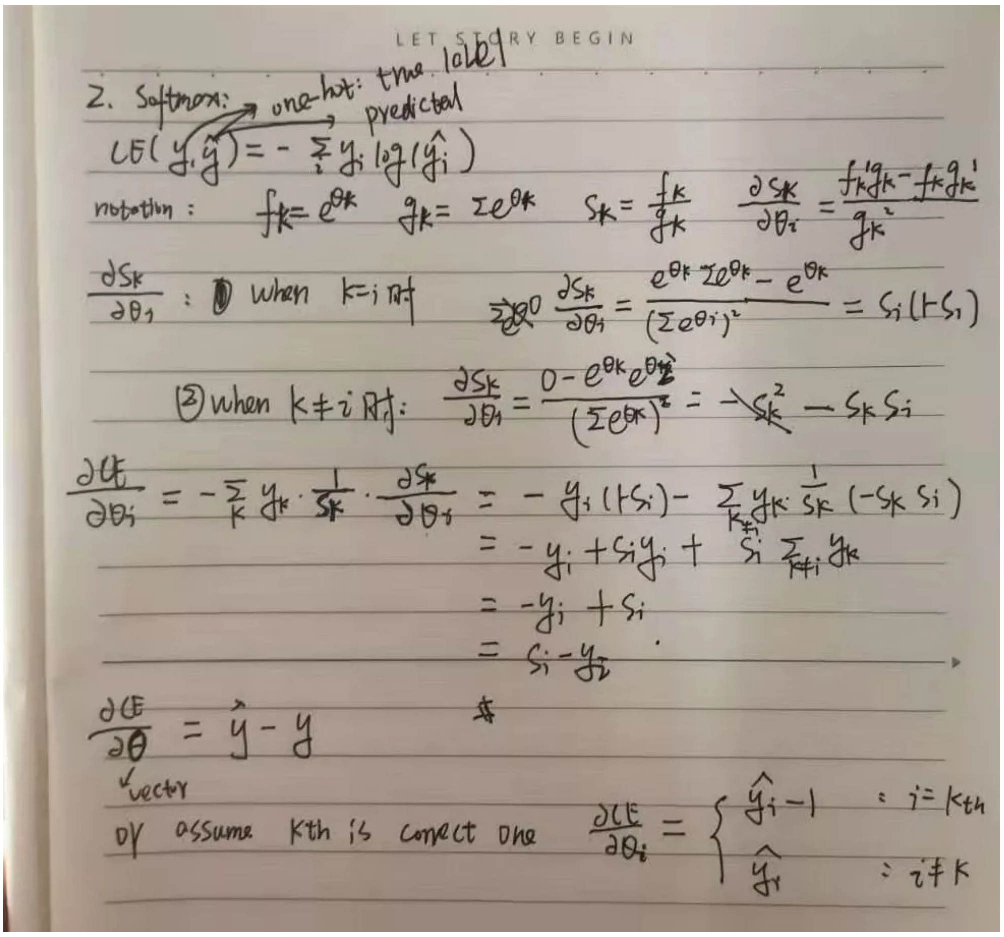
自然语言处理（4）——手推word2vec（下）

dreamer build ai dream 2019-08-13

上次写到了softmax的实现，然后看一下它的梯度求解：



LET STORY BEGIN

2. Softmax: one-hot: true label
predicted

$CE(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$

notation : $f_k = e^{\theta_k}$ $g_k = \sum e^{\theta_k}$ $S_k = \frac{f_k}{g_k}$ $\frac{\partial S_k}{\partial \theta_i} = \frac{f_k' g_k - f_k g_k'}{g_k^2}$

