您好，我叫梁潇，四川绵阳人，目前是重庆邮电大学即将研三的学生。我的研究方向是数据挖掘。研究生在校期间做过一个项目，网络资源探测，在里面担任挖掘组组长，并发表了一篇论文。我在校期间获得过国家励志奖学金、郭长波奖学金，研究生一等奖学金。我比较热衷将理论用于实践，因此我也参加了多个关于数据挖掘即我研究方向相关的比赛，并取得了不错的成绩。我现在正在海康威视研究院实习，是算法工程师-数据挖掘方向。

我对我的研究领域很感兴趣，希望自己能够更加脚踏实地，不断进步。

商场中精确定位用户所在店铺

目标为在商场内精确的定位用户当前所在商铺。用户去逛商场的时候，手机会搜到wifi热点，通过交易时wifi的列表，用户交易时经纬度，交易时间，店铺经纬度，店铺类型，店铺价格和商场名称等信息，预测该用户再次发生交易时，位于哪家店铺。训练集给出了2017年8月一整月的数据，预测2017年9月前两周的商铺定位情况。

1. 数据预处理

1. 去除唯一属性，唯一属性通常是一些id属性，这些属性并不能刻画样本自身的分布规律，所以简单地删除这些属性即可。

2. 处理缺失值

缺失值处理的三种方法：直接使用含有缺失值的特征；删除含有缺失值的特征（该方法在包含缺失值的属性含有大量缺失值而仅仅包含极少量有效值时是有效的）；缺失值补全。如有wifi强度为null的数据，直接删除掉，以免造成干扰。均值插补:如果的距离是不可度量的，则使用该属性有效值的众数来插补缺失的值,如一些离散特征。(常见的缺失值补全方法：均值插补、同类均值插补、建模预测)

3.特征编码：能够处理非数值属性

4.数据标准化

某些算法要求样本具有零均值和单位方差；

需要消除样本不同属性具有不同量级时的影响：①数量级的差异将导致量级较大的属性占据主导地位；②数量级的差异将导致迭代收敛速度减慢；③依赖于样本距离的算法对于数据的数量级非常敏感。

min-max标准化（归一化）：对于每个属性，设minA和maxA分别为属性A的最小值和最大值，将A的一个原始值x通过min-max标准化映射成在区间[0,1]中的值x'，其公式为：新数据=（原数据 - 最小值）/（最大值 - 最小值）

z-score标准化（规范化）：基于原始数据的均值（mean）和标准差（standarddeviation）进行数据的标准化。将A的原始值x使用z-score标准化到x'。z-score标准化方法适用于属性A的最大值和最小值未知的情况，或有超出取值范围的离群数据的情况。新数据=（原数据- 均值）/ 标准差

1. 构建候选集

一是构造候选店铺集合，然后在候选集中做二分类预测。而构造候选如果没做好，后面预测就没有意义，所以构造候选使用了覆盖率作为指标。（注：向测试集中推荐候选集，二分类就是单独对它建模啊 多分类是多目标建模）

1划分数据



2 构造候选集

用word2vec、tf-idf等算法构造候选集

TF-IDF选取前3样本 TF−IDF=TF(词频)∗IDF(逆文档频率)

把在此shop出现的wifi都连起来(一句话)，tf就是wifi（单词）在序列中出现的频率，idf就是log(总行数) / 此wifi在每一行中出现了的次数。每条样本所有wifi的tf-idf的值加起来做为此样本的tf-idf值，找出最相似的。

将单词(wifi序号)的强度从大到小排序，每个将单词序列看作一句话，生成词向量，然后将所有 Vector 相加并求平均，这样就可得到 Sentence Vector 了，然后再计算其夹角余弦值即可。

(3)特征工程

标记特征: wifi与候选shop出现过的wifi重合的个数

"总量-比例"特征：该mall的总历史记录数，该user的总历史记录数，wifi历史上出现过的总次数，在当前排序位置(如最强、第二强、第三强...)上wifi历史上出现过的总次数

