7 Skip-Gram的实现

由于CBOW的每个样本词只有一个正类标记，实现起来相对简单，因此选择更一般Skip-Gram实现，实际上在后文中将会看到，CBOW的实现就是Skip-Gram在指定上下文窗口为1时的特例。具体的代码放在了https://github.com/Y-WJ/NLP-PLAYER。本章主要的内容有：

1. 算法的详细介绍
2. 几种防止浮点数溢出的替代损失函数的推导
3. 训练损失的理论估计，与一些参数调整策略

7.1 文本的预处理

仿照word2vec的做法，按词频降序生成了列表count，并将所有未被计入词典的低频词用low-freq-word代替作为count的第0项，vocabulary-size参数指定了count的大小，这也是需要学习的词向量的数量。

按count的顺序生成了字典dictionary，以每个词作key，以词在count中的序号做value,为了方便根据序号反查对应的单词，生成了反向字典reverse-dictionary。

根据dictionary将训练文本中的所有单词转化为在count中的序号，得到word-number,接下来的用到训练文本的地方都用word-number代替。

7.2 训练样本的生成

采用批量梯度下降算法，用参数batch-size指定了一批样本的大小，参数sample-num指定了对每个样本所有采样（正负采样）的数量，联系似然公式：

$$G(C,\theta)=\prod_{w\in V}\prod_{u\in N_w}\{l_u^w\cdot sigmoid(\theta_u\cdot X_w^T)+(1-l_u^w)(1-sigmoid(\theta_u\cdot X_w^T))\}$$

可知batch-size=$|V|$,sample-num=$|N|$,一个batch的所有样本是batch-size大小的向量，而所有采样应该是(batch-size $\times$ sample-num)大小的矩阵。

参数context-widow指定了正采样的窗口也就是上下文窗口的大小，对当前词w来说，正采样策略是随机的在[-context-window,context-window]中选择一个不为0的整数 (positive-label-num)，总是以这个随机数到0之间的所有位置上的词作为w的正采样，比如context-window设置为6，随机数取到-3，则w之前的3个词被作为正采样写进采样矩阵的对应位置，随机数取到5，则w的之后的5个词作为正采样。这样做的好处是保留了w上下文的位置关系，因为越靠近w的词在w对应的采样序列里出现的越频繁，与之对应的想法是在w的上下文中的每个词与w的关系不是等重的，越靠近w的词权重应该越大。

参数sample-window指定了负采样的范围，只在count列表的前sample-window上做负采样，这相当于尽量取词频高的词做负标记，与之对应的想法是高频词与任意词的互熵更大，应该尽可能优先优化高频词的输出向量。

负采样的策略是在给定的sample-window上等概的选取总采样个数(sample-num)-正采样个数(positive-label-num)个词，NCE的理论只要求负采样的分布律值域覆盖正采样分布律的值域，这个要求弱到等于没有要求，因为对给定的一组样本，我们没法先验地判断它们所属的真实分布律，比如在count的前10个词随便取5个做负样本，我们同样可以把他当作在整个count上随机取得的5个负样本，不论是学习器还是人，在没有观察到取样过程的条件下是不可能断言那些样本属于那种分布的。

NCE理论给出了另一条指导意见：负采样的分布律越接近正采样的真实分布，理想的最优化值越优，且NCE损失的全局最优仅仅在负采样分布律等于正采样真实分布律时取得。在不知道真实分布的情况下

我们应该尽可能给出一个接近真实分布的采样。预测w的上下文有哪些词，比以等概的猜测字典中所有可能词更好的策略是根据词频猜测，将词典中每个词出现次数除以文本总词数作为这个词出现在所有词上下文的概率，负采样时按照sample-window中每个词的出现概率取样，这显然比随机乱取更接近正采样的真实分布。但在算法实现中这样的策略对损失下降的帮助似乎并不明显，而且在sample-window较大时，计算每个词的归一化概率，以及按概率取样的计算代价比随机乱取要大得多，因此算法实现采用简单的随机采样策略。（代码中在第28行提供了词频采样用的函数，但没有启用）

generate-batch函数用于从word-number中生成一批用于训练的样本向量，对应的采样矩阵以及标记采样属于正类或负类的标记矩阵（被初始化为全0），每次generate-batch从指向文本的全局循环指针pin(总是指向下一个要读入的词)开始读入一个正采样窗口，将采样窗口正中间的词作为样本w写入样本向量，根据随机数在上下文窗口采集正样本写入采样矩阵的对应行，并把标记矩阵对应位置上的标记置1，按负采样策略生成负样本填满采样矩阵在这一行中剩下的空位，一个词及其对应的采样和采样标记就完成了。循环上面的过程直到样本向量，和采样矩阵被填满。总之每次generate-batch返回三个矩阵：