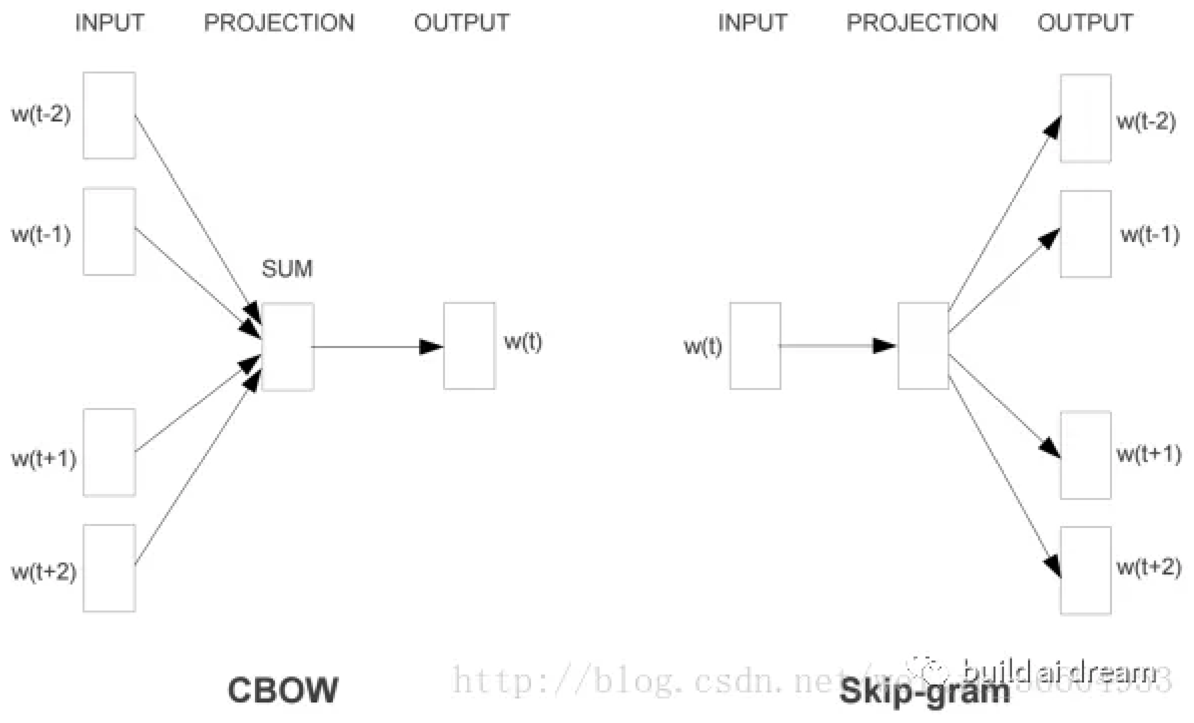
$\frac{\partial S_k}{\partial \theta_i}$: ① when $k=i$ 时 $\frac{\partial S_k}{\partial \theta_i} = \frac{e^{\theta_k} \sum e^{\theta_k} - e^{\theta_k}}{(\sum e^{\theta_i})^2} = S_i(1-S_i)$

② when $k \neq i$ 时: $\frac{\partial S_k}{\partial \theta_i} = \frac{0 - e^{\theta_k} e^{\theta_i}}{(\sum e^{\theta_k})^2} = -\cancel{S_k^2} - S_k S_i$

$$\frac{\partial CE}{\partial \theta_i} = -\sum_k y_k \cdot \frac{1}{S_k} \cdot \frac{\partial S_k}{\partial \theta_i} = -y_i(1-S_i) - \sum_{k\neq i} y_k \frac{1}{S_k}(-S_k S_i)$$
$$= -y_i + S_i y_i + S_i \sum_{k \neq i} y_k$$
$$= -y_i + S_i$$
$$= S_i - y_i$$

$\frac{\partial CE}{\partial \theta} = \hat{y} - y$
vector

or assume kth is correct one $\frac{\partial CE}{\partial \theta_i} = \begin{cases} \hat{y}_i - 1 & : i = k_{th} \\ \hat{y}_i & : i \neq k \end{cases}$

