# 作业三：实现Word2Vec的CBOW

## 作业要求

基于提供的Python文件/Jupyter Notebook文件，以代码填空的形式，实现Word2Vec的连续词袋模型（CBOW）的相关代码，填空完毕后，需展示代码中相应测试部分的输出结果。

本次作业共计15分。提示：只需填写代码中TODO标记的空缺位置即可，具体的代码说明和评分细则详见提供的脚本文件。

## 提交方式

以下两种方式选择其一提交至Canvas平台即可：

1. 补全 `w2v.ipynb` 代码后运行获得结果，并把notebook导出为 `w2v.pdf` ，将两者打包提交。
2. 补全 `w2v.py` 文件，完成实验报告，报告要求对代码作必要的说明，并展示运行结果。

## 文件说明

```
├── data
│   ├── debug.txt       # 用于debug的小语料
│   ├── synonyms.json   # 用于测试词向量的数据
│   └── treebank.txt    # 用于训练词向量的语料
├── README.md
├── w2v.ipynb
└── w2v.py
```

需要Python版本大于等于3.6，并检查是否已安装所依赖的第三方库。

In [1]:

```python
import importlib
import sys

assert sys.version_info[0] == 3
assert sys.version_info[1] >= 6

requirements = ["numpy", "tqdm"]
_OK = True

for name in requirements:
    try:
        importlib.import_module(name)
    except ImportError:
        print(f"Require: {name}")
        _OK = False

if not _OK:
    exit(-1)
else:
    print("All libraries are satisfied.")
```

```
All libraries are satisfied.
```

## 辅助代码

该部分包含：用于给句子分词的分词器 `tokenizer` 、用于构造数据的数据集类 `Dataset` 和用于构建词表的词表类 `Vocab` 。

> 注: 该部分无需实现。

### 分词器

该分词器会：

1. 将所有字母转为小写；
2. 将句子分为连续的字母序列（word）

In [2]:

```
import re
from typing import List


def tokenizer(line: str) -> List[str]:
    line = line.lower()
    tokens = list(filter(lambda x: len(x) > 0, re.split(r"\W", line)))
    return tokens


print(tokenizer("It's  useful. "))
```

```
['it', 's', 'useful']
```

### 数据集类

通过设定窗长 `window_size`，该数据集类会读取 `corpus` 中的行，并解析返回 `(context, target)` 元组。

假如一个句子序列为 `a b c d e`，且此时 `window_size=2`，`Dataset` 会返回：

```
([b, c], a)
([a, c, d], b)
([a, b, d, e], c)
([b, c, e], d)
([c, d], e)
```

In [3]:

```
class Dataset:
    def __init__(self, corpus: str, window_size: int):
        """
        :param corpus: 语料路径
        :param window_size: 窗口长度
        """
        self.corpus = corpus
        self.window_size = window_size

    def __iter__(self):
        with open(self.corpus, encoding="utf8") as f:
            for line in f:
                tokens = tokenizer(line)
                if len(tokens) <= 1:
                    continue
                for i, target in enumerate(tokens):
                    left_context = tokens[max(0, i - self.window_size): i]
                    right_context = tokens[i + 1: i + 1 + self.window_size]
                    context = left_context + right_context
                    yield context, target

    def __len__(self):
        """ 统计样本语料中的样本个数 """
        len_ = getattr(self, "len_", None)
        if len_ is not None:
            return len_

        len_ = 0
        for _ in iter(self):
            len_ += 1

        setattr(self, "len_", len_)
        return len_
```

In [4]:

```
debug_dataset = Dataset("./data/debug.txt", window_size=3)
print(len(debug_dataset))

for i, pair in enumerate(iter(debug_dataset)):
    print(pair)
    if i >= 3:
        break

del debug_dataset
```

```
50
(['want', 'to', 'go'], 'i')
(['i', 'to', 'go', 'home'], 'want')
(['i', 'want', 'go', 'home'], 'to')
(['i', 'want', 'to', 'home'], 'go')
```