1. batch-size×1 大小的样本矩阵V
2. batch-size×sample-num 大小的采样矩阵N
3. batch-size×sample-num 大小的标记矩阵l

7.3 损失计算

对于generate-batch生成的每一个batch,计算损失的过程如下：

1. 填充词向量，参数dim指定了词向量和输出向量的维度，对矩阵V中的每个词w选择对应的词向量X_w填充，对矩阵N中每个采样词u选择对应的输出向量θ_u填充。三个矩阵有如下表示:

$$V = (X_1^T, X_2^T, X_3^T, \cdots, X_{batch\_size}^T)$$

$$N = \begin{pmatrix} \theta_1^1 & \theta_2^1 & \theta_3^1 & \cdots & \theta_{sample\_num}^1 \\ \theta_1^2 & \theta_2^2 & \theta_3^2 & \cdots & \theta_{sample\_num}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_1^{batch\_size} & \theta_2^{batch\_size} & \theta_3^{batch\_size} & \cdots & \theta_{sample\_num}^{batch\_size} \end{pmatrix}$$

$$l = \begin{pmatrix} l_1^1 & l_2^1 & l_3^1 & \cdots & l_{sample\_num}^1 \\ l_1^2 & l_2^2 & l_3^2 & \cdots & l_{sample\_num}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_1^{batch\_size} & l_2^{batch\_size} & l_3^{batch\_size} & \cdots & l_{sample\_num}^{batch\_size} \end{pmatrix}$$

2. 计算网络输出$output$,由于矩阵的每一个元素都是向量，这实际上是一个二维矩阵不标准地乘以三维矩阵，不妨将这种运算记为∇，希望能得到结果(为了方便，记batch-size为b-s)：

$$output = \begin{pmatrix} \theta_1^1 \cdot X_1^T & \theta_2^1 \cdot X_1^T & \theta_3^1 \cdot X_1^T & \cdots & \theta_{sample\_num}^1 \cdot X_1^T \\ \theta_1^2 \cdot X_2^T & \theta_2^2 \cdot X_2^T & \theta_3^2 \cdot X_2^T & \cdots & \theta_{sample\_num}^2 \cdot X_2^T \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_1^{b\_s} \cdot X_{b\_s}^T & \theta_2^{b\_s} \cdot X_{b\_s}^T & \theta_3^{b\_s} \cdot X_{b\_s}^T & \cdots & \theta_{sample\_num}^{b\_s} \cdot X_{b\_s}^T \end{pmatrix}$$

为了实现这样的运算，考虑把$\theta_u * X_w$看做向量，$\theta_u * X_w$的每一维等于θ_u与X_w对应维度值的积，这

样只需对$V\nabla N$这个三维矩阵在z方向上求和降维，就能得到想要的运算结果了：

$$V\nabla N = \begin{pmatrix} \theta_1^1 * X_1^T & \theta_2^1 * X_1^T & \theta_3^1 * X_1^T & \cdots & \theta_{sample\_num}^1 * X_1^T \\ \theta_1^2 * X_2^T & \theta_2^2 * X_2^T & \theta_3^2 * X_2^T & \cdots & \theta_{sample\_num}^2 * X_2^T \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_1^{b\_s} * X_{b\_s}^T & \theta_2^{b\_s} * X_{b\_s}^T & \theta_3^{b\_s} * X_{b\_s}^T & \cdots & \theta_{sample\_num}^{b\_s} * X_{b\_s}^T \end{pmatrix}$$

$$\theta * X^T = (\theta_1 \cdot X_1, \theta_2 \cdot X_2, \theta_3 \cdot X_3, \cdots, \theta_{\dim} \cdot X_{\dim})$$

$$output = reduce\_sum\_z(V\nabla N) = \begin{pmatrix} \theta_1^1 \cdot X_1^T & \theta_2^1 \cdot X_1^T & \theta_3^1 \cdot X_1^T & \cdots & \theta_{sample\_num}^1 \cdot X_1^T \\ \theta_1^2 \cdot X_2^T & \theta_2^2 \cdot X_2^T & \theta_3^2 \cdot X_2^T & \cdots & \theta_{sample\_num}^2 \cdot X_2^T \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_1^{b\_s} \cdot X_{b\_s}^T & \theta_2^{b\_s} \cdot X_{b\_s}^T & \theta_3^{b\_s} \cdot X_{b\_s}^T & \cdots & \theta_{sample\_num}^{b\_s} \cdot X_{b\_s}^T \end{pmatrix}$$