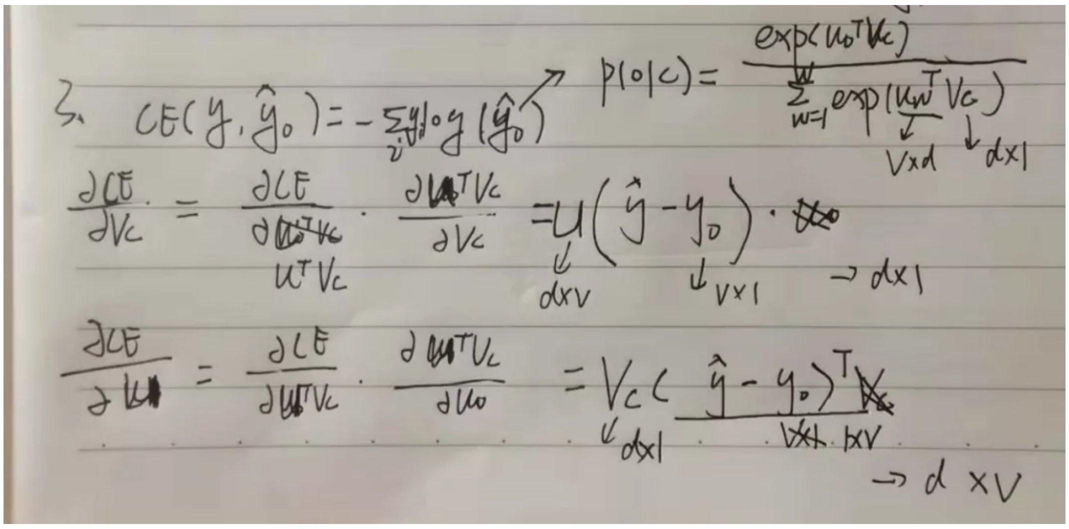


然后根据上图再继续推导关于词向量的梯度.在以softmax为假设函数的word2vec中：

Assume you are given a predicted word vector v c corresponding to the center word c for
skipgram, and word prediction is made with the softmax function found in word2vec models

$$\hat{\boldsymbol{y}}_o = p(\boldsymbol{o} \mid \boldsymbol{c}) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{w=1}^{W} \exp(\boldsymbol{u}_w^\top \boldsymbol{v}_c)}$$

关于中心词与output 词向量的梯度推导为：



3. $CE(y,\hat{y}_o) = -\sum_i y_i \log(\hat{y}_o)$ $\nearrow$ $p(o|c) = \frac{\exp(u_o^T V_c)}{\sum_{w=1}^{W}\exp(\underset{V\times d}{u_w^T} \underset{d\times 1}{V_c})}$

$$\frac{\partial CE}{\partial V_c} = \frac{\partial CE}{\partial u^T V_c}\cdot\frac{\partial u^T V_c}{\partial V_c} = \underset{d\times V}{u}\,\underset{V\times 1}{(\hat{y}-y_o)}\cdot\cancel{x} \rightarrow d\times 1$$

$$\frac{\partial CE}{\partial u} = \frac{\partial CE}{\partial u^T V_c}\cdot\frac{\partial u^T V_c}{\partial u_o} = \underset{d\times 1}{V_c}\,\underset{\cancel{V\times 1}\ 1\times V}{\underline{(\hat{y}-y_o)^T}}\cancel{X} \rightarrow d\times V$$

注意保证矩阵的维数正确即可 :这部分代码如下。predicted 表示中心词，taget表示需要预测的那个词的编号，outputvec表示输出的那个d*V大小的矩阵 。

```
def softmaxCostAndGradient(predicted, target, outputVectors, dataset):
     """
     Softmax cost function for word2vec models
    Implement the cost and gradients for one predicted word vector
    and one target word vector as a building block for word2vec
    models, assuming the softmax prediction function and cross
    entropy loss.
    Arguments:
```

```
    predicted -- numpy ndarray, predicted word vector (\hat{v} in
                 the written component)
    target -- integer, the index of the target word
    outputVectors -- "output" vectors (as rows) for all tokens
    dataset -- needed for negative sampling, unused here.

    Return:
    cost -- cross entropy cost for the softmax word prediction
    gradPred -- the gradient with respect to the predicted word
           vector
    grad -- the gradient with respect to all the other word
           vectors

    We will not provide starter code for this function, but feel
    free to reference the code you previously wrote for this
    assignment!
    """

    Vc=predicted.reshape(-1,1) # d*1
    z=np.dot( outputVectors,Vc)  #V*d . d*1=>v*1
    yhat=softmax(z).reshape(-1,1) # make sure the shape is  a column   v*1
    cost=-np.log(yhat[target])
    yhat[target]-=1
    gradPred=np.dot(outputVectors.T, yhat) # d*v .v*1=> d*1
    grad=np.dot(yhat,Vc.T)   # v*d
### END YOUR CODE
    return cost, gradPred, grad
```

```

def skipgram(currentWord, C, contextWords, tokens, inputVectors, outputVector
dataset, word2vecCostAndGradient=softmaxCostAndGradient):

     """ Skip-gram model in word2vec
    Implement the skip-gram model in this function
Arguments:
```

```
currentWord -- a string of the current center word
C -- integer, context size
contextWords -- list of no more than 2*C strings, the context words
tokens -- a dictionary that maps words to their indices in
the word vector list
inputVectors -- "input" word vectors (as rows) for all tokens
outputVectors -- "output" word vectors (as rows) for all tokens
word2vecCostAndGradient -- the cost and gradient function for
a prediction vector given the target
word vectors, could be one of the two
                               cost functions you implemented above.
Return:
cost -- the cost function value for the skip-gram model
grad -- the gradient with respect to the word vectors
    """
    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)  # w
    gradOut = np.zeros(outputVectors.shape) #w'
    center= tokens[currentWord]
    predicted=inputVectors[center]
    for word in contextWords:
        target=tokens[word]
        cost_sub, gradPred, grad=word2vecCostAndGradient(predicted, target, 
        cost+=cost_sub
        gradIn[ center] +=np.squeeze(gradPred )
        gradOut+=grad
    return cost, gradIn, gradOut
 print( skipgram("c", 3, ["a", "b", "e", "d", "b", "c"],
        dummy_tokens, dummy_vectors[:5,:], dummy_vectors[5:,:], dataset)  )