## 词表类

`Vocab` 可以用 `token_to_idx` 把token(str)映射为索引(int)，也可以用 `idx_to_token` 找到索引对应的token。

实例化 `Vocab` 有两种方法：

1. 读取 `corpus` 构建词表。
2. 通过调用 `Vocab.load_vocab` ，可以从已训练的中构建 `Vocab` 实例。

In [5]:

```python
import os
import warnings
from collections import Counter
from typing import Dict, Tuple


class Vocab:
    VOCAB_FILE = "vocab.txt"
    UNK = "<unk>"

    def __init__(self, corpus: str = None, max_vocab_size: int = -1):
        """
        :param corpus:         语料文件路径
        :param max_vocab_size: 最大词表数量，-1表示不做任何限制
        """
        self._token_to_idx: Dict[str, int] = {}
        self.token_freq: List[Tuple[str, int]] = []

        if corpus is not None:
            self.build_vocab(corpus, max_vocab_size)

    def build_vocab(self, corpus: str, max_vocab_size: int = -1):
        """ 统计词频，并保留高频词 """
        counter = Counter()
        with open(corpus, encoding="utf8") as f:
            for line in f:
                tokens = tokenizer(line)
                counter.update(tokens)

        print(f"总Token数: {sum(counter.values())}")

        # 将找到的词按照词频从高到低排序
        self.token_freq = [(self.UNK, 1)] + sorted(counter.items(),
                                                   key=lambda x: x[1], reverse=True)
        if max_vocab_size > 0:
            self.token_freq = self.token_freq[:max_vocab_size]

        print(f"词表大小: {len(self.token_freq)}")

        for i, (token, _freq) in enumerate(self.token_freq):
            self._token_to_idx[token] = i

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

    def __contains__(self, token: str):
        return token in self._token_to_idx

    def token_to_idx(self, token: str, warn: bool = False) -> int:
        """ Map the token to index """
        token = token.lower()
        if token not in self._token_to_idx:
            if warn:
                warnings.warn(f"{token} => {self.UNK}")
            token = self.UNK
        return self._token_to_idx[token]

    def idx_to_token(self, idx: int) -> str:
        """ Map the index to token """
        assert 0 <= idx < len(self)
        return self.token_freq[idx][0]

    def save_vocab(self, path: str):
        with open(os.path.join(path, self.VOCAB_FILE), "w", encoding="utf8") as f:
            lines = [f"{token} {freq}" for token, freq in self.token_freq]
            f.write("\n".join(lines))

    @classmethod
    def load_vocab(cls, path: str):
        vocab = cls()

        with open(os.path.join(path, cls.VOCAB_FILE), encoding="utf8") as f:
            lines = f.read().split("\n")

        for i, line in enumerate(lines):
            token, freq = line.split()
            vocab.token_freq.append((token, int(freq)))
            vocab._token_to_idx[token] = i

        return vocab
```

In [6]:

```
debug_vocab = Vocab("./data/debug.txt")
print(debug_vocab.token_freq)
del debug_vocab
```

```
总Token数: 50
词表大小: 21
[('<unk>', 1), ('want', 6), ('to', 6), ('go', 4), ('i', 3), ('home', 3), ('play', 3), ('like', 3), ('eating', 3), ('he', 3),
('she', 3), ('it', 2), ('is', 2), ('we', 2), ('useful', 1), ('awful', 1), ('can', 1), ('read', 1), ('books', 1), ('will', 1),
('now', 1)]
```

## Word2Vec实现

本节将实现Word2Vec的CBOW模型，为了便于实现，本实验不引入 `Hierarchical Softmax` 和 `Negative Sampling` 等加速技巧，若同学们对这些技术感兴趣，可参考：[word2vec Parameter Learning Explained (https://arxiv.org/pdf/1411.2738.pdf)](https://arxiv.org/pdf/1411.2738.pdf)。

TODO: 实现one-hot向量构建函数(1分)