为了描述算子∇，用(x, y, z)表示矩阵中元素的下标，那么能得到如下对应关系：

$$(V\nabla N)_{(x,y,z)} = V_{(x,z)} \cdot N_{(x,y,z)}$$
$$\nabla = (x, z) \cdot (x, y, z) \to (x, y, z)$$

3. 引入标记，由于我们的损失函数还要把$\theta_u \cdot X_w^T$与对应的标记l_u^w相乘，再次用算子Δ表示这种运算，显然有:

$$(output\Delta l)_{(x,y)} = output_{(x,y)} \cdot l_{(x,y)}$$
$$\Delta = (x, y) \cdot (x, y) \to (x, y)$$

4. 翻译损失函数，有了以上定义，可以把前一章推导得到的损失函数翻译成矩阵形式：

$$L = \sum_{w \in V} \sum_{u \in N_w} \{l_u^w \cdot \theta_u \cdot X_w^T - \theta_u \cdot X_w^T - \ln(1 + e^{-\theta_u \cdot X_w^T})\}$$

$$L = l\Delta output - output - \ln(1 + e^{-output})$$

这是个恒为负的只有上界的函数，为了用tensorflow提供的梯度下降优化，给损失添加符号翻转整个函数，最终得到的目标函数是：

$$L = -l\Delta output + output + \ln(1 + e^{-output})$$

这个目标函数理论上是可行的，但由浮点数限制，计算e的指数是非常危险的，实际实现的算法中也是如此，直接使用这个损失会在$output$是一个很大的负值时溢出，而且可以证明这样的溢出在损失足够小时一定会发生。解决这个问题的办法是选用等价而且没有危险计算的损失函数，这部分内容很多，不得不另起一节来详细地推导这些公式。

7.4 更安全的损失函数

1. 首先要说明的是为什么采用上一节推导的损失函数在损失足够小时一定会溢出。

先计算损失函数的理想值，根据上一章的假设，$sigmoid(\theta_u \cdot X_w^T)$是$u$在$w$上下文的概率，理想的损失仅仅发生在所有上下文词的预测概率为1且所有负采样词的预测概率为0,此时：

$$\begin{aligned} L &= \sum_{w \in V} \sum_{u \in N_w} \ln\{-l_u^w \cdot sigmoid(\theta_u \cdot X_w^T) + (1 - l_u^w)(sigmoid(\theta_u \cdot X_w^T) - 1)\} \\ &= \sum_{w \in V} \sum_{u \in N_w} \ln\{-l_u^w \cdot 1 + (1 - l_u^w)(0 - 1))\} \\ &= 0 \end{aligned}$$

记$\theta_u \cdot X_w^T = x$ 每个正采样词提供损失：

$$-\ln sigmoid(x) = \ln(1+e^{-x})$$

每个负采样词提供损失:

$$-\ln(1-sigmoid(x)) = \ln(1+e^{x})$$

在总损失趋于理想值0的情况下要求所有词提供的损失都趋于0，这意味着要求正类的x趋于正无穷，而负类的x趋于负无穷，但是当x是个较大的负值使计算e^{-x}必定上溢，因此上面的损失函数不可能优化得很小。

2. 一种替代方案，既然在优化的过程中正类的x被推向正无穷，负类的x被推向负无穷，那么分别对两类采样采用对应的负指数计算似乎是可行的，于是元损失函数可以变形成：

$$\begin{aligned} L &= -\sum_{w\in V}\sum_{u\in N_w}\{l_u^w \ln sigmoid(\theta_u \cdot X_w^T) + (1-l_u^w)\ln(1-sigmoid(\theta_u \cdot X_w^T))\} \\ &= \sum_{w\in V}\sum_{u\in N_w}\{l_u^w \ln(1+e^{-x}) + (1-l_u^w)\ln(1+e^{x}))\} \end{aligned}$$