```

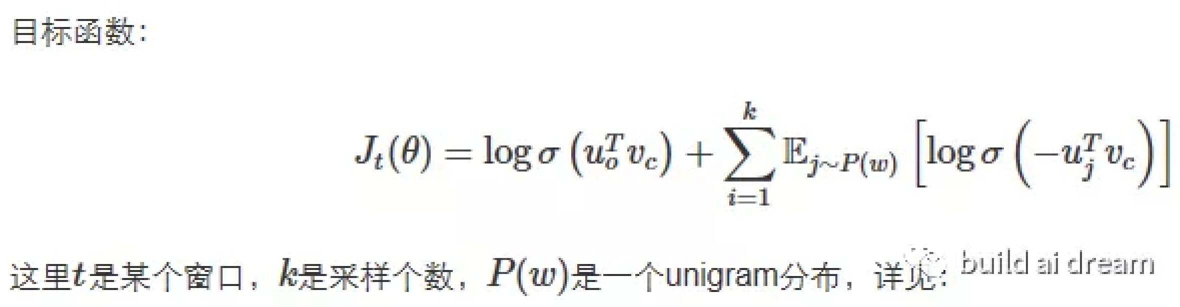
可以看到对于每一个中心，会将其需要预测的背景词遍历一遍，将cost和梯度都进行加一遍。

```
def test_word2vec():
""" Interface to the dataset for negative sampling """
    dataset = type('dummy', (), {})()
def dummySampleTokenIdx():
return random.randint(0, 4)
```

```

def getRandomContext(C):
        tokens = ["a", "b", "c", "d", "e"]
return tokens[random.randint(0,4)], \
            [tokens[random.randint(0,4)] for i in range(2*C)]
    dataset.sampleTokenIdx = dummySampleTokenIdx
    dataset.getRandomContext = getRandomContext
    random.seed(31415)
    np.random.seed(9265)
    dummy_vectors = normalizeRows(np.random.randn(10,3)) # d=3 , n=10/2=5
    dummy_tokens = dict([("a",0), ("b",1), ("c",2),("d",3),("e",4)])
print ("==== Gradient check for skip-gram ====")
    gradcheck_naive(lambda vec: word2vec_sgd_wrapper(
        skipgram, dummy_tokens, vec, dataset, 5, softmaxCostAndGradient),
        dummy_vectors)
    gradcheck_naive(lambda vec: word2vec_sgd_wrapper(
        skipgram, dummy_tokens, vec, dataset, 5, negSamplingCostAndGradient),
        dummy_vectors)
print ("\n==== Gradient check for CBOW      ====")
    gradcheck_naive(lambda vec: word2vec_sgd_wrapper(
        cbow, dummy_tokens, vec, dataset, 5, softmaxCostAndGradient),
        dummy_vectors) #梯度检查
    gradcheck_naive(lambda vec: word2vec_sgd_wrapper(
        cbow, dummy_tokens, vec, dataset, 5, negSamplingCostAndGradient),
        dummy_vectors)

    print( "\n=== Results ===")
    print( skipgram("c", 3, ["a", "b", "e", "d", "b", "c"],
        dummy_tokens, dummy_vectors[:5,:], dummy_vectors[5:,:], dataset)  )
print (skipgram("c", 1, ["a", "b"],
        dummy_tokens, dummy_vectors[:5,:], dummy_vectors[5:,:], dataset,
        negSamplingCostAndGradient))
print (cbow("a", 2, ["a", "b", "c", "a"],
        dummy_tokens, dummy_vectors[:5,:], dummy_vectors[5:,:], dataset))
print (cbow("a", 2, ["a", "b", "a", "c"],
        dummy_tokens, dummy_vectors[:5,:], dummy_vectors[5:,:], dataset,
        negSamplingCostAndGradient))
```

```
#为了梯度检查，返回函数值和其根据求导公式推出来的数值
def word2vec_sgd_wrapper(word2vecModel, tokens, wordVectors, dataset, C,
word2vecCostAndGradient=softmaxCostAndGradient):
batchsize = 50
cost = 0.0
grad = np.zeros(wordVectors.shape)
N =int( wordVectors.shape[0])
   # print(N/2)
inputVectors = wordVectors[:int(N/2),:]
outputVectors = wordVectors[int(N/2):,:]
for i in range(batchsize):
C1 = random.randint(1,C)
centerword, context = dataset.getRandomContext(C1)

if word2vecModel == skipgram:
denom = 1
else:
denom = 1

c, gin, gout = word2vecModel(
centerword, C1, context, tokens, inputVectors, outputVectors,
dataset, word2vecCostAndGradient)
cost += c / batchsize / denom
grad[:int(N/2), :] += gin / batchsize / denom
grad[int(N/2):, :] += gout / batchsize / denom

return cost, grad
```