需求：指定词向量的维度和需要置1的索引，返回类型为 `np.ndarray` 的one-hot行向量。

In [7]:

```
import numpy as np


def one_hot(dim: int, idx: int) -> np.ndarray:
    # TODO: 实现one-hot函数（1分）
    y = np.zeros(dim)
    y[idx] = 1
    return y


print(one_hot(4, 1))
```

```
[0. 1. 0. 0.]
```

TODO：实现softmax(2分)

> 注意数值溢出的可能

In [8]:

```
def softmax(x: np.ndarray) -> np.ndarray:
    # TODO: 实现softmax函数（2分）
    x -= np.max(x)
    x = np.exp(x)
    return x / np.sum(x)


print(softmax(np.array([i for i in range(10)])))
```

```
[7.80134161e-05 2.12062451e-04 5.76445508e-04 1.56694135e-03
 4.25938820e-03 1.15782175e-02 3.14728583e-02 8.55520989e-02
 2.32554716e-01 6.32149258e-01]
```

TODO：CBOW类，请补全 `train_one_step` 中的代码。

推荐按照TODO描述的步骤来实现（预计15行代码），也可在保证结果正确的前提下按照自己的思路来实现。

> tips: 建议使用numpy的向量化操作代替Python循环。 比如同样是实现两个向量 `a` 和 `b` 的内积， `np.dot(a,b)` 的运行效率可达纯Python实现的函数的百倍以上。同样的，向量外积也推荐使用 `np.outer(a,b)` 。具体的函数功能可参考Numpy文档。

In [9]:

```
import os
import pickle
import time

from tqdm import tqdm


class CBOW:
    def __init__(self, vocab: Vocab, vector_dim: int):
        self.vocab = vocab
        self.vector_dim = vector_dim

        self.U = np.random.uniform(-1, 1, (len(self.vocab), self.vector_dim))  # vocab_size x vector_dim
        self.V = np.random.uniform(-1, 1, (self.vector_dim, len(self.vocab)))  # vector_dim x vocab_size

    def train(self, corpus: str, window_size: int, train_epoch: int, learning_rate: float, save_path: str = None):
        dataset = Dataset(corpus, window_size)
        start_time = time.time()

        for epoch in range(1, train_epoch + 1):
            self.train_one_epoch(epoch, dataset, learning_rate)
            if save_path is not None:
                self.save_model(save_path)

        end_time = time.time()
        print(f"总耗时 {end_time - start_time:.2f}s")

    def train_one_epoch(self, epoch: int, dataset: Dataset, learning_rate: float):
        steps, total_loss = 0, 0.0

        with tqdm(iter(dataset), total=len(dataset), desc=f"Epoch {epoch}", ncols=80) as pbar:
            for sample in pbar:
                context_tokens, target_token = sample
                loss = self.train_one_step(context_tokens, target_token, learning_rate)
                total_loss += loss
                steps += 1
                if steps % 10 == 0:
                    pbar.set_postfix({"Avg. loss": f"{total_loss / steps:.2f}"})

        return total_loss / steps

    def train_one_step(self, context_tokens: List[str], target_token: str, learning_rate: float) -> float:
        """
        :param context_tokens:  目标词周围的词
        :param target_token:    目标词
        :param learning_rate:   学习率
        :return:    loss值 (标量)
        """
        C = len(context_tokens)

        # TODO: 构造输入向量和目标向量（3分）
        # context: 构造输入向量
        # target:  目标one-hot向量
        context = np.zeros(len(self.vocab))
        for token in context_tokens:
            context += one_hot(len(self.vocab), self.vocab.token_to_idx(token))
        context /= C
        target = one_hot(len(self.vocab), self.vocab.token_to_idx(target_token))

        # TODO: 前向步骤（3分）
        h = np.dot(self.U.T, context)
        y = softmax(np.dot(self.V.T, h))

        # TODO: 计算loss（3分）
        loss = -np.log(y[self.vocab.token_to_idx(target_token)])

        # TODO: 更新参数（3分）
        self.U -= learning_rate * np.outer(context, np.dot(y-target, self.V.T))
        self.V -= learning_rate * np.outer(h, y - target)

        return loss

    def similarity(self, token1: str, token2: str):
        """ 计算两个词的相似性 """
        v1 = self.U[self.vocab.token_to_idx(token1)]
        v2 = self.U[self.vocab.token_to_idx(token2)]
        v1 = v1 / np.linalg.norm(v1)
        v2 = v2 / np.linalg.norm(v2)
        return np.dot(v1, v2)

    def most_similar_tokens(self, token: str, n: int):
        """ 召回与token最相似的n个token """
        norm_U = self.U / np.linalg.norm(self.U, axis=1, keepdims=True)
```

```python
        idx = self.vocab.token_to_idx(token, warn=True)
        v = norm_U[idx]

        cosine_similarity = np.dot(norm_U, v)
        nbest_idx = np.argsort(cosine_similarity)[-n:][::-1]

        results = []
        for idx in nbest_idx:
            _token = self.vocab.idx_to_token(idx)
            results.append((_token, cosine_similarity[idx]))

        return results

    def save_model(self, path: str):
        """ 将模型保存到`path`路径下，如果不存在`path`会主动创建 """
        os.makedirs(path, exist_ok=True)
        self.vocab.save_vocab(path)

        with open(os.path.join(path, "wv.pkl"), "wb") as f:
            param = {"U": self.U, "V": self.V}
            pickle.dump(param, f)

    @classmethod
    def load_model(cls, path: str):
        """ 从`path`加载模型 """
        vocab = Vocab.load_vocab(path)

        with open(os.path.join(path, "wv.pkl"), "rb") as f:
            param = pickle.load(f)

        U, V = param["U"], param["V"]
        model = cls(vocab, U.shape[1])
        model.U, model.V = U, V

        return model
```

