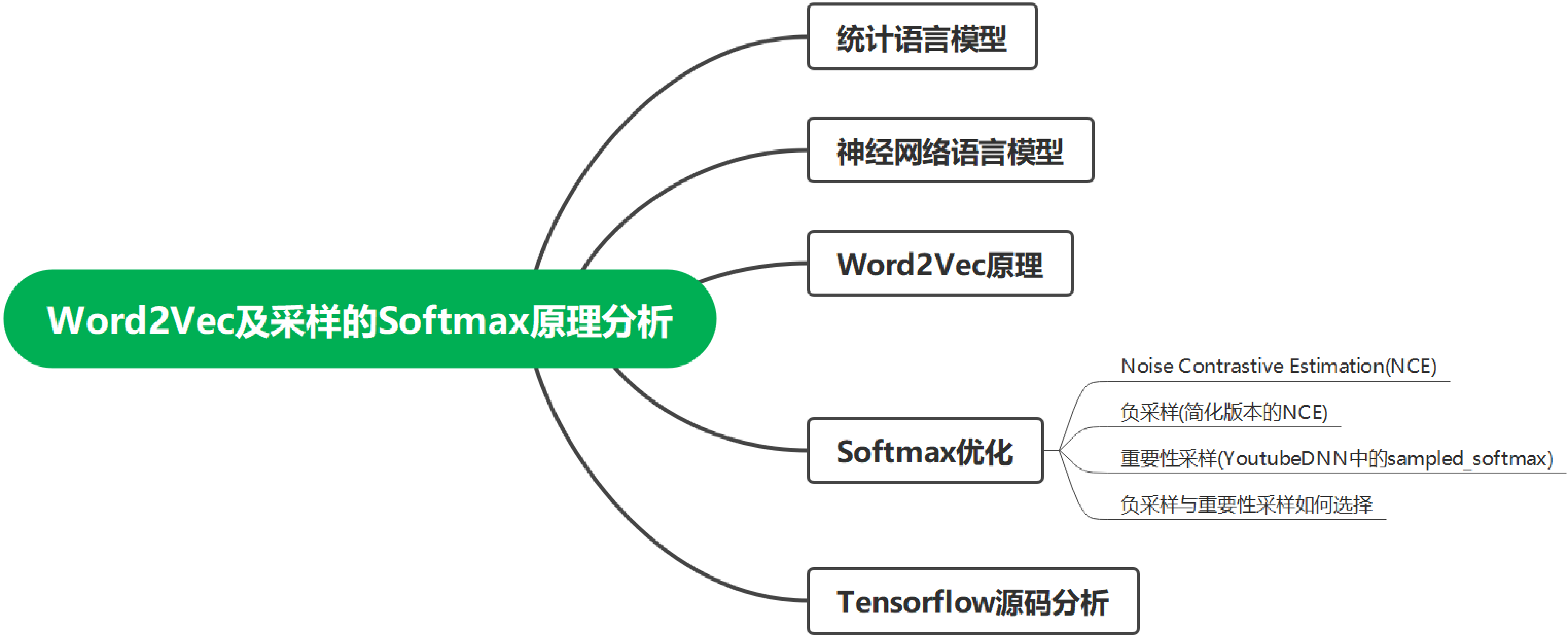
Datawhale

Word2Vec及 采样的Softmax原理分析

FunRec学习小组　罗如意

2022年4月17日


Word2Vec及采样的Softmax原理分析
统计语言模型
神经网络语言模型
Word2Vec原理
Softmax优化
Noise Contrastive Estimation(NCE)
负采样(简化版本的NCE)
重要性采样(YoutubeDNN中的sampled_softmax)
负采样与重要性采样如何选择
Tensorflow源码分析

统计语言模型

语言模型就是用来计算一个句子的概率的模型，给定一个句子($w_1, w_2, \dots, w_T$), 通过链式法则计算联合概率：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i | w_1, w_2, \dots, w_{i-1})$$

$$P(w_1, w_2, \dots, w_T) = \mathrm{P}(w_1)\mathrm{P}(w_2|w_1) \dots \mathrm{P}(w_T|w_1, w_2, \dots, w_{T-1})$$

如何计算条件概率？ 根据大数定律：用频率近似概率

$$P(w_k | w_1, w_2, \dots, w_{k-1}) = \frac{count(w_1, w_2, \dots, w_k)}{count(w_1, w_2, \dots, w_{k-1})}$$

上述条件概率的计算存在哪些问题？

（1）参数空间过大：语言模型的参数就是所有的条件概率

（2）数据太稀疏：大量的单词组合出现的次数为0，导致最红的概率为0

马尔科夫假设：任意一个词出现的概率只与它前面出现的有限的一个或者几个词有关(假如前k个词)，语言模型表示为：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i | w_{i-k}, \dots, w_{i-1})$$

$$P(w_1, w_2, \dots, w_T) = \mathrm{P}(w_1)\mathrm{P}(w_2|w_1) \dots \mathrm{P}(w_T|w_{T-k}, \dots, w_{T-1})$$

N-Gram模型： 本质上是N-1阶的马尔科夫假设，认为一个词出现的概率只与它前面的N-1个词有关

N=1时，被称为是一元模型（Unigram Model），即w_i与它前面的0个词相关，也就是每个词是相互独立的：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i)$$

N=2时，被称为是二元模型（Bigram Model），即w_i与它前面的1个词相关：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i | w_{i-1})$$

注意和后面介绍的Word2vec进行对比

$$P(w_1, w_2, \dots, w_T) = \mathrm{P}(w_1)\mathrm{P}(w_2|w_1) \dots \mathrm{P}(w_T|w_{T-2}, w_{T-1})$$

N=3时，被称为是三元模型（Trigram Model），即w_i与它前面的2个词相关：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i | w_{i-2}, w_{i-1})$$

N-Gram模型存在的问题：

N不能太大，否则参数量太大，同时带来数据稀疏问题

神经网络语言模型

语言模型就是用来计算一个句子的概率的模型，给定一个句子($\boldsymbol{w_1}, \boldsymbol{w_2}, \dots, \boldsymbol{w_T}$), 通过链式法则计算联合概率：

$$P(w_1, w_2, \dots, w_T) = \prod_i P(w_i | w_1, w_2, \dots, w_{i-1})$$

$$P(w_1, w_2, \dots, w_T) = \mathrm{P}(w_1)\mathrm{P}(w_2|w_1) \dots \mathrm{P}(w_T|w_1, w_2, \dots, w_{T-1})$$

对于上述条件概率，能不能用神经网络来计算呢？为了更好的描述下面的公式直接从愿论文中截图的，对于有n个词的句子($z_1, z_2, \dots, z_n$)，联合概率表示为：

$$\hat{P}(Z_1 = z_1, \cdots, Z_n = z_n) = \prod_i \hat{P}(Z_i = z_i | g_i(Z_{i-1} = z_{i-1}, Z_{i-2} = z_{i-2}, \cdots, Z_1 = z_1)),$$

条件概率的计算公式为： $\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$

The y_i are the unnormalized log-probabilities for each output word i, computed as follows, with parameters b,W,U,d and H:

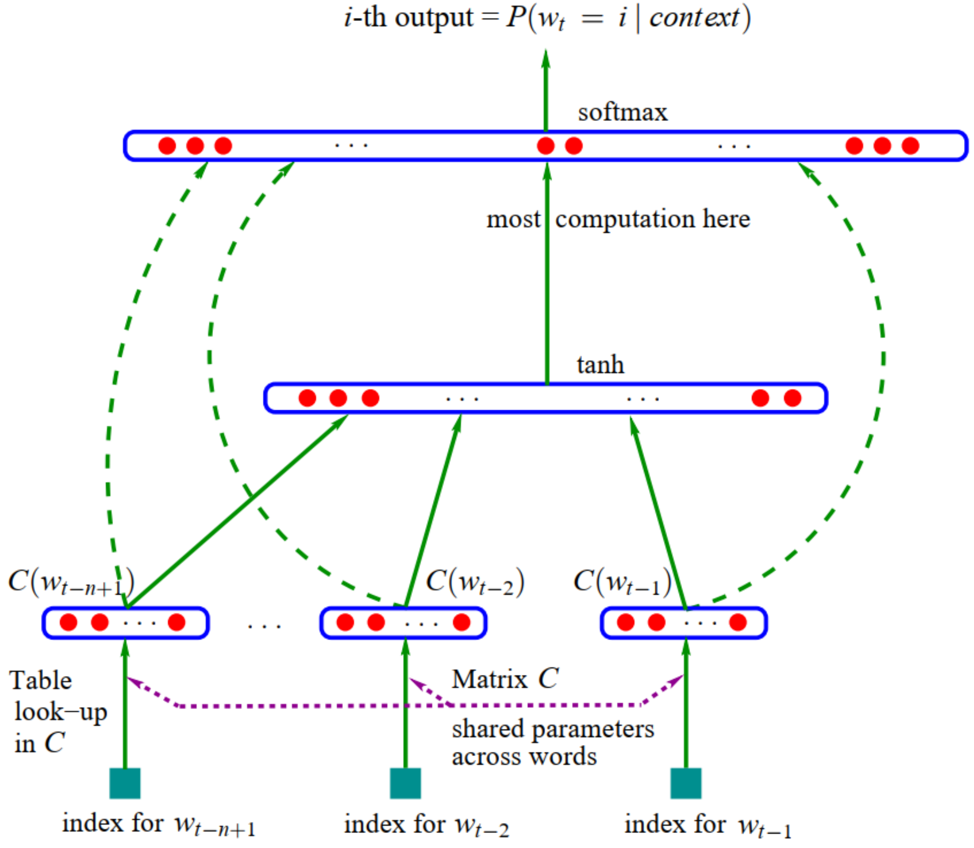
y和b都是词典维度的向量 $\boxed{y = b} + Wx + U \tanh(d + Hx)$ 对于Contex词先经过一个非线性变换，然后再映射成一个词典维度的向量 (1)

where the hyperbolic tangent tanh is applied element by element, W is optionally zero (no direct connections), and x is the word features layer activation vector, which is the concatenation of the input word features from the matrix C:

C表示的是输入Embedding矩阵

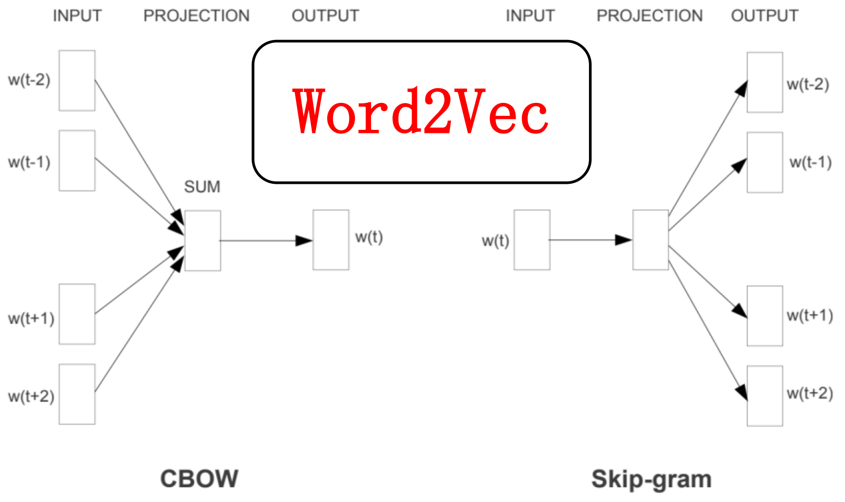
$$x = (C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1})).$$

Bengio 2003年提出的 A Neural Probabilistic Language Model



注意：
此时的logits计算方式是通过神经网络(DNN)计算，后面注意和Word2Vec计算logits的方式进行区分

Word2Vec



Skip-Gram正样本通过滑窗构建，负样本随机采样

Window Size	Text	Skip-grams
2	[The **wide** road shimmered] in the hot sun.	wide, the wide, road wide, shimmered
	The [wide road **shimmered** in the] hot sun.	shimmered, wide shimmered, road shimmered, in shimmered, the
	The wide road shimmered in [the hot **sun**].	sun, the sun, hot

Skip-Gram的学习目标：最大化，给定target词预测contex词的条件概率（最大似然估计）

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

条件概率的计算方式为：

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left({v'_w}^\top v_{w_I}\right)}$$

思考几个问题：

（1）word2vec和语言模型的logits计算方式有什么不同？（前者是向量内积，后者是DNN计算）

（2）word2vec和语言模型的优化目标有什么联系？（前者是最大化窗口内的词的条件概率，后者是最大化前n-1个词的条件概率）

TF实现Word2Vec模型

```
class Word2Vec(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(Word2Vec, self).__init__()
    self.target_embedding = layers.Embedding(vocab_size,
                                      embedding_dim,
                                      input_length=1,
                                      name="w2v_embedding")
    self.context_embedding = layers.Embedding(vocab_size,
                                       embedding_dim,
                                       input_length=num_ns+1)

  def call(self, pair):
    target, context = pair
    # target: (batch, dummy?)  # The dummy axis doesn't exist in TF2.7+
    # context: (batch, context)
    if len(target.shape) == 2:
      target = tf.squeeze(target, axis=1)
    # target: (batch,)
    word_emb = self.target_embedding(target)
    # word_emb: (batch, embed)
    context_emb = self.context_embedding(context)
    # context_emb: (batch, context, embed)
    dots = tf.einsum('be,bce->bc', word_emb, context_emb)
    # dots: (batch, context)
    return dots
```