在优化过程中，对应正采样的x计算负指数，对应负采样的x计算正指数，这样即使x的值被优化的很大，正负采样的指数计算都是趋于0的，这显然不会发生上溢了。

但采用这个损失函数的算法实际上还是会时不时的溢出，且负采样窗口设置的越大发生上溢的概率越大。定性的来说，这是由于负采样可能会采集到原本就是正样本的词，若这个正采样词与当前样本的x已经被优化到足够大的正值，而计算损失的时候仍被当作负样本计算正指数，就很可能溢出了。实际测试中，若负采样窗口设置在1000一下，这种情况几乎不会发生，这个替代函数能支撑到很小的损失而不溢出。但负采样窗口设置到1w以上，这个替代函数几乎不可能保证机器完整的跑完10W轮优化。（代码同样提供了这个损失函数，在不溢出时它的计算代价比算法启用的安全损失函数低不少）

3. 更好的替代方案，这是算法实际启用的损失函数。由第六章的损失函数：

$$L = \sum_{w\in V}\sum_{u\in N_w}\{-l_u^w \cdot \theta_u \cdot X_w^T + \theta_u \cdot X_w^T + \ln(1+e^{-\theta_u \cdot X_w^T})\}$$

对单个采样来说，如果是正类，损失为：

$$-lx + x + \ln(1+e^{x})$$

如果是负类，损失为：

$$x + \ln(1+e^{-x}) = x + \ln(\frac{e^x+1}{e^x}) = \ln(1+e^{x})$$

根据上对一个损失函数的讨论，一个正真的负类样本x会在训练中趋于负无穷，而一个真正的正类的x则会趋向正无穷，那么上面两个式子可以整合成一个：

$$L = \max(x,0) - lx + \ln(1+e^{-|x|})$$

可以看见不论对于x为正为负的真正正负类，这个式子的计算结果都是与原损失等价的，而且保证了不论正负类都计算它们的负指数，实际上采用这个损失后，算法没有在发生过上溢，当然有可能因为下溢损失了很多值，但这不会中断训练。唯一的瑕疵在于这个损失函数在$x=0$处是不可导的，但这影响并不大，如果不用初始化0值初始化，在之后的浮点数计算中用求导点刚好落在零点处的概率几乎可以忽略不计，这好比你在每次迭代后输出的损失几乎不可能刚好是个整数。

7.5 理论的损失值与参数调整策略

这一章没有涉及词向量的评价标准，一切调整均以使可能达到的损失最小为标准，但这并不意味着更小的损失等价于更好的词向量。

1. 初始化后的损失值的理论估计，为了尽量减少下溢带来的损失，算法的损失函数对所有采样的损失求和，但为了直观，在检查点输出的损失是所有采样的均值。对词向量矩阵和输出权重矩阵的初始化使用tensorflow自带的随机数发生器，均值为0，方差设置为1.0，任意词向量与输出权重向量的点积x的期望应该为0，在这样的条件下，初始化后的初始平均损失为：

$$\frac{L}{b\_s}=\frac{\sum_{w\in V}\sum_{u\in N_w}\{l_u^w\ln(1+e^{-0})+(1-l_u^w)\ln(1+e^{0})\}}{b\_s}=\ln(2)\approx 0.6931$$

   但在实际运行中，初始化后的损失分布在3.2附近，这可能是由于tensorflow提供的伪随机数发生器并不真的很随机，如果用常数0初始化矩阵，得到的结果是符合这里的计算的，0.6931可以作为一个参考值，损失低于这个值以后学习器才真正发挥了效果（比全0矩阵强一些），实际上负采样窗口小于迭代轮数时，这个算法可以很容易在将损失压到0.1以下。

2. 负采样数量对损失的影响非常大，从损失函数可以直观地看出，只需不断地增大负采样在所有采样中所占的比例，总能让损失总体损失更小。定性的看，若负采样数量远大于正采样，学习器的优化任务实际上变成了让所有的词在词向量空间中离的更远，只要空间中的词足够稀疏，对任意负样本都能取到非常大的x使损失趋近于0，而原少于负采样的正样本本应由于离样本词太远带来的损失就显得微乎其微了。或者笼统的说，负采样的数量决定了不相关词的区分度，而正采样的数量决定了相关词的聚合程度。正采样的数量增加（增加上下文窗口的大小）会显著提高损失，在相同的其他参数下，上下文窗口更大的模型需要更多次的迭代才能达到小窗口模型的损失，但就肉眼观察来看，大窗口模型的同类词聚集效果比小窗口好得多（当然这做不得准）。