# 测试

测试部分可用于验证CBOW实现的正确性，此部分的结果不计入总分。

## 测试1

本测试可用于调试，最终一个epoch的平均loss约为0.5，并且“i”、“he”和“she”的相似性较高。

In [10]:

```python
import random


def test1():
    random.seed(42)
    np.random.seed(42)

    vocab = Vocab(corpus="./data/debug.txt")
    cbow = CBOW(vocab, vector_dim=8)
    cbow.train(corpus="./data/debug.txt", window_size=3,
               train_epoch=10, learning_rate=1.0)

    print(cbow.most_similar_tokens("i", 5))
    print(cbow.most_similar_tokens("he", 5))
    print(cbow.most_similar_tokens("she", 5))


test1()
```

```
总Token数: 50
词表大小: 21

Epoch 1: 100%|████████████████| 50/50 [00:00<00:00, 2939.45it/s, Avg. loss=2.89]
Epoch 2: 100%|████████████████| 50/50 [00:00<00:00, 2827.95it/s, Avg. loss=1.54]
Epoch 3: 100%|████████████████| 50/50 [00:00<00:00, 3073.29it/s, Avg. loss=1.05]
Epoch 4: 100%|████████████████| 50/50 [00:00<00:00, 3460.08it/s, Avg. loss=0.82]
Epoch 5: 100%|████████████████| 50/50 [00:00<00:00, 3943.87it/s, Avg. loss=0.76]
Epoch 6: 100%|████████████████| 50/50 [00:00<00:00, 3836.02it/s, Avg. loss=0.67]
Epoch 7: 100%|████████████████| 50/50 [00:00<00:00, 4355.18it/s, Avg. loss=0.53]
Epoch 8: 100%|████████████████| 50/50 [00:00<00:00, 4001.05it/s, Avg. loss=0.54]
Epoch 9: 100%|████████████████| 50/50 [00:00<00:00, 4166.47it/s, Avg. loss=0.52]
Epoch 10: 100%|███████████████| 50/50 [00:00<00:00, 4973.44it/s, Avg. loss=0.50]

总耗时 0.19s
[('i', 1.0), ('he', 0.9925540605382075), ('she', 0.966337856762682), ('<unk>', 0.635699270231461), ('is', 0.3974123537637741)]
[('he', 1.0), ('i', 0.9925540605382075), ('she', 0.98580084006031), ('<unk>', 0.6171017925293522), ('is', 0.3582327872195807
4)]
[('she', 1.0000000000000002), ('he', 0.98580084006031), ('i', 0.966337856762682), ('<unk>', 0.5012279262065746), ('is', 0.3869
8246680231224)]
```

