了一些值得思考的地方：

- **语料规模比较小**

  我发现语料太小会导致训练出来的词向量泛化能力比较差，比如一些罕见的词可能根本没学到语义。如果有条件的话，其实可以尝试用更大的数据集（比如 Wikipedia 或新闻语料），效果应该会更好。

- **窗口大小的影响**

  在实验中，我理解了 `CONTEXT_SIZE` 的作用，窗口大一点确实能捕捉到更广的语义关系，但同时噪声也会变多。感觉这里其实是个平衡问题，要根据语料大小和任务目标去调。

- **Embedding 维度的选择**

这次实验用了 128 维，整体效果还不错。如果设得太小语义表达不够，太大会导致训练时间变长、甚至过拟合。128 算是一个折中的选择。

- **训练停止条件**

  实验时我发现训练到后期 Loss 下降特别慢，其实可以加一个“早停”机制，比如当连续几轮 Loss 变化很小就自动停止训练，这样可以节省不少时间。

六、总结

通过本实验，我掌握了以下内容：

- 使用 **PyTorch** 搭建 **CBOW 模型**
- 理解词向量的训练原理
- 学会词典构建、样本生成、模型训练的完整流程
- 掌握超参数（学习率、窗口大小、Embedding 维度）对效果的影响

最终成功生成中英文词向量文件：

```
# 详见附件
word2vec_256_zh.pt
word2vec_256_en.pt
```
