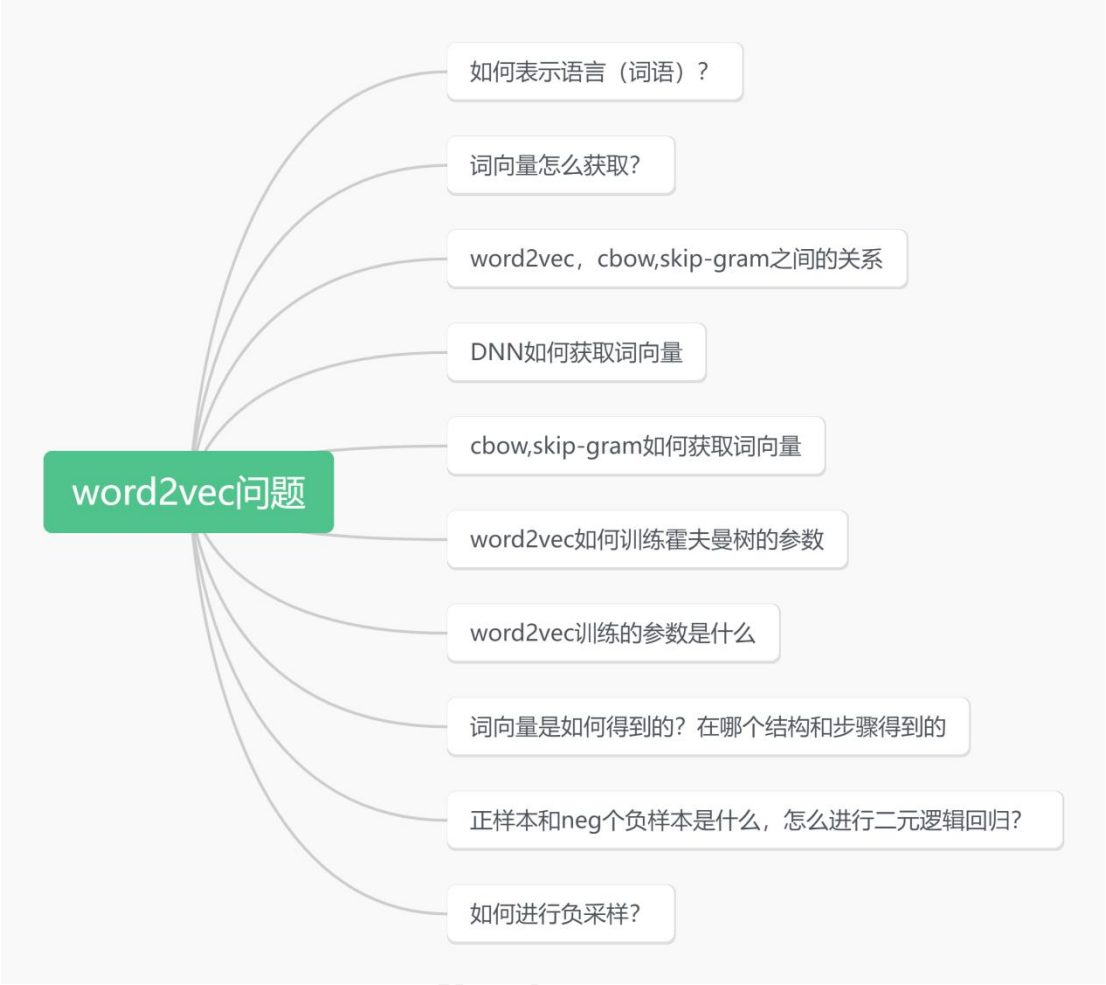


1. CBOW 与 Skip-Gram 基础模型

前面我们讲到 one-hot 编码有 2 个大的缺点：

（1）维度灾难：词汇表一般都非常大，假如词汇表有 10k 个词，那么一个词向量的长度就需要达到 10k，而其中却仅有一个位置是 1，其余全是 0，内存占用过高且表达效率很低。

（2）无法体现出**词与词之间的关系**：比如“爱”和“喜欢”这两个词，它们的意思是相近的，但基于 one-hot 编码后的结果取决于它们在词汇表中的位置，不能很好地刻画词与词之间的相似性。因为任何两个 one-hot 向量之间的内积都是 0，很难区分他们之间的差别。

也讲了词向量的形式和优势，为了生成这种词向量，研究者们提出了很多方法，这节我们讲解一下目前通用的生成词向量的模型——Google2013 年开源的一款用于词向量计算的工具——Word2Vec。

首先，Word2Vec 利用浅层神经网络可以在百万数量级的词典和上亿的数据集上进行高效地训练，不仅提高了表示的质量，还显著提升了训练速度和效率；其次，该工具训练过程中得到的词向量（word embedding），可以很好地度量词与词之间的相似性。

Word2Vec 只是一种工具的名称，其本身不生成词向量，依靠的是其背后用于计算 word vector(Continuous Bag-of-Words)的 CBOW 模型和 Skip-gram 模型。接下来让我们分别介绍这两个模型以及它们的训练方法。

