word2vec code review

词的结构体

```
struct vocab_word {
  long long cn;
  int *point;
  char *word, *code, codelen;
};
```

cn: word的词频
point: 存放该词在Huffuman Tree中的路径
word: 词的内容
code：Huffuman 编码
codelen：编码的长度

和词有关的存储单元

```
struct vocab_word *vocab;
vocab = (struct vocab_word *)calloc(vocab_max_size,
sizeof(struct vocab_word));
vocab_hash = (int *)calloc(vocab_hash_size, sizeof(int));
expTable = (real *)malloc((EXP_TABLE_SIZE + 1) *
sizeof(real));
```

和word vector相关的存储单元

```
// syn0存放词向量
posix_memalign((void **)&syn0, 128, (long long)vocab_size *
layer1_size * sizeof(real));

// syn1存放中间节点的向量
posix_memalign((void **)&syn1, 128, (long long)vocab_size *
layer1_size * sizeof(real));

// syn1neg 存放negative sampling得到的词对应的向量
// 和vocabulary word的向量用不同的形式表示
posix_memalign((void **)&syn1neg, 128, (long long)vocab_size
* layer1_size * sizeof(real));
```

创建Huffuman树

```
// 前面的vocab_size + 1个元素是词的词频，后面的vocab_size个元素存中间节点对应的count
long long *count = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));

// 0或者1，说明是父节点的左孩子还是右孩子
long long *binary = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));

// 存放节点的父节点在vocab数组中对应的下标
long long *parent_node = (long long *)calloc(vocab_size * 2 + 1, sizeof(long long));
```

创建Huffuman树

STEP 1:

pos1 = vocab_size - 1;

pos2 = vocab_size;

// find two smallest nodes min1, min2

loop: a from 0 to vocab_size - 2

count[vocab_size + a] = count[min1i] + count[min2i];

parent_node[min1i] = vocab_size + a;

parent_node[min2i] = vocab_size + a;

binary[min2i] = 1;(initial 0)

STEP 2:

// assign binary code to each vocabulary word

// move from leaf to root and get Huffuman code and path

set vocab[a].code

set vocab[a].point

Objective Function

$$\frac{1}{T}\sum_{t=1}^{T} log P(w_t|w_{avg})$$

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} \log p(w_{t+j}|w_t)$$

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma\left([\![n(w,j+1) = \mathrm{ch}(n(w,j))]\!] \cdot {v'_{n(w,j)}}^{\top} v_{w_I}\right)$$

Train Model（CBOW）

```
// neu1存放窗口中边缘词的向量平均值
real *neu1 = (real *)calloc(layer1_size, sizeof(real));

// 存放边缘词的向量更新（包括hs和negative sampling）
real *neu1e = (real *)calloc(layer1_size, sizeof(real));

neu1[c] += syn0[c + last_word * layer1_size];

// 然后再求平均，cw为边缘次的个数
neu1[c] /= cw;
```

CBOW hierachical softmax

```
for (d = 0; d < vocab[word].codelen; d++) {
    // l2是Huffuman树路径上中间节点对应的向量在syn1中的下标
    l2 = vocab[word].point[d] * layer1_size;
    // 做内积
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
    f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
    // 梯度和学习率之积
    g = (1 - vocab[word].code[d] - f) * alpha;
    // 每走一步词向量的更新值保存起来，保存在neu1e中
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
    // 更新对应的中间节点向量
    for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
}
```

CBOW negative sampling

```
for (d = 0; d < negative + 1; d++) {
    next_random = next_random * (unsigned long long)25214903917 + 11;
    target = table[(next_random >> 16) % table_size];
    l2 = target * layer1_size;
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
    g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP /
2))]) * alpha;
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
    for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
}
// 注意采样的方式，d=0时，采的样本是本词，正样本，label取1，其余的为负样
本，label取0

// 做完hs 和 negative sampling之后更新边缘词的向量
for (a = b; a < window * 2 + 1 - b; a++) {
    if (a != window) {
        for (c = 0; c < layer1_size; c++)
            syn0[c + last_word * layer1_size] += neu1e[c];
    }
}
```

Train model (Skip-gram)

```
// 存放串口中间词的向量更新（包括hs和negative sampling）
real *neu1e = (real *)calloc(layer1_size, sizeof(real));
```

注意neu1e在Skip-gram中和CBOW中代表的含义的不同

Skip-gram hs

```
// 注意比CBOW多了一层循环，因为由中间词预测边缘词，有2*window-1个边缘词
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
    // hs
    for (d = 0; d < vocab[word].codelen; d++) {
        l2 = vocab[word].point[d] * layer1_size;
        for (c = 0; c < layer1_size; c++) f += syn0[c + l1] * syn1[c + l2];
        f = expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))];
        g = (1 - vocab[word].code[d] - f) * alpha;
        for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
        for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * syn0[c + l1];
    }

    // negative sampling 同CBOW

    // 更新中间词的向量
    for (c = 0; c < layer1_size; c++) syn0[c + l1] += neu1e[c];
}
```

容易混淆的地方

1. vocabulary word有向量表示，Huffuman树上的中间节点也有向量表示，它们是存储到不同的数组中的，对应论文里objective function中加’和不加’的向量

2. 向量更新：使用随机梯度上升（窗口滑动一次相当于一个sample），似然函数（条件概率）中有哪些向量就更新哪些向量

随机梯度上升每次更新的内容

CBOW: 窗口滑动到一个位置，用窗口边缘的多个词（向量平均值）预测中间词。在Huffuman树上从根节点走向窗口中间词对应的叶节点，每走一步，更新路径上当前所在的中间节点的向量；还需要更新窗口边缘的多个词的向量（更新多个词向量），这些词的向量是在走完Huffuman树上的路径之后进行的，每走一步，更新值累加保存起来

Skip-gram: 窗口滑动到一个位置，用中间词预测边缘词（有多少个边缘词就循环多少次，对应skip-gram程序中的最外层循环）。在Huffuman树上从根节点走向窗口边缘词对应的叶节点，每走一步，更新路径上当前所在的中间节点的向量；还需要更新窗口中间词的向量（更新一个词向量），也是在树上走完一条路径之后进行更新的。

Thanks!