### 测试2

本测试将会在 `treebank.txt` 上训练词向量，为了加快训练流程，实验只保留高频的4000词，且词向量维度为20。

在每个epoch结束后，会在 `data/treebank.txt` 中测试词向量的召回能力。如下所示， `data/treebank.txt` 中每个样例为 `word` 以及对应的同义词，同义词从wordnet中获取。

```
[
  "about",
  [
    "most",
    "virtually",
    "around",
    "almost",
    "near",
    "nearly",
    "some"
  ]
]
```

> 本阶段预计消耗25分钟，具体时间与 `train_one_step` 代码实现有关

> 最后一个epoch平均loss降至5.1左右，并且在同义词上的召回率约为20%左右

In [11]:

```python
import json


def calculate_recall_rate(model: CBOW, word_synonyms: List[Tuple[str, List[str]]], topn: int) -> float:
    """ 测试CBOW的召回率 """
    hit, total = 0, 1e-9
    for word, synonyms in word_synonyms:
        synonyms = set(synonyms)
        recalled = set([w for w, _ in model.most_similar_tokens(word, topn)])
        hit += len(synonyms & recalled)
        total += len(synonyms)

    print(f"Recall rate: {hit / total:.2%}")
    return hit / total


def test2():
    random.seed(42)
    np.random.seed(42)

    corpus = "./data/treebank.txt"
    lr = 1e-1
    topn = 40

    vocab = Vocab(corpus, max_vocab_size=4000)
    model = CBOW(vocab, vector_dim=20)

    dataset = Dataset(corpus, window_size=4)

    with open("data/synonyms.json", encoding="utf8") as f:
        word_synonyms: List[Tuple[str, List[str]]] = json.load(f)

    for epoch in range(1, 11):
        model.train_one_epoch(epoch, dataset, learning_rate=lr)
        calculate_recall_rate(model, word_synonyms, topn)


test2()
```

```
总Token数: 205068
词表大小: 4000

Epoch 1: 100%|██████████| 205058/205058 [06:44<00:00, 507.12it/s, Avg. loss=5.99]

Recall rate: 8.88%

Epoch 2: 100%|██████████| 205058/205058 [06:51<00:00, 498.55it/s, Avg. loss=5.59]

Recall rate: 13.02%

Epoch 3: 100%|██████████| 205058/205058 [06:48<00:00, 501.69it/s, Avg. loss=5.44]

Recall rate: 14.20%

Epoch 4: 100%|██████████| 205058/205058 [06:49<00:00, 500.15it/s, Avg. loss=5.34]

Recall rate: 15.98%

Epoch 5: 100%|██████████| 205058/205058 [06:49<00:00, 501.01it/s, Avg. loss=5.26]

Recall rate: 17.75%

Epoch 6: 100%|██████████| 205058/205058 [07:07<00:00, 479.93it/s, Avg. loss=5.20]

Recall rate: 18.93%

Epoch 7: 100%|██████████| 205058/205058 [07:05<00:00, 481.77it/s, Avg. loss=5.15]

Recall rate: 19.23%

Epoch 8: 100%|██████████| 205058/205058 [07:06<00:00, 480.66it/s, Avg. loss=5.11]

Recall rate: 19.82%

Epoch 9: 100%|██████████| 205058/205058 [06:50<00:00, 498.93it/s, Avg. loss=5.07]

Recall rate: 20.12%

Epoch 10: 100%|██████████| 205058/205058 [06:50<00:00, 499.05it/s, Avg. loss=5.04]

Recall rate: 19.53%
```