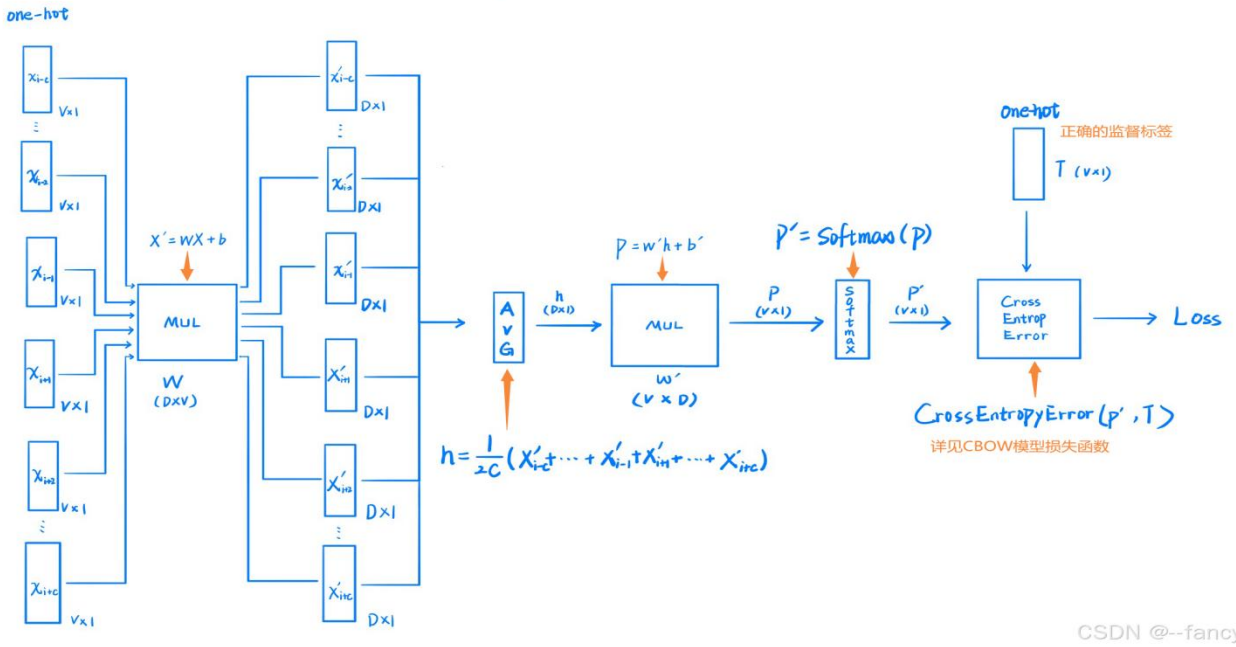
1.1 CBOW 模型

CBOW 模型，即 Continuous Bag-of-Words，顾名思义就是一袋子词语，那么如何选择这些词语呢？

Question is the best answer to study natural language processing.

context context

如上，answer 是想要学习的中心词，假设窗口大小为 3，前面的“is the best”和后面的“to study natural”就是该中心词对应的上下文，就是模型的输入。由于 CBOW 使用的是词袋模型，因此这 8 个词都是平等的，也就是不考虑和关注的词之间的距离大小，只要在上下文之内即可。



如图所示是 CBOW 模型的结构，

（1）首先关注一下输入输出，

输入：某一个中心词的上下文相关对应的单词（根据设置的窗口大小决定单词数 C=2c）的 one-hot 编码（C*V，其中 C 是两边的上下文的词语总数，V 是词汇量的长度）；

输出：中心词的 one-hot 编码（1*V）。

（2）学习一下模型的数据流的处理过程，

第一步是将上下文独热编码 X=C*V 输入模型，然后与隐藏层的参数

矩阵 W=V*D 相乘得到隐藏层 X’=XW=(C*V)(V*D)=C*D；

第二步是将隐藏层的矩阵 X’在行维度上求平均数得到向量 h=1*D；

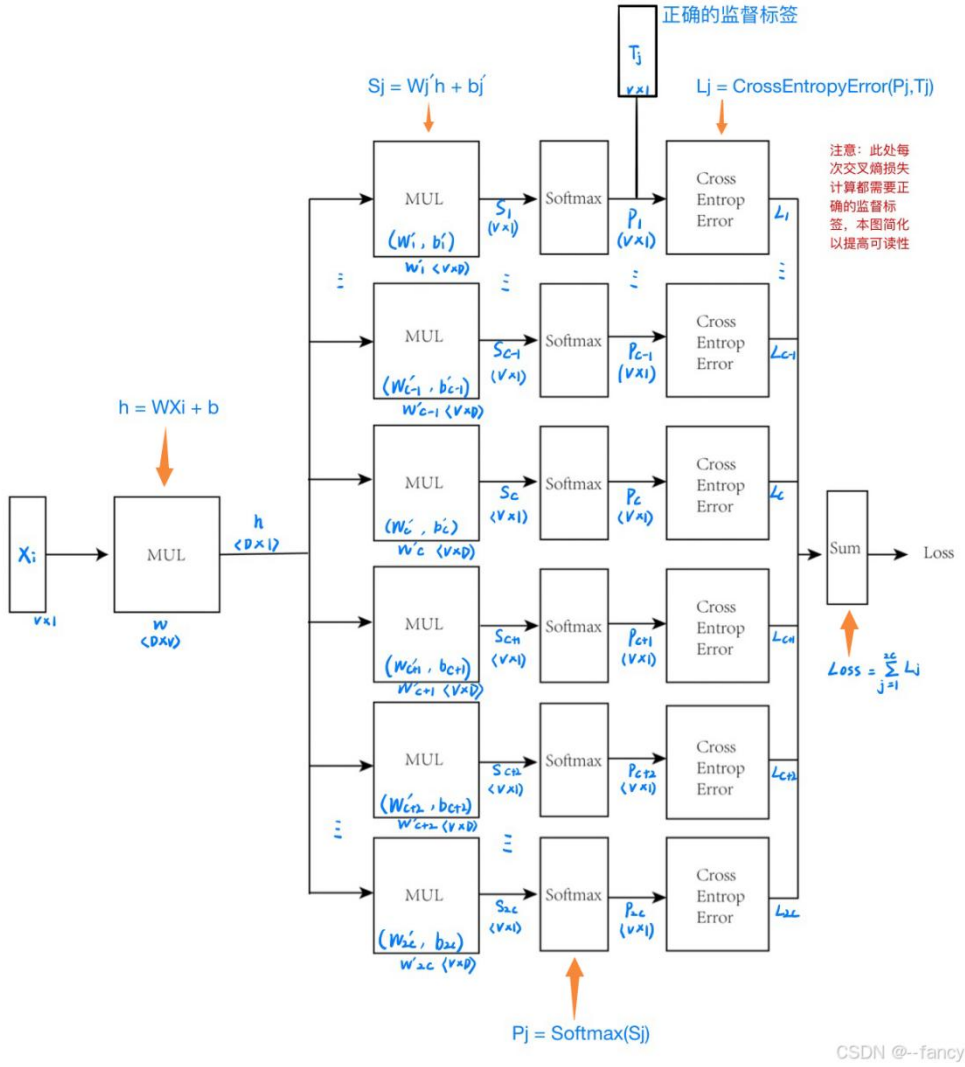
第三步是将中间向量 h=1*D 再乘上参数矩阵 W’=D*V 得到一个中间结果向量 P=hW’=(1*D)(D*V)=1*V，然后将其 softmax 归一化，这样得到词汇表中（长度为 V）所有单词的概率，值最大的就是预测得到的中心词。