之前讲的都是正常的采样方法，这样的方法需要遍历整个语料库中的每个词然后预测每个词的上下文。
可以根据其求导公式看，梯度下降时运算量很大，要用整个d*W矩阵，当W很大时，运算量就很大。所以提出负采样方法：

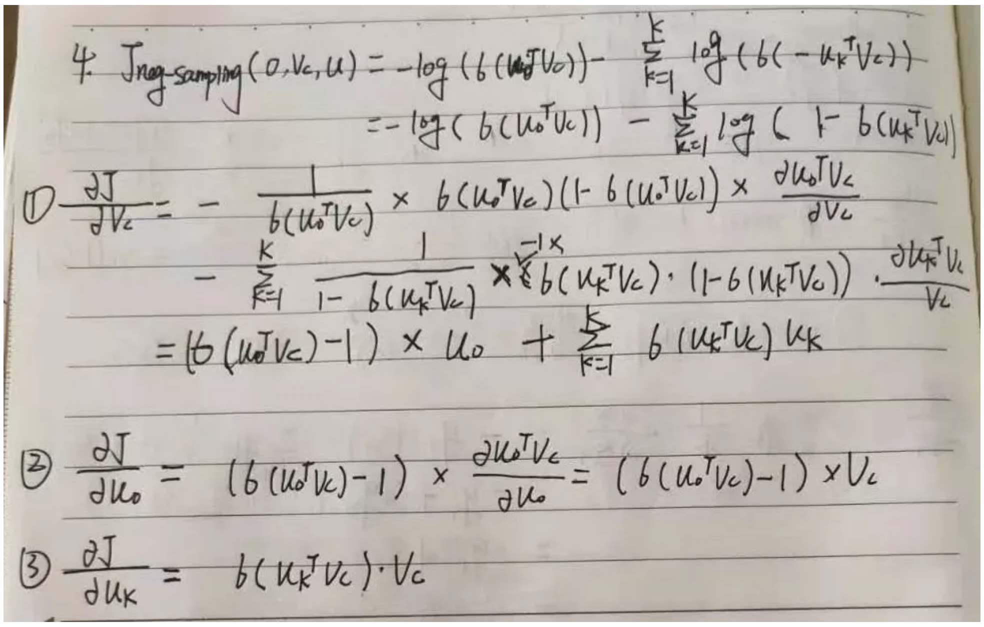


目标函数：

$$J_t(\theta) = \log \sigma\left(u_o^T v_c\right) + \sum_{i=1}^{k} \mathbb{E}_{j \sim P(w)} \left[\log \sigma\left(-u_j^T v_c\right)\right]$$

这里t是某个窗口，k是采样个数，$P(w)$是一个unigram分布，详见：

就是故意取K个不是预测目标的词语，negative sampling实现skip-gram。这是一种采样子集简化运算的方法。具体做法是，对每个正例（中央词语及上下文中的一个词语）采样几个负例（中央词语和其他随机词语），训练binary logistic regression（也就是二分类器）。