3. 负采样窗口对计算代价的影响非常大，同样负采样窗口更大的模型需要更多次迭代才能达到相同的损失，但显然词的区分度会更好，如果只想把某些词同其他词区分开，完全可以只在这些需要区分的词范围内做负采样。

4. 可以由$batch_size\times(2context_window+1)\times train_step_num$估计学习器遍历的单词数量，在算法中设置了参数period用来记录训练结束时训练文本正处在第几轮遍历中。

5. 计算相似度，输出相似词，绘制降维图的部分照搬了word2vec的代码。

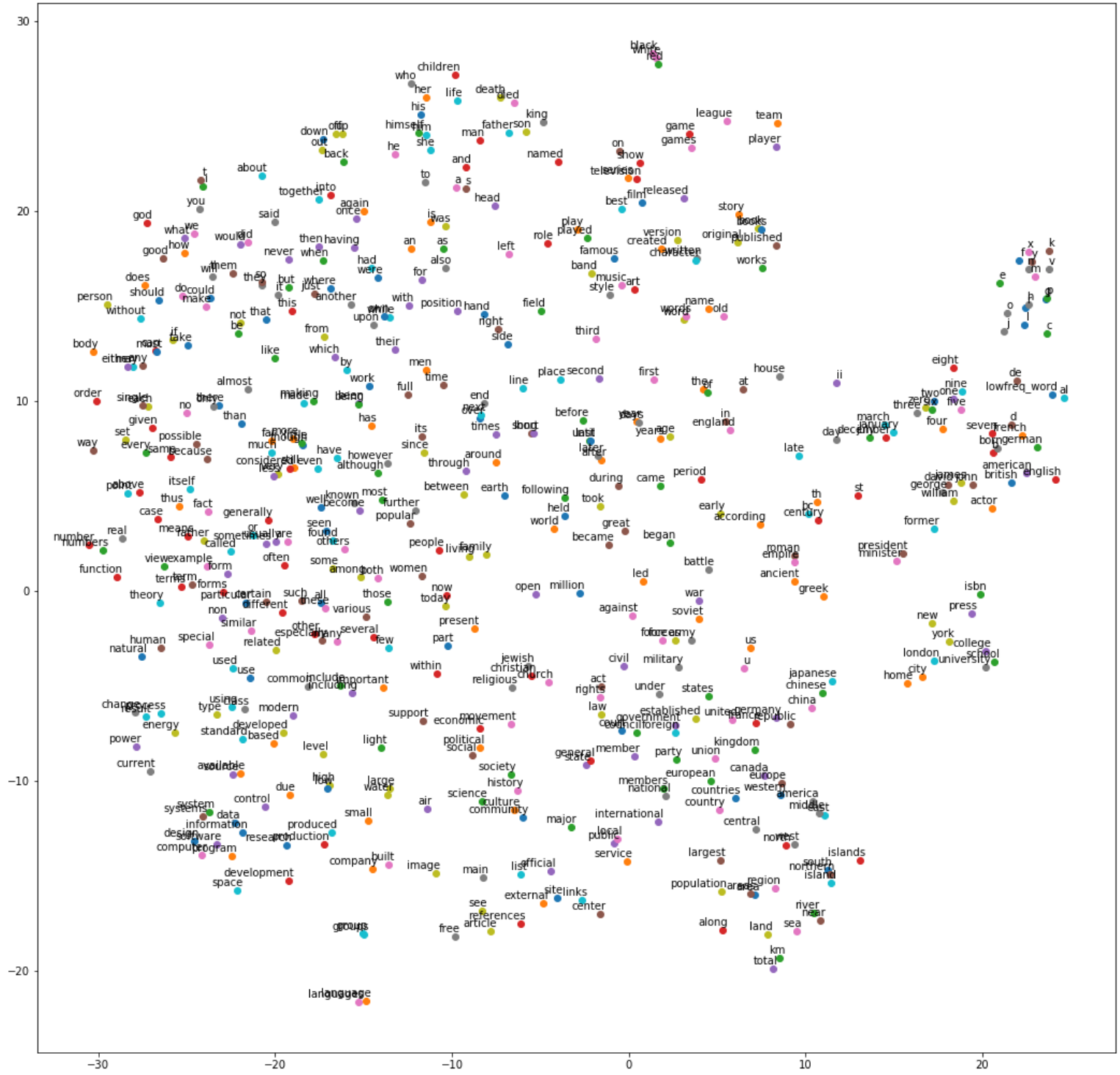

图 5: 10W轮训练效果图