第四步就是将标签值和预测值代入损失函数并利用优化算法优化中间的 2 个参数矩阵，直到满足训练要求。

（3）词向量到底在哪？了解了整个训练过程，似乎没看到词向量在哪里。事实上，词向量就是训练时第一个参数矩阵 W=V*D，而且是经过训练将词汇表中的所有 V 个词的词向量一起训练得到的。

1.2. Skip-gram 模型

Skip-Gram 模型和 CBOW 的思路是反着来的（互为镜像），即使用中心词来预测上下文词。还是上面的例子，上下文大小取值为 3， 特定的中心词"answer"是输入，而窗口内的 6 个上下文词是输出。



（1）首先关注一下输入输出，

输入是特定中心词的 one-hot 编码（1*V），

输出是中心词对应的所有上下文单词的 one-hot 编码。

（2）学习一下模型的数据流的处理过程，

第一步是将中心词独热编码 X=1*V 输入模型，然后与隐藏层的参数矩阵 W=V*D 相乘得到隐藏向量 h=XW=(1*V)(V*D)=1*D；

第二步是将隐藏向量 h 分别乘上 C 个参数矩阵 W'=V*D 得到 C 个结果向量 P=1*V，然后分别将这 C 个结果向量归一化 softmax 得到词汇表（长度为 V）中每个单词的概率；

第三步是将标签值和预测值代入损失函数并利用优化算法优化中间的 2 个参数矩阵，直到满足训练要求。

（3）词向量到底在哪？词向量同样是训练时第一个参数矩阵 W=V*D，而且是经过训练将词汇表中的所有 V 个词的词向量一起训练得到的。

2 分层 Softmax

上面我们介绍了基础的 CBOW 与 Skip-Gram 模型，但在实际训练中一般不会直接使用基础模型。为什么呢？这是因为基础模型的训练过程非常耗时。词汇表一般在百万级别以上，而基础模型的输出层需要进行 **softmax** 计算各个词的输出概率，对于每一次预测，我们都要计算所有 V 个单词出现的概率，这意味着很大的计算量。Skip-Gram 对于一个训练样本就需要做多次多分类任务，因此 Skip-Gram 相对于 CBOW 的计算量更大。有没有简化一点点的方法呢？

有的，研究者们提出了两种高效训练的方法：层序 softmax （hierarchical softmax）和负采样（negative sampling）。二者都是在计算输出概率的时候对计算量进行了优化，分层 Softmax 和负采样是可以相互替代的作为 Word2Vec 的一种加速计算的方式。接下来将详细介绍一下。

2.1 哈夫曼树

层序 softmax 的核心是哈夫曼树，我们先来看一看什么是哈夫曼树，它又是怎么应用的。

哈夫曼树是特殊的二叉树结构，划分为多个节点。现在的关键就是明确每个节点代表什么，每个节点是如何创建的？

哈夫曼树的节点

哈夫曼树有 2 大类节点叶子节点和内节点，

叶子节点：一类是下面无子节点的叶子节点起到输出层神经元的作用，叶子节点的个数即为词汇表的大小。

内部节点：有子节点的内部节点，起到隐藏层神经元的作用。根节点是特殊的内部节点，是整个哈夫曼树的起点。

哈夫曼树编码方式：一般对于一个哈夫曼树的节点（根节点除外），可以约定左子树编码为 0，右子树编码为 1。

哈夫曼树的创建

哈夫曼树是根据词汇表的词频来创建的，所谓词频，就是一个单词在语料库中出现的次数，词频越大的单词在哈夫曼树的层级越浅（越靠近根节点）。

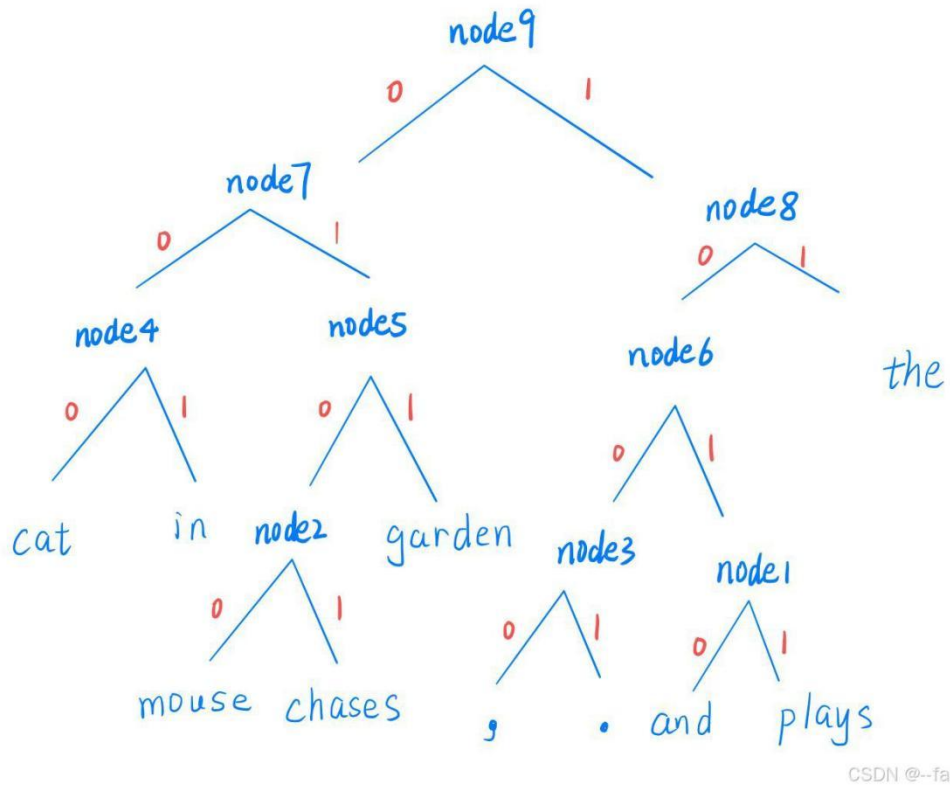
举个简单的例子来看哈夫曼树具体创建的过程，下面是一个语料库中出现的词语的次数和索引，

index	0	1	2	3	4	5	6	7	8	9
x_i	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
word	the	cat	plays	in	garden	,	and	chases	mouse	.
frequency	5	2	1	2	2	1	1	1	1	1