4. $$J_{neg\text{-}sampling}(o, v_c, U) = -\log(\sigma(u_o^T v_c)) - \sum_{k=1}^{K} \log(\sigma(-u_k^T v_c))$$
$$= -\log(\sigma(u_o^T v_c)) - \sum_{k=1}^{K} \log(1 - \sigma(u_k^T v_c))$$

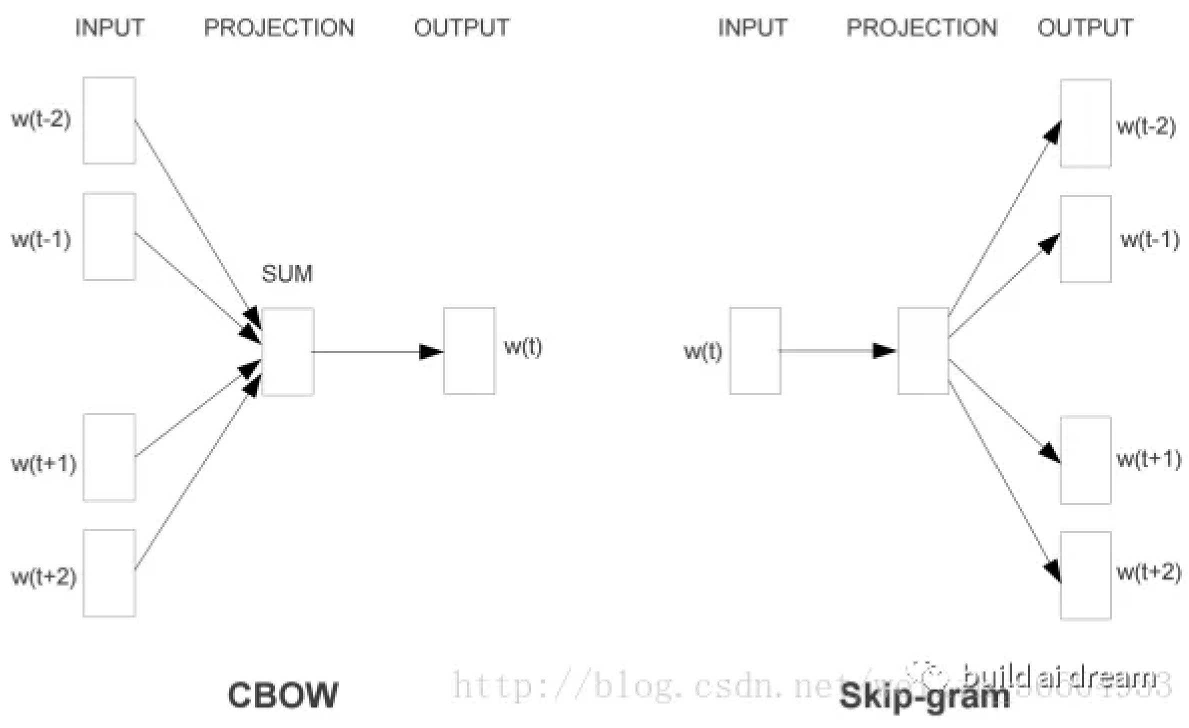
① $$\frac{\partial J}{\partial v_c} = -\frac{1}{\sigma(u_o^T v_c)} \times \sigma(u_o^T v_c)(1 - \sigma(u_o^T v_c)) \times \frac{\partial u_o^T v_c}{\partial v_c}$$
$$- \sum_{k=1}^{K} \frac{1}{1 - \sigma(u_k^T v_c)} \times \overset{-1\times}{} \sigma(u_k^T v_c) \cdot (1 - \sigma(u_k^T v_c)) \cdot \frac{\partial u_k^T v_c}{v_c}$$
$$= (\sigma(u_o^T v_c) - 1) \times u_o + \sum_{k=1}^{K} \sigma(u_k^T v_c) u_k$$

② $$\frac{\partial J}{\partial u_o} = (\sigma(u_o^T v_c) - 1) \times \frac{\partial u_o^T v_c}{\partial u_o} = (\sigma(u_o^T v_c) - 1) \times v_c$$