样本构建代码

```
def skipgrams(sequence, vocabulary_size,
              window_size=4, negative_samples=1., shuffle=True,
              categorical=False, sampling_table=None, seed=None):
    """Generates skipgram word pairs.
    """
    couples = []
    labels = []
    for i, wi in enumerate(sequence):
        if not wi:
            continue
        if sampling_table is not None:
            if sampling_table[wi] < random.random():
                continue

        window_start = max(0, i - window_size)
        window_end = min(len(sequence), i + window_size + 1)
        for j in range(window_start, window_end):
            if j != i:
                wj = sequence[j]
                if not wj:
                    continue
                couples.append([wi, wj])  (target, contex)
                if categorical:
                    labels.append([0, 1])
                else:
                    labels.append(1)

    if negative_samples > 0:
        num_negative_samples = int(len(labels) * negative_samples)
        words = [c[0] for c in couples]
        random.shuffle(words)

        couples += [[words[i % len(words)],
                     random.randint(1, vocabulary_size - 1)]
                    for i in range(num_negative_samples)]
        if categorical:
            labels += [[1, 0]] * num_negative_samples
        else:
            labels += [0] * num_negative_samples

    if shuffle:
        if seed is None:
            seed = random.randint(0, 10e6)
        random.seed(seed)
        random.shuffle(couples)
        random.seed(seed)
        random.shuffle(labels)

    return couples, labels
```

负样本为什么要采样呢？

（1）计算条件概率时需要计算一个softmax，其分母的计算需要遍历整个词典，训练效率太低了。

（2）对负样本进行采样是为了优化softmax的计算

Softmax计算如何优化？

（1）保持softmax层不变，**但是修改它的结构来提升计算效率（例如分层softmax）**

（2）通过采样的方法，**使用采样之后的损失函数来近似原softmax损失函数（后面只介绍这种优化方式）**

Logistic Regression

逻辑回归的模型(函数/假设)为：

$$h_\theta(x) = g(\theta^T x)$$

其中$g(z) = \frac{1}{1+e^{-z}}$为$sigmoid$函数，$x$为模型输入，$\theta$为模型参数，$h_\theta(x)$为模型预测输入$x$为正样本(类别为1)的概率，而y为输入$x$对应的真实类别(**只有类别0与类别1两种**)。其对应的损失函数如下：

$$\boxed{J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right]}$$

上述损失函数称为交叉熵($cross-entropy$)损失，也叫log损失。通过优化算法(SGD/Adam)极小化该损失函数，可确定模型参数θ。

logloss

Softmax Regression

softmax回归的模型(函数/假设)为：

$$h_\theta\left(x^{(i)}\right) = \begin{bmatrix} p\left(y^{(i)}=1\,|x^{(i)}\right) \\ p\left(y^{(i)}=2\,|x^{(i)}\right) \\ \vdots \\ p\left(y^{(i)}=k\,|x^{(i)}\right) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

其中$\theta_1, \theta_2, \cdots, \theta_k$为模型参数，$h_\theta\left(x^{(i)}\right)$表示第$i$个样本输入$x^{(i)}$属于各个类别的概率，且所有概率和为1。其对应的损失函数如下：

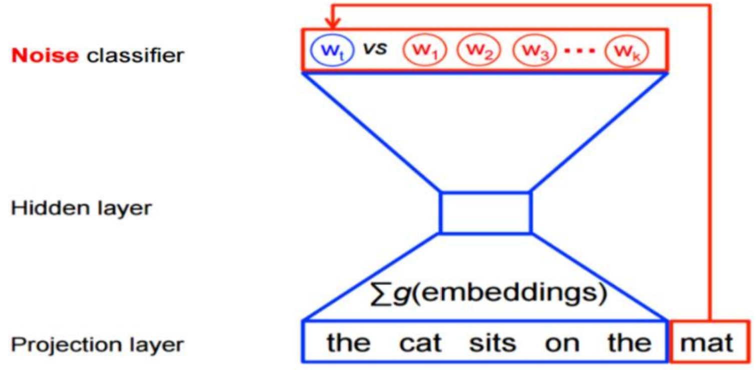
onehot编码，只有一个位置是1

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\boxed{\sum_{j=1}^{k} I\left(y^{(i)}=j\right)}\log\frac{e^{\theta_j^T x^{(i)}}}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}}\right]$$

其中$I\left(y^{(i)}=j\right)$表示第i个样本的标签值是否等于第j个类别，等于的话为1，否则为0。该损失函数与逻辑回归的具有相同的形式，**都是对概率取对数后与实际类别的one-hot编码进行逐位相乘再求和的操作**，最后记得加个负号。

Noise Contrastive Estimation(NCE)

NCE核心思想：将预测正确单词的问题简化为二分类任务，在该任务中，模型试图从噪声样本中区分真实的数据(二分类问题)，如图所示(注意：早期NEC是用在语言模型中的)：



噪声对比：使得窗口内的样本的score比窗口外的score要更高，窗口内的样本就是target词和context词，窗口外就是target词和random的噪声词

记词w_i的上下文为c_i，$\tilde{w}_{ij}(j=1,2,\cdots,k)$为从某种噪音分布$Q$中生成的$k$个噪音词(从词表中采样生成)。则$(c_i, w_i)$构成了正样本($y=1$)，$(c_i, \tilde{w}_{ij})$构成了负样本($y=0$)。

基于上述描述，可用逻辑回归模型构造如下损失函数

近似softmax，遍历所有类别取值，但是对于二分类来说只有1和0，和逻辑回归类似

$$J_\theta = -\sum_{w_i \in V}\boxed{\left[\log P\left(y=1|c_i, w_i\right) + \sum_{j=1}^{k}\log P\left(y=0|c_i, \tilde{w}_{ij}\right)\right]}$$

上述损失函数中共有$k+1$个样本。可看成从两种不同的分布中分别采样得到的，一个是依据训练集的经验分布P_train每次从词表中采样一个目标样本，其依赖于上下文c；而另一个是依据噪音分布Q每次从词表中采样k个噪音样本(不包括目标样本)。基于上述两种分布，有如下混合分布时的采样概率：

$$P(y, w|c) = \frac{1}{k+1}P_{train}(w|c) + \frac{k}{k+1}Q(w)$$

更进一步地，有

Q(w)是一个噪声分布，是已知的常数，所以可以把前面的系数合并起来

$$P(y=1|w,c) = \frac{\boxed{\frac{1}{k+1}P_{train}(w|c)}}{\frac{1}{k+1}P_{train}(w|c) + \frac{k}{k+1}Q(w)} = \frac{P_{train}(w|c)}{P_{train}(w|c) + kQ(w)}$$

Noise Contrastive Estimation (NCE)

记词w_i的上下文为c_i，$\tilde{w}_{ij}(j=1,2,\cdots,k)$为从某种噪音分布$Q$中生成的$k$个噪音词(从词表中采样生成)。则$(c_i,w_i)$构成了正样本$(y=1)$，$(c_i,\tilde{w}_{ij})$构成了负样本$(y=0)$。

基于上述描述，可用逻辑回归模型构造如下损失函数

近似softmax，遍历所有类别取值，但是对于二分类来说只有1和0，和逻辑回归类似

$$J_\theta=-\sum_{w_i\in V}\boxed{\left[\log P\left(y=1|c_i,w_i\right)+\sum_{j=1}^{k}\log P\left(y=0|c_i,\tilde{w}_{ij}\right)\right]}$$

上述损失函数中共有$k+1$个样本。可看成从两种不同的分布中分别采样得到的，一个是依据训练集的经验分布P_train每次从词表中采样一个目标样本，其依赖于上下文c；而另一个是依据噪音分布Q每次从词表中采样k个噪音样本(不包括目标样本)。基于上述两种分布，有如下混合分布时的采样概率：

$$P\left(y,w|c\right)=\frac{1}{k+1}P_{train}\left(w|c\right)+\frac{k}{k+1}Q\left(w\right)$$