（1）将单词和词频作为一个节点的属性，创建所有词语的节点。

（2）将所有节点放入到优先队列中（词频越小越靠前），每次取出两个最小频次的节点索引，例如 2、6，组成新的节点 node1，该节点的频次（权重）为二者之和（权重 2）。同理我们用 7、8 和 5、9 组合成 node2（权重 2）和 node3（权重 2）节点；

（3）此时队列中最小的频次（权重）为 2，继续取出节点进行合并，过程依次为将 1、3 合并为 node4（权重 4），node2、4 合并为 node5（权重 4），node1、node3 合并为 node6（权重 4），node4、node5 合并成 node7（权重 8），然后将 node6 和 1 合并成 node8（权重 9），最后将 node7 和 node8 合并为 node9（权重 17）得到 Huffman 树，如下图所示。

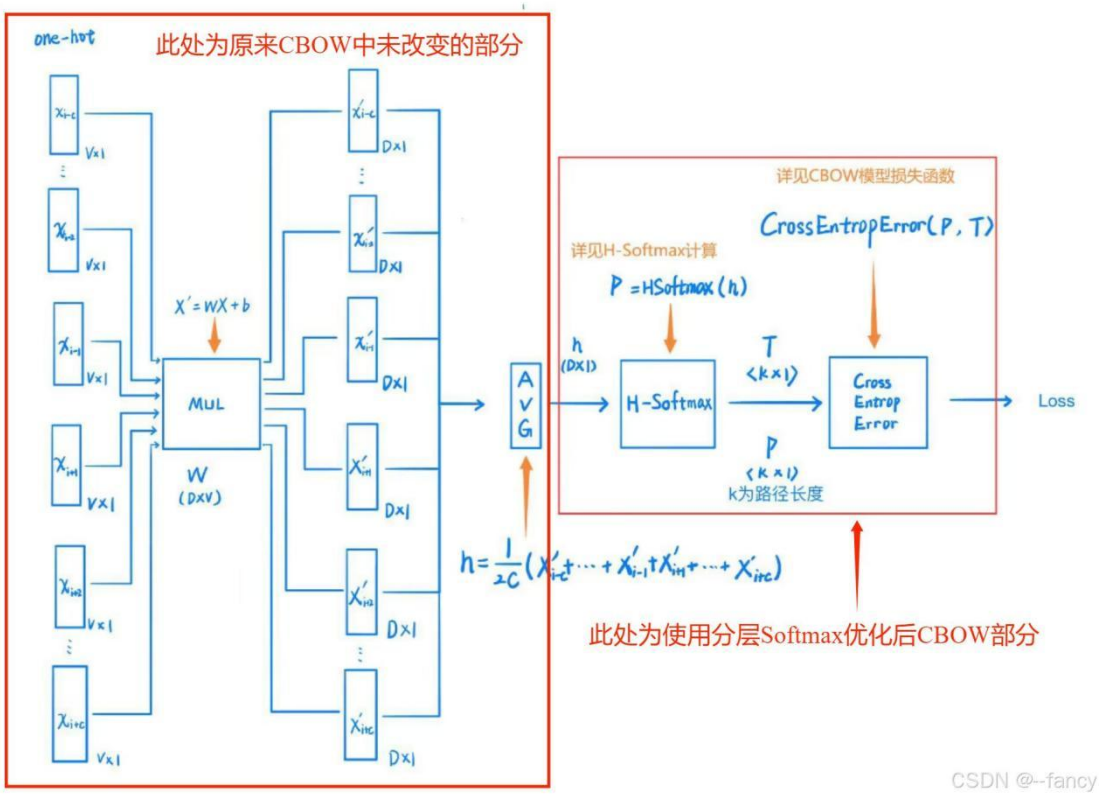


对应于每一个非叶子节点，向左子树方向的编码为 0，向右子树方向的编码为 1。根据路径我们可以得到从根节点到叶子节点（也就是每个单词）的路径，如下表所示。

index	0	1	2	3	4	5	6	7	8	9
word	the	cat	plays	in	garden	,	and	chases	mouse	.
path	11	000	1011	001	011	1000	1010	0101	0100	1001

2.2. 基于分层 softmax 的 CBOW 模型

创建完成哈夫曼树后，介绍一下基于哈夫曼树的 CBOW 模型是如何工作的。



如上图，模型的前半部分是没有变化的，变化的是后面加了哈夫曼树的部分，我们逐步看一下：

（1）首先关注一下输入输出，

输入：某一个中心词的**上下文相关对应的**单词（根据设置的窗口大小决定单词数 C=2c）的 one-hot 编码（C*V，其中 C 是两边的上下文的词语总数，V 是词汇量的长度）；

输出：中心词的哈夫曼树编码路径（1*k,k 为路径长度，不确定值）。

（2）模型的数据流和基础模型相同的处理过程，

第一步是将上下文独热编码 X=C*V 输入模型，然后与隐藏层的参数矩阵 W=V*D 相乘得到隐藏层 X'=XW=(C*V)(V*D)=C*D；

第二步是将隐藏层的矩阵 X'在行维度上求平均数得到向量 h=1*D；

前两步和基础模型一致，关键看一下后面的步骤。

（3）基于哈夫曼树的数据流处理过程。

前面 2 步获得了中间向量 h=1*D，也建立了基于词频的哈夫曼树，那么怎么输出最终的预测结果，也就是词语的哈夫曼树路径呢？

我们需要先给予先前创建的哈夫曼树的每个非叶子节点一个参数向量 θ=D*1，第三步就是利用第二步获得的中间向量 h=1*D 与参数θ相乘，再加上偏移量 b'，然后取 Sigmoid 得到在此节点往正向走的概率：

$$P_i^1 = sigmoid(h\theta + b')$$

相应的，往反向走的概率$P_i^0 = 1 - P_i^1$。这样遍历每个非叶子节点，得到了每个非叶子节点往左还是往右的概率值；

第 4 步就是，根据从根节点到叶子节点（词语）的路径，通过每个非叶子节点往左还是往右的概率值求得根节点到每个叶子节点的概率：

$$P_{word(i)} = \sum_{i=1}^{l} P_i^d$$

其中，d 取 0 或 1，往左还是往右走，l 是词语从根节点到叶子节点（词语）的路径。

第 5 步就是根据根节点到每个叶子节点的概率值，最大的为网络训练预测的词语。然后根据实际标签路径，代入损失函数进行网络训练优化，使得预测路径向正确路径靠拢。

（3）分层 softmax 如何优化计算量的呢？在原本的 Word2Vec 模型的 Softmax 层中，对于每一次预测，我们都要计算所有 V 个单词出现的概率，在分层 Softmax 中，由于使用了 Huffman 树，我们最多计算 V 个单词的概率，平均计算为 logV 次，在数量级巨大时，优化计算十分明显。例如当 V=1000000 时，在 Softmax 层中，我们将计算 1000000 次 e^x 运算，而 log(1000000)≈14 次 (Sigmoid 运算)， 这个优化十分巨大