③ $$\frac{\partial J}{\partial u_k} = \sigma(u_k^T v_c) \cdot v_c$$

```python
def getNegativeSamples(target, dataset, K):  #负采样函数，取得负样本的编号
    """ Samples K indexes which are not the target """

    indices = [None] * K
    for k in range(K):
        newidx = dataset.sampleTokenIdx()
        while newidx == target:
            newidx = dataset.sampleTokenIdx()
        indices[k] = newidx
    return indices


def negSamplingCostAndGradient(predicted, target, outputVectors, dataset,
                               K=10):
    """ Negative sampling cost function for word2vec models

    Implement the cost and gradients for one predicted word vector
    and one target word vector as a building block for word2vec
    models, using the negative sampling technique. K is the sample
    size.

    Note: See test_word2vec below for dataset's initialization.

    Arguments/Return Specifications: same as softmaxCostAndGradient
    """

    # Sampling of indices is done for you. Do not modify this if you
    # wish to match the autograder and receive points!
    indices = [target]
    indices.extend(getNegativeSamples(target, dataset, K))

    ### YOUR CODE HERE
    grad = np.zeros(outputVectors.shape)
    gradPred = np.zeros(predicted.shape)
    cost = 0
    z = sigmoid(np.dot(outputVectors[target], predicted))

    cost -= np.log(z)
    grad[target] += predicted * (z - 1.0)
    gradPred += outputVectors[target] * (z - 1.0)

     for k in  indices[1:] :
        z = sigmoid(np.dot(outputVectors[k], predicted))
        cost -= np.log(1.0 - z)
        grad[k] += predicted * z
        gradPred += outputVectors[samp] * z
    ### END YOUR CODE
    return cost, gradPred, grad
```

上面讲完了skip-gram 以及 两种训练方法的实现。下面讲一下cbow的实现：



其实训练过程和梯度更新都一样，就是predicted 和target 有所不同：

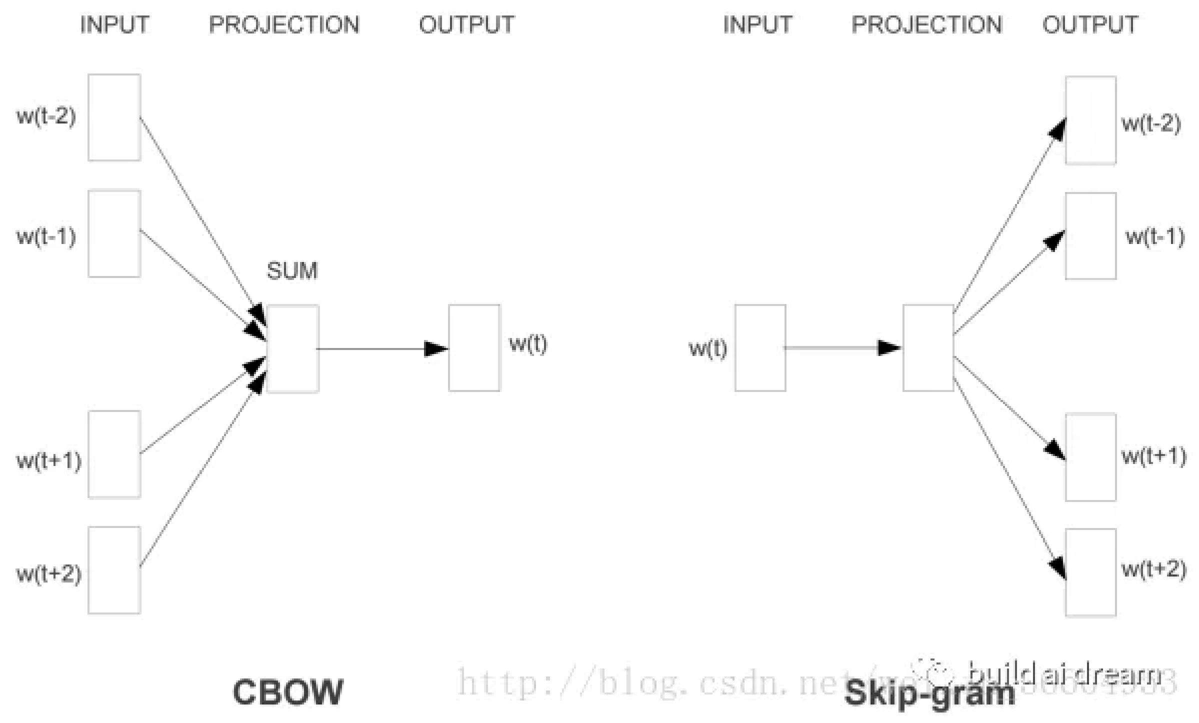
```
def cbow(currentWord, C, contextWords, tokens, inputVectors, outputVectors,
         dataset, word2vecCostAndGradient=softmaxCostAndGradient):
    """CBOW model in word2vec

    Implement the continuous bag-of-words model in this function.

    Arguments/Return specifications: same as the skip-gram model

    Extra credit: Implementing CBOW is optional, but the gradient
    derivations are not. If you decide not to implement CBOW, remove
    the NotImplementedError.
    """


    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)

    context_id=[tokens[context] for context in contextWords ]
    predicted_vec=inputVectors[context_id]
    predicted=np.sum(predicted_vec,axis=0)    # 求和函数
    target= tokens[currentWord]
    cost_sub, gradPred, grad=word2vecCostAndGradient(predicted, target, outpu
    cost+=cost_sub
    gradOut+=grad
    # 注意这儿:
    for i in  context_id:
        gradIn[i]+=np.squeeze(gradPred)
    return cost, gradIn, gradOut