更进一步地，有

Q(w)是一个噪声分布，是已知的常数，所以可以把前面的系数合并起来

$$P\left(y=1|w,c\right)=\frac{\boxed{\frac{1}{k+1}P_{train}\left(w|c\right)}}{\frac{1}{k+1}P_{train}\left(w|c\right)+\frac{k}{k+1}Q\left(w\right)}=\frac{P_{train}\left(w|c\right)}{P_{train}\left(w|c\right)+kQ\left(w\right)}$$

条件概率表示如下：

$$P_{train}\left(w|c\right)=\frac{\exp\left(h^Tv_w'\right)}{\sum_{w_i\in V}\exp\left(h^Tv_{w_i}'\right)}$$

引入一个假设：将分母部分固定为1，实验发现并没有影响模型的性能，此外，通过实验对分母进行统计，发现分母的值真的是以一个较小的方差在1附近波动，此外，固定为1方便转化为逻辑回归的损失，最终条件概率：

$$P_{train}\left(w|c\right)=\exp\left(h^Tv_w'\right)$$

正样本的概率表示为：

$$P\left(y=1|w,c\right)=\frac{\exp\left(h^Tv_w'\right)}{\exp\left(h^Tv_w'\right)+kQ\left(w\right)}$$

损失函数表示为：

$$J_\theta=-\sum_{w_i\in V}\left[\log\frac{\exp\left(h^Tv_{w_i}'\right)}{\exp\left(h^Tv_{w_i}'\right)+kQ\left(w_i\right)}+\sum_{j=1}^{k}\log\left(1-\frac{\exp\left(h^Tv_{\tilde{w}_{ij}}'\right)}{\exp\left(h^Tv_{\tilde{w}_{ij}}'\right)+kQ\left(\tilde{w}_{ij}\right)}\right)\right]$$

注意：NCE具有很好的理论保证：**随着噪音样本数k的增加，NCE的导数趋向于softmax的梯度。** 有研究证明25个噪音样本足以匹配常规softmax的性能，且有45X的加速。

负采样：

负采样(NEG)可看成是NCE的近似估计，其并不保证趋向于softmax。因为NEG的目标是学习高质量的词表示，而不是语言模型中的低困惑度(perplexity【**句子概率越大，语言模型越好，迷惑度越小**】)。

负采样与NCE一样，也是以逻辑回归的损失函数为目标进行学习的。主要的区别在于将原先NCE的正样本概率表达式

$$P\left(y=1|w,c\right)=\frac{\exp\left(h^Tv_w'\right)}{\exp\left(h^Tv_w'\right)+kQ\left(w\right)}$$

NCE

$$P\left(y=1|w,c\right)=\frac{\exp\left(h^Tv_w'\right)}{\exp\left(h^Tv_w'\right)+1}\Rightarrow P(y=1|w,c)=\frac{1}{1+\exp(-h^\top v_w')}.$$

负采样

If we now insert this back into the logistic regression loss from before, we get:

$$J_\theta=-\sum_{w_i\in V}[\log\frac{1}{1+\exp(-h^\top v_{w_i}')}+\boxed{\sum_{j=1}^{k}\log(1-\frac{1}{1+\exp(-h^\top v_{\tilde{w}_{ij}}')}]}$$

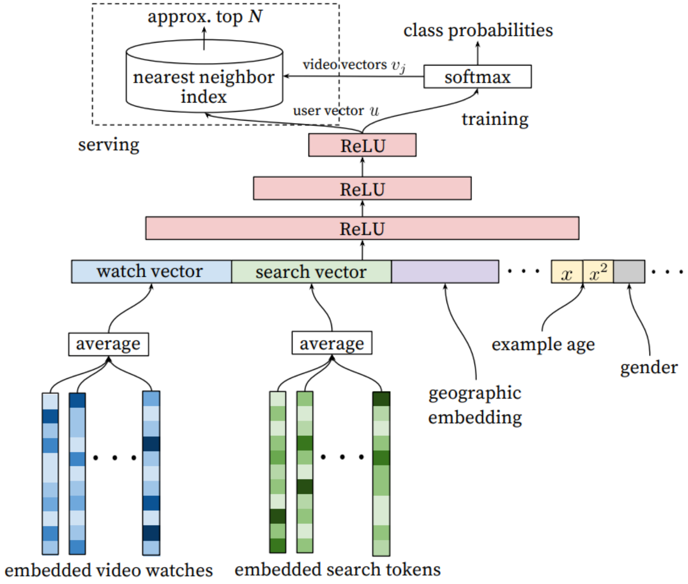
By simplifying slightly, we obtain:

$$J_\theta=-\sum_{w_i\in V}[\log\frac{1}{1+\exp(-h^\top v_{w_i}')}+\boxed{\sum_{j=1}^{k}\log(\frac{1}{1+\exp(h^\top v_{\tilde{w}_{ij}}')}].}$$

Setting $\boxed{\sigma(x)=\frac{1}{1+\exp(-x)}}$ finally yields the NEG loss:

$$\boxed{J_\theta=-\sum_{w_i\in V}[\log\sigma(h^\top v_{w_i}')+\sum_{j=1}^{k}\log\sigma(-h^\top v_{\tilde{w}_{ij}}')].}$$

YoutueDNN召回中的负采样



优化目标：将召回问题看成多分类问题

$$P(w_t = i|U, C) = \boxed{\frac{e^{v_i u}}{\sum_{j \in V} e^{v_j u}}}$$

通过重要性采样（蒙特卡洛方法）近似Softmax计算：

YoutubeDNN原论文说，借鉴《On Using Very Large Target Vocabulary for Neural Machine Translation》论文中的重要性采样来优化的，下面简单分析一下

原问题（神经网络机器翻译问题）：

The probability of the next target word in Eq. (2) is then computed by

$$p(y_t \mid y_{<t}, x) = \boxed{\frac{1}{Z}} \exp\left\{\mathbf{w}_t^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_t\right\}, \quad (6)$$

where ϕ is an affine transformation followed by a nonlinear activation, and $\mathbf{w}_t$ and b_t are respectively the *target word vector* and the target word bias. Z is the normalization constant computed by

$$Z = \boxed{\sum_{k: y_k \in V}} \exp\left\{\mathbf{w}_k^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_k\right\}, \quad (7)$$

where V is the set of all the target words.

Approximate Learning Approach：

As mentioned earlier, the computational inefficiency of training a neural machine translation model arises from the normalization constant in Eq. (6). In order to avoid the growing complexity of computing the normalization constant, we propose here to use only a small subset V' of the target vocabulary at each update. The proposed approach is based on the earlier work of (Bengio and Sénécal, 2008).