（4）词向量到底在哪？了解了整个训练过程，似乎没看到词向量在哪里。事实上，词向量就是训练时第一个参数矩阵 W=V*D，而且是经过训练将词汇表中的所有 V 个词的词向量一起训练得到的。

2. 3 Skip-gram

skip-gram 模型的结构与 CBOW 模型正好相反，skip-gram 模型输入某个单词，输出对它上下文词向量的预测。

Skip-gram 的核心同样是一个哈夫曼树, 每一个单词从树根开始到达叶节点，可以预测出它上下文中的一个单词。对每个单词进行 N-1 次迭代，得到对它上下文中所有单词的预测, 根据训练数据调整词向量得到足够精确的结果。

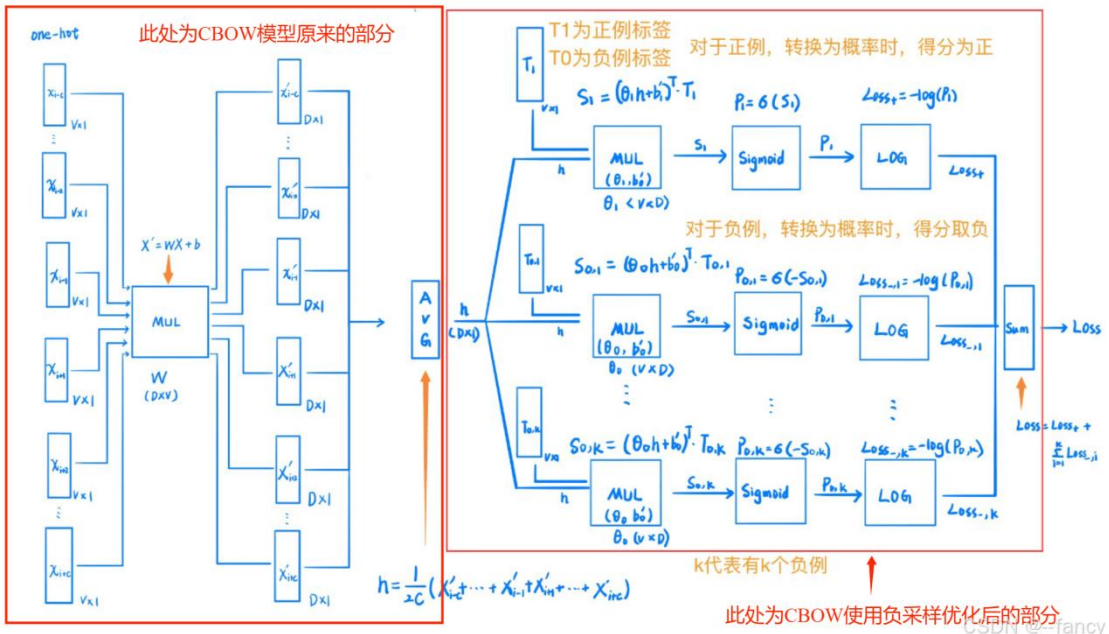
3 负采样

在原本的 Word2Vec 模型中，在 Softmax 层中，我们会进行 V 次 e^x 运算，这个计算量在 V 较大时，计算的时间复杂度特别高。问题的关键就在于我们需要将词汇表所有词语的概率都计算出来，而负采样的基本思想就是不再单独计算词汇表里每个词语的概率，而是从词汇表中选择少数几个负样本来参与每次的训练。

那么什么是负样本呢？在 CBOW 中，根据上下文预测的中心词为正样本，非中心词则都为负样本；在 Skip-gram 中，根据中心词预测的上下文为正样本，非上下文则都为负样本；使用少数几个样本作为负样本，例如我们令负样本数 k = 5 （通常 k 为 5 ~ 20 ），这将把计算时间复杂度将为常数级。

在负采样中，通常不使用 Softmax 多分类，而是使用 Sigmoid 函数进行二分类。我们通常将这 k 个负例分别与正例使用 Sigmoid 函数做二分类计算获得每个样例的得分并组合成得分向量，将多分类转换成 k 个二分类，最后使用 Softmax 归一化得分得到样例的概率值。

比如我们有一个训练样本，中心词是 w，它周围上下文共有 2c 个词，记为 context(w)。由于这个中心词 w 的确和 context(w)相关存在，因此它是一个真实的正例。通过 Negative Sampling 采样，我们得到 neg 个和 w 不同的中心词 wi，i=1,2,..neg。利用这一个正例和 neg 个负例，进行二元逻辑回归，得到负采样对应每个词 wi 的模型参数\theta _{i}，和每个词的词向量。（w 和 wi 都是中心词）



如上为负采样的网络架构，可以看出，前面仍和基础模型一致，只是后面采用了负采样的方法。

（1）首先关注一下输入输出，

输入：某一个中心词的**上下文相关对应的**单词（根据设置的窗口大小决定单词数 C=2c）的 one-hot 编码（C*V，其中 C 是两边的上下文的词语总数，V 是词汇量的长度）；