差值特征:wifi强度 - 候选shop的历史记录中该wifi的平均强度，wifi强度 - 候选shop的历史记录中该wifi的最小强度，wifi强度 - 候选shop的历史记录中该wifi的最大强度

三个wifi强度差值特征，按照信号强度由强到弱排列，可生成10个特征。

距离特征:与候选shop位置的GPS距离，与候选shop历史记录中心位置的GPS距离

其他特征:特征中还包括多分类的输出概率。

(3)特征选择

在比赛中，我们的基本思路是，不同模型使用不同的特征。这是由于在比赛过程中，我们发现在某个模型上十分有效的特征在另外一个模型上并不一定能够得到很好的结果。

此外，我们在比赛中使用了“前向特征选择”的方法。特征子集X从空集开始，每次选择一个特征x加入特征子集X，使得特征函数J( X)最优。简单说就是，每次都选择一个使得评价函数的取值达到最优的特征加入，其实就是一种简单的贪心算法。

糖尿病递归特征消除 (wrapper：Recursive Feature Elimination)

递归消除特征法使用一个基模型来进行多轮训练，每轮训练后，移除若干权值系数的特征，再基于新的特征集进行下一轮训练。

sklearn官方解释：对特征含有权重的预测模型(例如，线性模型对应参数coefficients)，RFE通过递归减少考察的特征集规模来选择特征。首先，预测模型在原始特征上训练，每个特征指定一个权重。之后，那些拥有最小绝对值权重的特征被踢出特征集。如此往复递归，直至剩余的特征数量达到所需的特征数量。

（4）模型融合

Blending

把原始的训练集先分成两部分，比如70%的数据作为新的训练集，剩下30%的数据作为测试集。第一层我们在这70%的数据上训练多个模型，然后去预测那30%数据的label。在第二层里，我们就直接用这30%数据在第一层预测的结果做为新特征继续训练即可。

Stacking

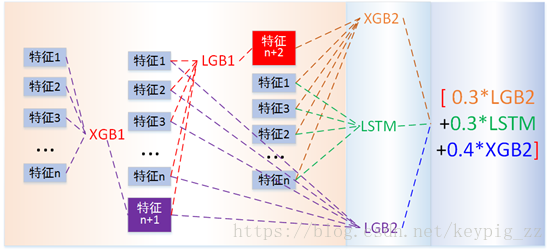
类似blending。首先，直接用所有的训练数据对第一层多个模型进行k折交叉验证，这样每个模型在训练集上都有一个预测值，然后将这些预测值做为新特征对第二层的模型进行训练。相比blending，stacking两层模型都使用了全部的训练数据。

Blending与stacking相比优点在于：比stacking简单（因为不用进行k次的交叉验证来获得新特征,而缺点在于：数据量比较少所以可能会过拟合。

Stacking

stacking 就是当用初始训练数据学习出若干个基学习器后，将这几个学习器的预测结果作为新的训练集，来学习一个新的学习器。

在比赛中我们借鉴了Stacking的思想，融合了LightGBM、XGBoost以及LSTM三个模型。其中前两类可以看作是树模型，LSTM为神经网络模型。这两类模型原理相差较大，产生的结果相关性较低，融合有利于提高预测准确性。



我们使用Xgboost\_1对特征组合F1进行学习，得到Xgboost\_1的预测结果（包括对于训练集和测试集的预测结果），该结果会作为新特征，加入特征组合F2，F3中，分别作为第二层LightGBM\_1 和 LightGBM\_2的输入特征，LightGBM\_1的结果再次作为新特征，加入特征组合F4中，作为第三层Xgboost\_2的输入特征，同时第三层包含一个LSTM模型，该模型使用特征组合F5训练，第二层LightGBM\_2的结果则与第三层Xgboost\_2,LSTM的预测结果进行加权融合作为最终结果。