Let us consider the gradient of the log-probability of the output in Eq. (6). The gradient is composed of a positive and negative part:

$$\nabla \log p(y_t \mid y_{<t}, x) \quad (8)$$

$$= \underbrace{\nabla \mathcal{E}(y_t)}_{\text{原分子部分}} - \underbrace{\boxed{\sum_{k: y_k \in V} p(y_k \mid y_{<t}, x) \nabla \mathcal{E}(y_k)}}_{\text{原分母部分}},$$

整体是梯度的期望

where we define the energy $\mathcal{E}$ as

$$\boxed{\mathcal{E}(y_j) = \mathbf{w}_j^\top \phi\left(y_{j-1}, z_j, c_j\right) + b_j.}$$

The second, or negative, term of the gradient is in essence the expected gradient of the energy:

$$\mathbb{E}_P\left[\nabla \mathcal{E}(y)\right], \quad (9)$$

where P denotes $p(y \mid y_{<t}, x)$.

The main idea of the proposed approach is to approximate this expectation, or the negative term of the gradient, by importance sampling with a small number of samples. Given a predefined proposal distribution Q and a set V' of samples from Q, we approximate the expectation in Eq. (9) with

$$\mathbb{E}_P\left[\nabla \mathcal{E}(y)\right] \approx \boxed{\sum_{k: y_k \in V'}} \frac{\boxed{\omega_k}}{\sum_{k': y_{k'} \in V'} \omega_{k'}} \nabla \mathcal{E}(y_k), \quad (10)$$

where

$$\boxed{\omega_k = \exp\left\{\mathcal{E}(y_k) - \log Q(y_k)\right\}.} \quad (11)$$

This may be understood as having a separate proposal distribution Q_i for each partition of the training corpus. The distribution Q_i assigns equal probability mass to all the target words included in the subset V_i', and zero probability mass to all the other words, i.e.,

$$Q_i(y_k) = \begin{cases} \frac{1}{|V_i'|} & \text{if } y_t \in V_i' \\ 0 & \text{otherwise.} \end{cases}$$

This choice of proposal distribution cancels out the correction term $-\log Q(y_k)$ from the importance weight in Eqs. (10)–(11), which makes the proposed approach equivalent to approximating the exact output probability in Eq. (6) with

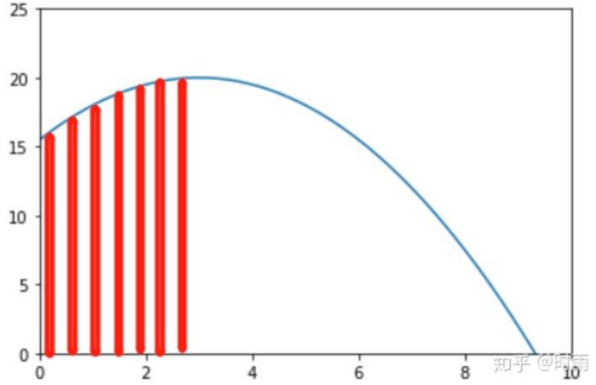
$$p(y_t \mid y_{<t}, x) = \frac{\exp\left\{\mathbf{w}_t^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_t\right\}}{\boxed{\sum_{k: y_k \in V'}} \exp\left\{\mathbf{w}_k^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_k\right\}}.$$

It should be noted that this choice of Q makes the estimator biased.

蒙特卡洛方法求积分

首先，当我们想要求一个函数 $f(x)$ 在区间 $[a,b]$ 上的积分 $\int_a^b f(x)dx$ 时有可能会面临一个问题，那就是积分曲线难以解析，无法直接求积分。这时候我们可以采用一种估计的方式，即在区间 $[a,b]$ 上进行采样： $\{x_1,x_2\ldots,x_n\}$ ，值为 $\{f(x_1),f(x_2),\ldots,f(x_n)\}$

如果采样是均匀的，即如下图所示：

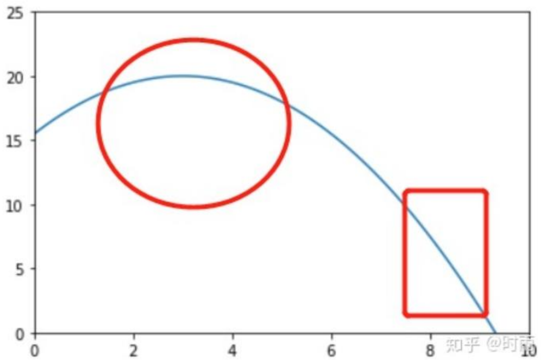


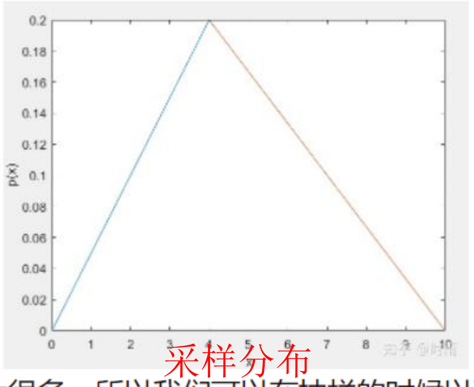
均匀采样

那么显然可以得到这样的估计： $\int_a^b f(x)dx = \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)$ ，在这里 $\frac{b-a}{N}$ 可以看作是上面小长方形的底部的“宽”，而 $f(x_i)$ 则是竖直的“长”。

重要性采样（蒙特卡洛方法求积分的一种策略）

上述的估计方法随着取样数的增长而越发精确，那么有什么方法能够在一定的抽样数量基础上来增加准确度，减少方差呢？这就需要我们人为地对抽样的分布进行干预，首先我们看下图：





很明显在圆形区域的函数值对积分的贡献比方形区域要大很多，所以我们可以在抽样的时候以更大的概率抽取圆形区域的样本，这样一来就能够提高估计的准确度。假设我们以分布 $p(x)$ 在原函数上进行采样：

依照这个分布进行采样我们一定程度上可以使得在原函数对积分贡献大的区域获得更多的采样机会。但这时我们不能对 $\{f(x_1),f(x_2),\ldots,f(x_n)\}$ 进行简单的求和平均来获得估计值，因为此时采样不是均匀分布的，小矩形的“宽”并不等长，所以我们要对其进行加权，这个权重就是**重要性权重。**

在得到重要性权重之前我们要重新思考一个问题：为什么我们要引入一个新的分布 $p(x)$ ？

原因就是原函数 $f(x)$ 也许本身就是定义在一个分布之上的，我们定义这个分布为 $\pi(x)$ ，我们无法直接从 $\pi(x)$ 上进行采样，所以另辟蹊径重新找到一个更加简明的分布 $p(x)$ ，从它进行取样，希望间接地求出 $f(x)$ 在分布 $\pi(x)$ 下的期望。

搞清楚了这一点我们可以继续分析了。首先我们知道函数 $f(x)$ 在概率分布 $\pi(x)$ 下的期望为：$E[f]=\int_x \pi(x)f(x)dx$ ，但是这个期望的值我们无法直接得到，因此我们需要借助 $p(x)$ 来进行采样，当我们在 $p(x)$ 上采样 $\{x_1,x_2,\ldots,x_n\}$ 后可以估计 f 在分布 $p(x)$ 下的期望为：