输出：k 组二分类结果。

（2）学习一下模型的数据流的处理过程，

第一步是将上下文独热编码 X=C*V 输入模型，然后与隐藏层的参数矩阵 W=V*D 相乘得到隐藏层 X’=XW=(C*V)(V*D)=C*D；

第二步是将隐藏层的矩阵 X’在行维度上求平均数得到向量 h=1*D；

（3）负采样层，第三步是将中间向量 h=1*D 再乘上参数矩阵θ’=D*1 然后对其 sigmoid 求是正还是负的概率值，转变为二分类问题。

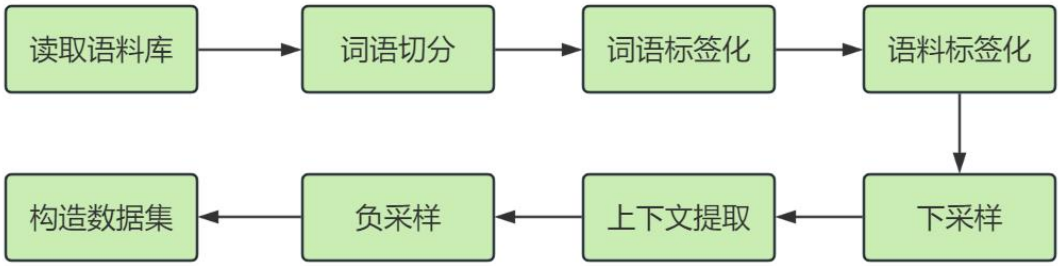
第四步就是将标签值和预测值代入损失函数并利用优化算法优化中间的 2 个参数矩阵，直到满足训练要求。

（3）词向量到底在哪？了解了整个训练过程，似乎没看到词向量在哪里。事实上，词向量就是训练时第一个参数矩阵 W=V*D，而且是经过训练将词汇表中的所有 V 个词的词向量一起训练得到的。