```

这里上下文向量是所有上下文词向量的和，却把梯度应用到了所有的上下文词向量上去。虽然看上去挺疯狂，但就是管用。

至此：W2V的两种方法我们都实现了：一般来说一个词语有两个向量，可以将其拼接或者相加。相加用得更多

下面为常见的面试题目：
q :word2vec有哪两种结构？有什么区别

ans:
word2vec包含两种结构，一种是skip-gram结构，一种是cbow结构，skip-gram结构是利用中间词预测邻近词，cbow模型是利用上下文词预测中间词，网络结构如下图：



区别:CBOW - continous bag-of-words， 是根据相邻单词来计算当前单词的词向量，因而不考虑window 内的单词的顺序；skip gram 则相反，是根据当前单词来计算

其邻近单词的词向量，因而考虑window 内的单词语序（距离当前词比较近的单词被赋予的权重就大）；CBOW 训练速度比 skip gram 快，但skip gram 对于低频词(多个背景词对其训练)的效果比较好.

q：　word2vec中有没有非线性层

ans:

　word2vec两种模型有三层，输入层，映射层，输出层，隐藏层是线性结构，这也是为什么相对于语言模型训练较快的原因。

q: CBOW模型输入是什么，输出是什么，初始化出入是怎么做的?

ans:

　CBOW模型输入是上下文的词向量，这个词向量是随机初始化的产生的，将这个上下文词向量加起来得到映射层，输出层是层次softmax(hierarchical Softmax)，最终输出中心词的最大概率。

q: 层次softmax（hierarchical Softmax）在word2vec中是怎么应用的，为什么要用，用了有哪些好处?

ans:

　面试官问此问题的目的：主要是考察应聘者对word2vec优化的理解及对word2vec更深层次的理解。在回答这个问题的时候应聘者应先阐述普通softmax会产生什么样的问题，然后再阐述使用层次softmax后能够达到什么样的效果。

　　相应答案（仅供参考）

　假设我们采用普通的softmax，那么在输出层需要得到一个词典大小的概率分布，假设公式我们采用softmax（Wx+b）（这里一定要在纸或者白板上写出），那么W将会有一个维度是10000维，这是一个很大的权重矩阵，那么我们整个优化目标是10000个，这个工作量的巨大的，会严重影响计算效率。如果采用层次softmax，那么我们的输出层将是一个霍夫曼二叉树，这个二叉树的最大深度是log(D)-1,每个节点处进行一个二分类，二分类的权重矩阵W2有一个维度是2，相对于词典大小10000来说就是很小的数。采用层次softmax好处是将一个多分类问题转换成一个一个的二分类问题，降低的计算复杂度，提高了运算效率。

q:在word2vec优化中采用层次softmax的哈夫曼树如何构建，叶子节点代表什么?

ans:

面试官问此问题的目的：主要是考察应聘者对word2vec优化底层原理的理解。

相应答案（仅供参考）

在word2vec中构建哈夫曼树中，我们以词表中的全部词作为叶子节点，词频作为节点的权，构建Huffman树，叶子节点为词表中的全部词，权重是词频。

(层次化索引在后续会补充上来)

refer: 微信公众号：AI壹号堂

https://blog.csdn.net/weixin_40547993/article/details/88779922