$E[f]=\int_x p(x)f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$ 。

接着我们可以对式子进行改写，即： $\pi(x)f(x)=p(x)\frac{\pi(x)}{p(x)}f(x)$ ，所以我们可以得到：

$$\boxed{E[f]=\int_x p(x)\frac{\pi(x)}{p(x)}f(x)dx}$$

这个式子我们可以看作是函数 $\frac{\pi(x)}{p(x)}f(x)$ 定义在分布 $p(x)$ 上的期望，当我们在 $p(x)$ 上采样 $\{x_1,x_2,\ldots,x_n\}$ 后可以估计 f 的期望 $E[f]=\frac{1}{N}\sum_{i=1}^{N}\frac{\pi(x_i)}{p(x_i)}f(x_i)$ ，**在这里** $\frac{\pi(x_i)}{p(x_i)}$ **就是重要性权重。**

The probability of the next target word in Eq. (2) is then computed by

$$p(y_t \mid y_{<t}, x) = \boxed{\frac{1}{Z}} \exp\left\{\mathbf{w}_t^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_t\right\}, \tag{6}$$

where ϕ is an affine transformation followed by a nonlinear activation, and $\mathbf{w}_t$ and b_t are respectively the *target word vector* and the target word bias. Z is the normalization constant computed by

$$Z = \boxed{\sum_{k:y_k \in V}} \exp\left\{\mathbf{w}_k^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_k\right\}, \tag{7}$$

where V is the set of all the target words.

Let us consider the gradient of the log-probability of the output in Eq. (6). The gradient is composed of a positive and negative part:

$$\nabla \log p(y_t \mid y_{<t}, x) \tag{8}$$
$$= \underbrace{\nabla \mathcal{E}(y_t)}_{\text{原分子部分}} - \boxed{\underbrace{\sum_{k:y_k \in V}}_{\text{原分母部分}} \underbrace{p(y_k \mid y_{<t}, x) \nabla \mathcal{E}(y_k)}_{\text{整体是梯度的期望}}},$$

where we define the energy $\mathcal{E}$ as

$$\boxed{\mathcal{E}(y_j) = \mathbf{w}_j^\top \phi\left(y_{j-1}, z_j, c_j\right) + b_j.}$$

The second, or negative, term of the gradient is in essence the expected gradient of the energy:

$$\mathbb{E}_P\left[\nabla \mathcal{E}(y)\right], \tag{9}$$

where P denotes $p(y \mid y_{<t}, x)$.

The main idea of the proposed approach is to approximate this expectation, or the negative term of the gradient, by importance sampling with a small number of samples. Given a predefined proposal distribution Q and a set V' of samples from Q, we approximate the expectation in Eq. (9) with

$$\mathbb{E}_P\left[\nabla \mathcal{E}(y)\right] \approx \boxed{\sum_{k:y_k \in V'}} \frac{\boxed{\omega_k}}{\sum_{k':y_{k'} \in V'} \omega_{k'}} \nabla \mathcal{E}(y_k), \tag{10}$$

where

$$\boxed{\omega_k = \exp\left\{\mathcal{E}(y_k) - \log Q(y_k)\right\}.} \tag{11}$$

问题描述：

$$P(y_t) = \frac{\exp[\mathcal{E}(y_t)]}{\sum_{y_k \in V} \exp[\mathcal{E}(y_k)]}$$

其中，$P(y_t) = P(y_t \mid y_{<t}, x)$，$\mathcal{E}(y_t) = W_t^T \phi(y_{t-1}, Z_t, C_t) + b_t$

将 $P(y_t)$ 转换成 log 形式：

$$\log P(y_t) = \mathcal{E}(y_t) - \log \sum_{y_k \in V} \exp[\mathcal{E}(y_k)]$$

对上式进行求导：

$$\nabla \log P(y_t) = \nabla \mathcal{E}(y_t) - \frac{1}{\sum_{y_k \in V} \exp[\mathcal{E}(y_k)]} \cdot \sum_{y_k \in V} \exp[\mathcal{E}(y_k)] \cdot \nabla \mathcal{E}(y_k)$$

$$= \nabla \mathcal{E}(y_t) - \sum_{y_k \in V} \frac{\exp[\mathcal{E}(y_k)]}{\sum_{y_k \in V} \exp[\mathcal{E}(y_k)]} \cdot \nabla \mathcal{E}(y_k)$$

$$= \nabla \mathcal{E}(y_t) - E[\nabla \mathcal{E}(y_k)]$$

由重要性采样原理，采样 N 个点近似求期望：

$$E[\nabla \mathcal{E}(y_k)] \approx \frac{1}{N} \sum_{y_i \in Q} \frac{P(y_i)}{Q(y_i)} \nabla \mathcal{E}(y_i)$$

此时，$Q(y_i)$ 是一个已知的分布，$P(y_i) = \frac{\exp[\mathcal{E}(y_i)]}{\sum_{y_j \in V} \exp[\mathcal{E}(y_j)]}$

$P(y_i)$ 的分母 $\sum_{y_j \in V} \exp[\mathcal{E}(y_j)] = M \cdot \underbrace{\sum_{y_j \in V} \frac{1}{M} \cdot \exp[\mathcal{E}(y_j)]}_{\text{将其看成期望.}}$

令 $R(y_i) = \frac{1}{M}$，分母 $= M \cdot \sum_{y_i \in V} R(y_i) \cdot \exp[\mathcal{E}(y_i)]$

分母 $= M \cdot \sum_{y_i \in V} R(y_i) \cdot \exp[\mathcal{E}(y_i)] = M \cdot E[\exp[\mathcal{E}(y_i)]]$

由重要性采样原理，采样 N 个点，近似求解上述期望.

$$\text{分母} \approx M \cdot \frac{1}{N} \sum_{y_j \in Q} \frac{R(y_j)}{Q(y_j)} \cdot \exp[\mathcal{E}(y_j)]$$

$$= \frac{M}{N} \sum_{y_j \in Q} \frac{\frac{1}{M} \cdot \exp[\mathcal{E}(y_j)]}{Q(y_j)} = \frac{1}{N} \sum_{y_j \in Q} \frac{\exp[\mathcal{E}(y_j)]}{Q(y_j)}$$

将分母代入 $P(y_i)$ 有：

$$P(y_i) = \frac{\exp[\mathcal{E}(y_i)]}{\frac{1}{N} \sum_{y_j \in Q} \frac{\exp[\mathcal{E}(y_j)]}{Q(y_j)}}$$

将 $P(y_i)$ 代入 $E[\nabla \mathcal{E}(y_k)]$ 有：

$$E[\nabla \mathcal{E}(y_k)] \approx \frac{1}{N} \sum_{y_i \in Q} \frac{P(y_i)}{Q(y_i)} \cdot \nabla \mathcal{E}(y_i)$$

$$= \frac{1}{N} \sum_{y_i \in Q} \frac{\frac{\exp[\mathcal{E}(y_i)]}{\frac{1}{N} \sum_{y_j \in Q} \frac{\exp[\mathcal{E}(y_j)]}{Q(y_j)}}}{Q(y_i)} \nabla \mathcal{E}(y_i)$$

$$= \sum_{y_i \in Q} \frac{\exp[\mathcal{E}(y_i)]}{Q(y_i) \cdot \sum_{y_j \in Q} \frac{\exp[\mathcal{E}(y_j)]}{Q(y_j)}} \cdot \nabla \mathcal{E}(y_i)$$

$$= \sum_{y_i \in Q} \frac{\exp[\mathcal{E}(y_i) - \log Q(y_i)]}{\sum_{y_j \in Q} \exp[\mathcal{E}(y_j) - \log Q(y_j)]} \cdot \nabla \mathcal{E}(y_i)$$