代码

数据处理

流程如下图所示




```
import torch
import torch.nn as nn
import io
import os
import sys
import requests
from collections import OrderedDict
import math
import random
import numpy as np
from tqdm import tqdm
from torch.utils.data import Dataset
from time import sleep
# #下载语料用来训练 word2vec
# def download():
#     #可以从百度云服务器下载一些开源数据集（dataset.bj.bcebos.com）
#     text_url = "https://dataset.bj.bcebos.com/word2vec/text8.txt"
#     #使用 python 的 requests 包下载数据集到本地
#     web_request = requests.get(text_url)
```

```
#     text = web_request.content
#     #把下载后的文件存储在当前目录的 text8.txt 文件内
#     with open("./text8.txt", "wb") as f:
#         f.write(text)
#     f.close()
## #读取整个语料库数据
def load_text(filepath):
    with open(filepath,'r') as f:
        corpus=f.read()#.strip("\n")
    f.close()
    return corpus

#对语料库中的词语进行切分，去除换行符，空格等，并可以将大写转化为小写
def word_preprocess(corpus):
    # strip() 方法在没有参数的情况下，会去除字符串开头和结尾的所有空白字符（该方法只能删除开头或是结尾的字符，不能删除中间部分的字符。）
    # 有参数则只去除指定参数
    # 这包括空格制表符（\t）、换行符（\n）、回车符（\r）、换页符（\f）和垂直制表符（\v）。
    corpus=corpus.strip()#.lower()
    #通过指定分隔符对一个完整的字符串进行切片，如果参数 num 有指定值，将一个完整的字符串分隔 num+1 个子字符串
    corpus=corpus.split(" ")
    return corpus

# #在经过切词后，需要对语料进行统计，为每个词构造 ID。一般来说，可以根据每个词在语料中出现的频次构造 ID，频次越高，ID 越小，便于对词典进行管理。
# # 构造词典，统计每个词的频率，并根据频率将每个词转换为一个整数 id
def word_freq2id(corpus):
# 首先统计每个不同词的频率（出现的次数），使用一个词典记录
    word_freq_dict=dict()
    for word in corpus:
        #如果第一次不在词典里，则创建键值对，初始数字为 0；
        if word not in word_freq_dict:
            word_freq_dict[word]=0
        #若已存在词典里，则数字+1
        word_freq_dict[word]+=1

    #将这个词典中的词，按照出现次数排序，出现次数越高，排序越靠前，赋予的 id 值越小
    # 一般来说，出现频率高的高频词往往是：I，the，you 这种代词，而出现频率低的词，往往是一些名词，如：nlp
    word_freq_dict=sorted(word_freq_dict.items(),key=lambda x:x[1],reverse=True)
```

```
    # 构造 3 个不同的词典，分别存储，
    # 每个词到 id 的映射关系：word2id_dict
    # 每个 id 出现的频率：word2id_freq
    # 每个 id 到词典映射关系：id2word_dict
    word2id_dict=dict()
    word2id_freq=dict()
    id2word_dict=dict()

# 按照频率，从高到低，开始遍历每个单词，并为这个单词构造一个独一无二的 id
    for word,freq in word_freq_dict:
        #根据 word 词典当前存储的词数作为频率，前面已经排序，频率越高，越靠前，id 越小
        curr_id=len(word2id_dict)
        word2id_dict[word]=curr_id
        word2id_freq[word2id_dict[word]]=freq
        id2word_dict[curr_id]=word
    return word2id_dict,word2id_freq,id2word_dict

# #把语料转换为 id 序列
def corpus2id(corpus,word2id_dict):
    # 使用一个循环，将语料中的每个词替换成对应的 id，以便于神经网络进行处理
    corpus=[word2id_dict[word] for word in corpus]
    return corpus

# 使用下采样算法（subsampling）处理语料，强化训练效果
def subsampling(corpus, word2id_freq):
    # 这个 discard 函数决定了一个词会不会被替换，这个函数是具有随机性的，每次调用结果不同
    # 如果一个词的频率很大，那么它被遗弃的概率就很大
    def discard(word_id):
        return random.uniform(0, 1) < 1 - math.sqrt(
            1e-4 / word2id_freq[word_id] * len(corpus))

    corpus = [word for word in corpus if not discard(word)]
    return corpus

def create_context_target(corpus,window_size):
    #python x[:] x[::] x[:::]用法
    # 负数在左侧，则从后往前数 n 个的位置开始
    #负数在右侧，则是排除了后 n 个的位置结束
    #所以这里的 target 是把语料库前后 window_size 个字符排除，确保每个 target 都有 window_size 大小的上下文
```

```
    targets=corpus[window_size:-window_size]
    contexts=[]
    total=len(corpus)-window_size-window_size
    #  tqdm 是 Python 中专门用于进度条美化的模块，通过在非 while 的循环体内嵌入 tqdm，可以得到一个能更好展现程序运行过程的提示进度条
    #这里遍历每一个 target，以其为中心，左右 window_size 范围寻找上下文
    for idx in tqdm(range(window_size,len(corpus)-window_size),total=total,leave=False):
        context=[]
        for t in range(-window_size,window_size+1):
            if t==0:#此时就是 target，略过
                continue
            context.append(corpus[idx+t])
        #每一个 target 对应一个 context 列
        contexts.append(context)
    return contexts,targets

class NegativeSampler:
    def __init__(self,word2id_dict,word2id_freq,id2word_dict,neg_num,power=0.75):
        self.word2id_dict=word2id_dict
        self.word2id_freq=word2id_freq
        self.id2word_dict=id2word_dict
        self.neg_num=neg_num
        #计算存储所有单词的频率之和
        total_freq=0
        for word_id,freq in word2id_freq.items():
            #把每个中心词的频率进行幂计算
            new_freq=math.pow(freq,power)
            word2id_freq[word_id]=new_freq
            total_freq+=new_freq
        #存储词语总数
        self.vocab_size=len(word2id_freq)
        #存储计算每个词汇被选取成负样本的概率值，以词汇在语料库中出现的频率比值计算得到
        self.neg_word_prob=np.zeros(self.vocab_size)
        for word_id,freq in word2id_freq.items():
            self.neg_word_prob[word_id]=freq/total_freq

    def negative_sample(self,target):
        #中心词的数量
        target_size=len(target)
        #计算得到的负样本是一个 target_size*neg_num 的张量
        negative_sample=np.zeros((target_size,self.neg_num),dtype=np.int32)
        for i in range(target_size):
```

```
        #浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。
        p=self.neg_word_prob.copy()
        target_idx=target[i]
        #中心词不会被选作负样本，概率值为 0
        print(target_idx)
        p[target_idx]=0
        p/=p.sum()
        #从大小为 3 的 np.arange(5)生生成一个非均匀的随机样本，没有替换（重复）：
        #>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
        #array([2, 3, 0])

negative_sample[i,:]=np.random.choice(self.vocab_size,size=self.neg_num,replace=False,p=p)
    return negative_sample

#自定义数据集，继承 Dataset 类并重写类的三个函数
class CBOWDataset(Dataset):
  a = 0
  def __init__(self,contexts,targets,negative_sampler):
    self.contexts=contexts
    self.targets=targets
    self.negative_sampler=negative_sampler
  def __len__(self):
    return len(self.contexts)
  def __getitem__(self,idx):

    #对于一组样本，给定 idx,输出对应的 context（2*window_size 大小）在 cbow 模型里作为输入，target（包含一个正样本中心词和 neg 个随机抽取的负样本）
    # 及标签 labels（和 target 一一对应，正样本中心词为 1，负样本为 0）
    contexts=self.contexts[idx]
    targets=[self.targets[idx]]
    #这里会返回一个中心词所对应的 neg 个负样本
    negative_samples=self.negative_sampler.negative_sample(targets)
    #targets 实际上等效于实际的输出值，包含一个正样本中心词和 neg 个随机抽取的负样本
    targets+=[x for x in negative_samples[0]]
    labels=[1]+[0 for _ in range(len((negative_samples[0])))]

    item={
      "contexts":contexts,
      "targets":targets,
      "labels":labels
```

```
        }

        if self.a==0:
            print("item:")
            print(item)
            self.a=1
        return item
    def generate_batch(self, item_list):
        contexts = [x["contexts"] for x in item_list]
        targets = [x["targets"] for x in item_list]
        labels = [x["labels"] for x in item_list]

        outputs = {
            "contexts": torch.LongTensor(contexts),
            "targets": torch.LongTensor(targets),
            "labels": torch.LongTensor(labels),
        }

        return outputs

class SkipGramDataset(Dataset):
    def __init__(self, contexts, centers, negative_sampler):
        self.contexts = contexts
        self.centers = centers
        self.negative_sampler = negative_sampler

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

    def __getitem__(self, idx):
        #skip 和 cbow 的核心区别就是，中心词作为输入，上下文词和负采样值作为输出
        center = self.centers[idx]
        context = self.contexts[idx]
        negative_samples = self.negative_sampler.get_negative_sample(context)
        #z.reshape(-1)变成只有一行的数组
        #.tolist()将数组或矩阵转化为列表
        negative_samples = negative_samples.reshape(-1).tolist()
        label = [1] * len(context) + [0] * len(negative_samples)
        context_negative_samples = context + negative_samples

        item = {
            "center": center,
            "context": context_negative_samples,
```

```
            "label": label,
        }
        return item

    def generate_batch(self, item_list):
        center_ids = [x["center"] for x in item_list]
        context_ids = [x["context"] for x in item_list]
        labels = [x["label"] for x in item_list]

        outputs = {
            "center_ids": torch.LongTensor(center_ids),
            "context_ids": torch.LongTensor(context_ids),
            "labels": torch.LongTensor(labels),
        }

        return outputs
def test():
    import os
    import sys
    from torch.utils.data import DataLoader
    os.chdir(sys.path[0])

    filepath = "./text8.txt"
    window_size = 5
    neg_num = 3

    #读取语料库
    corpus=load_text(filepath)
    print("s1")
    print(corpus[:10])
    #语料库预处理
    corpus=word_preprocess(corpus)
    print("s2")
    print(corpus[:10])
    #词汇标签化
    word2id_dict, word2id_freq, id2word_dict=word_freq2id(corpus)
    corpus=corpus2id(corpus,word2id_dict)
    print("s3")
    print(corpus[:10])
        #下采样
   # corpus=subsampling(corpus,word2id_freq)

    #中心词及上下文配对选择
    contexts,targets=create_context_target(corpus,window_size)
```

```
    print("s6")
    print(contexts[:10])
    print("s7")
    print(targets[:10])
    #负采样
    negative_sampler=NegativeSampler(word2id_dict, word2id_freq,
id2word_dict,neg_num=neg_num)
    print("s8")
  #  print(negative_sampler[:10])
    cbow_dataset=CBOWDataset(contexts,targets,negative_sampler)

    cbow_dataloader = DataLoader(
        dataset=cbow_dataset,
        batch_size=10,
        shuffle=False,
        collate_fn=cbow_dataset.generate_batch,
    )

    for batch in tqdm(cbow_dataloader, total=len(cbow_dataloader)):
        pass

    # sg_dataset = SkipGramDataset(contexts, targets, negative_sampler)
    # sg_dataloader = DataLoader(
    #     dataset=sg_dataset,
    #     batch_size=10,
    #     shuffle=False,
    #     collate_fn=sg_dataset.generate_batch,
    # )
    #
    # for batch in tqdm(sg_dataloader, total=len(sg_dataloader)):
    #     pass


if __name__ == "__main__":
    test()
```