对比原期望：$E[\nabla \mathcal{E}(y_k)] = \sum_{y_k \in V} P(y_k) \cdot \nabla \mathcal{E}(y_k)$

$$= \sum_{y_k \in V} \frac{\exp[\mathcal{E}(y_k)]}{\sum_{y_k \in V} \exp[\mathcal{E}(y_k)]} \cdot \nabla \mathcal{E}(y_k)$$

所以原 softmax 被通过引入一个已知分布采样的样本近似为：

```python
@tf_export(v1=["nn.nce_loss"])
def nce_loss(weights,
             biases,
             labels,
             inputs,
             num_sampled,
             num_classes,
             num_true=1,
             sampled_values=None,
             remove_accidental_hits=False,
             partition_strategy="mod",
             name="nce_loss"):
  """Computes and returns the noise-contrastive estimation training loss
  """
  logits, labels = _compute_sampled_logits(
      weights=weights,
      biases=biases,
      labels=labels,
      inputs=inputs,
      num_sampled=num_sampled,
      num_classes=num_classes,
      num_true=num_true,
      sampled_values=sampled_values,
      subtract_log_q=True,
      remove_accidental_hits=remove_accidental_hits,
      partition_strategy=partition_strategy,
      name=name)
  sampled_losses = sigmoid_cross_entropy_with_logits(
      labels=labels, logits=logits, name="sampled_losses")
  # sampled_losses is batch_size x {true_loss, sampled_losses...}
  # We sum out true and sampled losses.
  return _sum_rows(sampled_losses)
@tf_export(v1=["nn.sampled_softmax_loss"])
def sampled_softmax_loss(weights,
                         biases,
                         labels,
                         inputs,
                         num_sampled,
                         num_classes,
                         num_true=1,
                         sampled_values=None,
                         remove_accidental_hits=True,
                         partition_strategy="mod",
                         name="sampled_softmax_loss",
                         seed=None):
  """Computes and returns the sampled softmax training loss.
  """
  logits, labels = _compute_sampled_logits(
      weights=weights,
      biases=biases,
      labels=labels,
      inputs=inputs,
      num_sampled=num_sampled,
      num_classes=num_classes,
      num_true=num_true,
      sampled_values=sampled_values,
      subtract_log_q=True,
      remove_accidental_hits=remove_accidental_hits,
      partition_strategy=partition_strategy,
      name=name,
      seed=seed)
  labels = array_ops.stop_gradient(labels, name="labels_stop_gradient")
  sampled_losses = nn_ops.softmax_cross_entropy_with_logits_v2(
      labels=labels, logits=logits)
  # sampled_losses is a [batch_size] tensor.
  return sampled_losses
```

```python
def _compute_sampled_logits(weights,
                            biases,
                            labels,
                            inputs,
                            num_sampled,
                            num_classes,
                            num_true=1,
                            sampled_values=None,
                            subtract_log_q=True,
                            remove_accidental_hits=False,
                            partition_strategy="mod",
                            name=None,
                            seed=None):
  """Helper function for nce_loss and sampled_softmax_loss functions.
  """

  if isinstance(weights, variables.PartitionedVariable):
    weights = list(weights)
  if not isinstance(weights, list):
    weights = [weights]

  with ops.name_scope(name, "compute_sampled_logits",
                      weights + [biases, inputs, labels]):
    if labels.dtype != dtypes.int64:
      labels = math_ops.cast(labels, dtypes.int64)
    labels_flat = array_ops.reshape(labels, [-1])

    # Sample the negative labels.
    #   sampled shape: [num_sampled] tensor
    #   true_expected_count shape = [batch_size, 1] tensor
    #   sampled_expected_count shape = [num_sampled] tensor
    if sampled_values is None:
      sampled_values = candidate_sampling_ops.log_uniform_candidate_sampler(
          true_classes=labels,
          num_true=num_true,
          num_sampled=num_sampled,
          unique=True,
          range_max=num_classes,
          seed=seed)
    # NOTE: pylint cannot tell that 'sampled_values' is a sequence
    # pylint: disable=unpacking-non-sequence
    sampled, true_expected_count, sampled_expected_count = (
        array_ops.stop_gradient(s) for s in sampled_values)
    # pylint: enable=unpacking-non-sequence
    sampled = math_ops.cast(sampled, dtypes.int64)

    # labels_flat is a [batch_size * num_true] tensor
    # sampled is a [num_sampled] int tensor
    all_ids = array_ops.concat([labels_flat, sampled], 0)

    # Retrieve the true weights and the logits of the sampled weights.

    # weights shape is [num_classes, dim]
    all_w = embedding_ops.embedding_lookup(
        weights, all_ids, partition_strategy=partition_strategy)
    if all_w.dtype != inputs.dtype:
      all_w = math_ops.cast(all_w, inputs.dtype)

    # true_w shape is [batch_size * num_true, dim]
    true_w = array_ops.slice(all_w, [0, 0],
                             array_ops.stack(
                                 [array_ops.shape(labels_flat)[0], -1]))

    sampled_w = array_ops.slice(
        all_w, array_ops.stack([array_ops.shape(labels_flat)[0], 0]), [-1, -1])
```