训练

trainer.py 训练过程可视化

```
import os
import torch
```

```
from tqdm import tqdm
from tensorboardX import SummaryWriter

class Trainer():
    def __init__(self,
                 model,
                 optimizer,
                 train_dataloader,
                 outputs_dir,
                 num_epochs,
                 device,
                 ):
        self.model = model
        self.optimizer = optimizer
        self.train_dataloader = train_dataloader
        self.outputs_dir = outputs_dir
        self.num_epochs = num_epochs
        self.device = device
        self.writer = SummaryWriter(outputs_dir)

    def train(self):
        model = self.model
        optimizer = self.optimizer
        train_dataloader = self.train_dataloader
        total_loss = 0

        for epoch in tqdm(range(self.num_epochs), total=self.num_epochs):
            epoch_loss = 0
            for idx, batch in tqdm(enumerate(train_dataloader), total=len(train_dataloader),
leave=False,
                                   desc=f"Epoch {epoch + 1}"):
                inputs = {k: v.to(self.device) for k, v in batch.items()}

                loss = model(inputs)

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

                epoch_loss += loss.item()
                total_loss += loss.item()

                global_step = epoch * len(train_dataloader) + idx + 1
                avg_loss = total_loss / global_step
```

```
            self.writer.add_scalar("Train-Step-Loss", avg_loss, global_step=global_step)

        epoch_loss /= len(train_dataloader)
        self.writer.add_scalar("Train-Epoch-Loss", epoch_loss, global_step=epoch)
        for name, params in model.named_parameters():
            self.writer.add_histogram(name, params, global_step=epoch)

        save_name = f"model_{epoch}.pth"
        save_path = os.path.join(self.outputs_dir, save_name)
        torch.save(model.state_dict(), save_path)
```

train.py 实际训练代码

```
import os
import sys
import time
import torch
import pickle
from torch.utils.data import DataLoader

from tools import word2vec_trainer
from models.nlp import word2vec
from tools import word2vec_build_data
from tools.word2vec_build_data import *

def train_cbow():
    #../..")))  # 返回上上个目录
    filepath = "../../tools/text8.txt"
    #超参数的设置，包括
    window_size = 5    #上下文窗口
    embed_dim = 100    #词向量维度
    batch_size = 100   #批大小
    num_epochs = 10    #训练 epoch
    neg_num = 5        #负样本数
    learning_rate = 1e-3 #学习率
    #记录开始训练时间，start
    now_time = time.strftime("%Y%m%d-%H%M%S", time.localtime())
    outputs_dir = f"../outputs/cbow-{now_time}"
    os.makedirs(outputs_dir, exist_ok=True)
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    # 读取语料库
    corpus= load_text(filepath)
    # 语料库预处理
    corpus=word_preprocess(corpus)
```

```
    # 词汇标签化
    word2id_dict,word2freq_dict,id2word_dict=word_freq2id(corpus)
    corpus=corpus2id(corpus,word2id_dict)
    # 下采样
    corpus=subsampling(corpus,word2freq_dict)
    # 中心词及上下文配对选择
    contexts, targets = create_context_target(corpus, window_size)
    #计算语料库词汇总数
    vocab_size = len(word2id_dict)
    corpus_info = {
        "word2id": word2id_dict,
        "id2word": id2word_dict,
    }
    save_path = os.path.join(outputs_dir, "corpus_info.pkl")
    with open(save_path, "wb") as f:
        pickle.dump(corpus_info, f)
    #负采样
    negative_sampler = NegativeSampler(word2id_dict,word2freq_dict,id2word_dict,
neg_num)
    #利用重写的数据类加载数据集
    train_dataset = CBOWDataset(
        contexts=contexts,
        targets=targets,
        negative_sampler=negative_sampler,
    )

    train_dataloader = DataLoader(
        dataset=train_dataset,
        batch_size=batch_size,
        shuffle=True,
        collate_fn=train_dataset.generate_batch,
        num_workers=0,
        pin_memory=True,
    )

    model = word2vec.CBOW(vocab_size, embed_dim)
    model = model.to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    trainer = word2vec_trainer.Trainer(
        model=model,
        optimizer=optimizer,
        train_dataloader=train_dataloader,
```

```
        outputs_dir=outputs_dir,
        num_epochs=num_epochs,
        device=device,
    )

    trainer.train()


def train_skipgram():
    filepath = "../../tools/text8.txt"
    window_size = 5
    embed_dim = 100
    batch_size = 100
    num_epochs = 10
    negative_sample_size = 5
    learning_rate = 1e-3
    now_time = time.strftime("%Y%m%d-%H%M%S", time.localtime())
    outputs_dir = f"../outputs/skipgram-{now_time}/"
    os.makedirs(outputs_dir, exist_ok=True)
    device = torch.device("cuda")

    corpus, word2id, id2word = load_text(filepath)
    contexts, targets = create_context_target(corpus, window_size)
    vocab_size = len(word2id)

    corpus_info = {
        "corpus": corpus,
        "word2id": word2id,
        "id2word": id2word,
        "contexts": contexts,
        "targets": targets,
    }

    with open("../../tools/text8.txt", "wb") as f:
        pickle.dump(corpus_info, f)

    negative_sampler = NegativeSampler(corpus, negative_sample_size)

    train_dataset = SkipGramDataset(
        contexts=contexts,
        centers=targets,
        negative_sampler=negative_sampler,
    )
```

```
    train_dataloader = DataLoader(
        dataset=train_dataset,
        batch_size=batch_size,
        shuffle=True,
        collate_fn=train_dataset.generate_batch,
        num_workers=0,
        pin_memory=True,
    )
    model = word2vec.SkipGram(vocab_size, embed_dim)
    model = model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # trainer = Trainer(
    #     model=model,
    #     optimizer=optimizer,
    #     train_dataloader=train_dataloader,
    #     outputs_dir=outputs_dir,
    #     num_epochs=num_epochs,
    #     device=device,
    # )
    #
    # trainer.train()

if __name__ == "__main__":
    os.chdir(sys.path[0])
    train_cbow()
   # train_skipgram()
```


参考资料

[1] https://zh.d2l.ai/index.html
https://blog.csdn.net/weixin_44555174/article/details/142751299