负样本采样

采样分布Q

```python
    # inputs has shape [batch_size, dim]
    # sampled_w has shape [num_sampled, dim]
    # Apply X*W', which yields [batch_size, num_sampled]
    sampled_logits = math_ops.matmul(inputs, sampled_w, transpose_b=True)

    # Retrieve the true and sampled biases, compute the true logits, and
    # add the biases to the true and sampled logits.
    all_b = embedding_ops.embedding_lookup(
        biases, all_ids, partition_strategy=partition_strategy)
    if all_b.dtype != inputs.dtype:
      all_b = math_ops.cast(all_b, inputs.dtype)
    # true_b is a [batch_size * num_true] tensor
    # sampled_b is a [num_sampled] float tensor
    true_b = array_ops.slice(all_b, [0], array_ops.shape(labels_flat))
    sampled_b = array_ops.slice(all_b, array_ops.shape(labels_flat), [-1])

    # inputs shape is [batch_size, dim]
    # true_w shape is [batch_size * num_true, dim]
    # row_wise_dots is [batch_size, num_true, dim]
    dim = array_ops.shape(true_w)[1:2]
    new_true_w_shape = array_ops.concat([[-1, num_true], dim], 0)
    row_wise_dots = math_ops.multiply(
        array_ops.expand_dims(inputs, 1),
        array_ops.reshape(true_w, new_true_w_shape))
    # We want the row-wise dot plus biases which yields a
    # [batch_size, num_true] tensor of true_logits.
    dots_as_matrix = array_ops.reshape(row_wise_dots,
                                       array_ops.concat([[-1], dim], 0))
    true_logits = array_ops.reshape(_sum_rows(dots_as_matrix), [-1, num_true])
    true_b = array_ops.reshape(true_b, [-1, num_true])
    true_logits += true_b
    sampled_logits += sampled_b

    if remove_accidental_hits:
      acc_hits = candidate_sampling_ops.compute_accidental_hits(
          labels, sampled, num_true=num_true)
      acc_indices, acc_ids, acc_weights = acc_hits

      # This is how SparseToDense expects the indices.
      acc_indices_2d = array_ops.reshape(acc_indices, [-1, 1])
      acc_ids_2d_int32 = array_ops.reshape(
          math_ops.cast(acc_ids, dtypes.int32), [-1, 1])
      sparse_indices = array_ops.concat([acc_indices_2d, acc_ids_2d_int32], 1,
                                        "sparse_indices")
      # Create sampled_logits_shape = [batch_size, num_sampled]
      sampled_logits_shape = array_ops.concat(
          [array_ops.shape(labels)[:1],
           array_ops.expand_dims(num_sampled, 0)], 0)
      if sampled_logits.dtype != acc_weig
        acc_weights = math_ops.cast(acc_w
      sampled_logits += gen_sparse_ops.sp
          sparse_indices,
          sampled_logits_shape,
          acc_weights,
          default_value=0.0,
          validate_indices=False
```

```python
    if subtract_log_q:
      # Subtract log of Q(l), prior probability that l appears in sampled.
      true_logits -= math_ops.log(true_expected_count)
      sampled_logits -= math_ops.log(sampled_expected_count)

    # Construct output logits and labels. The true labels/logits start at col 0.
    out_logits = array_ops.concat([true_logits, sampled_logits], 1)

    # true_logits is a float tensor, ones_like(true_logits) is a float
    # tensor of ones. We then divide by num_true to ensure the per-example
    # labels sum to 1.0, i.e. form a proper probability distribution.
    out_labels = array_ops.concat([
        array_ops.ones_like(true_logits) / num_true,
        array_ops.zeros_like(sampled_logits)
    ], 1)

    return out_logits, out_labels
```

问题1：
没有传入采样相关的频率信息，代码是如何实现log-uniform采样的呢？

The base distribution for this operation is an approximately log-uniform or Zipfian distribution:

```
P(class) = (log(class + 2) - log(class + 1)) / log(range_max + 1)
```

This sampler is useful when the target classes approximately follow such a distribution - for example, if the classes represent words in a lexicon sorted in decreasing order of frequency. If your classes are not ordered by decreasing frequency, do not use this op.

问题2：
如何自定义采样分布呢？

```python
tf.random.fixed_unigram_candidate_sampler(
    true_classes, num_true, num_sampled, unique, range_max, vocab_file='',
    distortion=1.0, num_reserved_ids=0, num_shards=1, shard=0, unigrams=(),
    seed=None, name=None
)
```

vocab_file

Each valid line in this file (which should have a CSV-like format) corresponds to a valid word ID. IDs are in sequential order, starting from num_reserved_ids. The last entry in each line is expected to be a value corresponding to the count or relative probability. Exactly one of vocab_file and unigrams needs to be passed to this operation.

负采样与重要性采样如何选择

负采样
NEG

$$J_\theta = -\sum_{w_i \in V} [\log \frac{1}{1+\exp(-h^\top v'_{w_i})} + \sum_{j=1}^{k} \log(\frac{1}{1+\exp(h^\top v'_{\tilde{w}_{ij}})}].$$

$$J_\theta = -\sum_{w_i \in V} [\log\sigma(h^\top v'_{w_i}) + \sum_{j=1}^{k} \log\sigma(-h^\top v'_{\tilde{w}_{ij}})].$$

本质上等价于多个二分类，适合用于label有多个类别的场景，例如Skip-Gram及其相关的召回模型（Item2Vec，EGES，Airbnb等）

重要性采样
sampled_softmax_loss

$$p(y_t \mid y_{<t}, x) = \frac{\exp\left\{\mathbf{w}_t^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_t\right\}}{\sum_{k: y_k \in V'} \exp\left\{\mathbf{w}_k^\top \phi\left(y_{t-1}, z_t, c_t\right) + b_k\right\}}.$$

$$\mathbb{E}_P\left[\nabla \mathcal{E}(y)\right] \approx \sum_{k: y_k \in V'} \boxed{\frac{\omega_k}{\sum_{k': y_{k'} \in V'} \omega_{k'}}} \nabla \mathcal{E}(y_k), \quad (10)$$

矫正之后的采样的softmax

where

$$\omega_k = \boxed{\exp\left\{\mathcal{E}(y_k) - \log Q(y_k)\right\}}. \quad (11)$$

本质上还是一个softmax，让某个类别的概率最大，适用于lable只有单个类别的场景，例如CBOW及相关的召回模型（YoutubeDNN，MIND，SDM等）

Table of Candidate Sampling Algorithms

	Positive training classes associated with training example (x_i, T_i): $POS_i =$	Negative training classes associated with training example (x_i, T_i): $NEG_i =$	Input to Training Loss $G(x,y) =$	Training Loss	$F(x,y)$ gets trained to approximate:
Noise Contrastive Estimation (NCE)	T_i	S_i	$F(x,y) - log(Q(y\|x))$	Logistic	$log(P(y\|x))$
Negative Sampling	T_i	S_i	$F(x,y)$	Logistic	$log\left(\frac{P(y\|x)}{Q(y\|x)}\right)$
Sampled Logistic	T_i	$(S_i - T_i)$	$F(x,y) - log(Q(y\|x))$	Logistic	$logodds(y\|x) = log\left(\frac{P(y\|x)}{1-P(y\|x)}\right)$
Full Logistic	T_i	$(L - T_i)$	$F(x,y)$	Logistic	$log(odds(y\|x)) = log\left(\frac{P(y\|x)}{1-P(y\|x)}\right)$
Full Softmax	$T_i = \{t_i\}$	$(L - T_i)$	$F(x,y)$	Softmax	$log(P(y\|x)) + K(x)$
Sampled Softmax	$T_i = \{t_i\}$	$(S_i - T_i)$	$F(x,y) - log(Q(y\|x))$	Softmax	$log(P(y\|x)) + K(x)$

- $Q(y|x)$ is defined as the probability (or $\boxed{\text{expected count}}$) according to the sampling algorithm of the class y in the (multi-)set of sampled classes given the context x.
- $K(x)$ is an arbitrary function that does not depend on the candidate class. Since Softmax involves a normalization, addition of such a function does not affect the computed probabilities.
- $logistic\ training\ loss = \sum_i \left(\sum_{y \in POS_i} log(1 + exp(-G(x_i, y)) + \sum_{y \in NEG_i} log(1 + exp(G(x_i, y)) \right)$
- $softmax\ training\ loss = \sum_i \left(-G(x_i, t_i) + log \left(\sum_{y \in POS_i \cup NEG_i} exp(G(x_i, y)) \right) \right)$
- NCE and Negative Sampling generalize to the case where T_i is a multiset. In this case, $P(y|x)$ denotes the expected count of y in T_i. Similarly, NCE, Negative Sampling, and Sampled Logistic generalize to the case where S_i is a multiset. In this case $Q(y|x)$ denotes the expected count of y in S_i.

Q&A

扫描下方二维码关注公众号：Datawhale

知识星球

FunRec学习小组

若如意

星主

微信扫码加入星球