过拟合与欠拟合

欠拟合：根本原因是特征维度过少，模型过于简单，但是数据量很大，所需模型用不完大量数据提供的信息，误差较大； 解决方法：（1）增加特征维度；

　 过拟合：根本原因是特征维度过多，模型假设过于复杂，参数过多，训练数据过少，噪声过多，导致拟合的函数完美的预测训练集，但对新数据的测试集预测结果差。过度的拟合了训练数据，而没有考虑到泛化能力。解决方法：（1）减少特征维度；（2）正则化，降低参数值 （3）对于神经网络，参数膨胀原因可能是因为随着网路深度的增加，同时参数也不断增加。那么可以采取减少神经网络规模（深度）的方法。也可以用一种叫dropout的方法，思想是当一组参数经过某一层神经元的时候，去掉这一层上的一部分神经元，让参数只经过一部分神经元进行计算。注意这里的去掉并不是真正意义上的去除，只是让参数不经过一部分神经元计算而已。

在本问题中，一个严重的问题是过拟合，我们发现模型在训练样本中表现优越，但是在验证数据集以及测试数据集中表现不佳，在采取了减少树深度等有效方法后，我们发现和验证以下创新的方法可以进一步减少过拟合。

（1）在Light GBM模型中，我们对于某些特征执行了一次复制，即这些特征会在训练中出现两次，结果显示训练集误差几乎一致，但是线上线下验证集误差都更小，也就是说，使用重复特征，减少了过拟合的程度，使得模型的泛化，预测效果更佳。有可能与单个树模型在抽样特征的时候，重复特征会有更大的机率被抽到有关。(和数据集相关)

（2）使用了交叉验证的方法，比如对Xgboost模型，5折交叉验证，得到5个不同的“Xgboost模型”，用这4个模型分别对测试集做一次预测，最后Xgboost的预测结果是5次预测结果的平均值。

（3）每个单模型所使用的特征都不同,每个单模型（包括LSTM)，我们有90%的特征是一致的，还有大约10%特征是每个单模型独有的，通过这种方式增加模型间的差异性，摆脱对于单一特征组合的依赖，增加了模型的泛化能力，可以获得更好的融合效果。

长短期记忆网络（Long Short-Term Memory, LSTM）

在LSTM中，Cell是基本的单元。在一个Cell中，包含三个称为“门”的结构。使用了一个包含200个单元的LSTM单层网络

Bagging和Boosting：

Bagging从原始样本集中抽取训练集。每轮从原始样本集中使用Bootstraping的方法抽取n个训练样本（在训练集中，有些样本可能被多次抽取到，而有些样本可能一次都没有被抽中）。共进行k轮抽取，得到k个训练集。（k个训练集之间是相互独立的），每次使用一个训练集得到一个模型，k个训练集共得到k个模型。Boosting的意思是这样，他通过迭代地训练一系列的分类器，每个分类器采用的样本分布都和上一轮的学习结果有关。

Bagging 和Boosting的区别：

样本选择上：Bagging:训练集是有放回选取的，各轮训练集之间独立；Boosting:每一轮的训练集不变

样例权重：Bagging:使用均匀取样，每个样例的权重相等，Boosting:根据错误率不断调整样例的权值，错误率越大权重越大

并行计算：Bagging:各个预测函数可以并行生成，Boosting:各个预测函数只能顺序生成。

两种方法都是把若干个分类器整合为以个分类器的方法，只是整合方式不一样，最终得到不一样效果。

偏差和方差

就机器学习算法来说，其泛化误差可以分解为两部分，偏差和方差。偏差指的是算法的期望预测与真实预测之间的偏差程度，反应了模型本身的拟合能力；方差度量了同等大小的训练集的变动导致学习性能的变化，刻画了数据扰动所导致的影响。

当模型越复杂时，拟合的程度就越高，模型的训练偏差就越小。但此时如果换一组数据可能模型的变化就会很大，即模型的方差很大。所以模型过于复杂的时候会导致过拟合。当模型越简单时，即使我们再换一组数据，最后得出的学习器和之前的学习器的差别就不那么大，模型的方差很小。还是因为模型简单，所以偏差会很大。

  对于Boosting来说，每一步我们都会在上一轮的基础上更加拟合原数据，所以可以保证偏差（bias）,所以对于每个基分类器来说，问题就在于如何选择variance更小的分类器，即更简单的分类器，所以我们选择了深度很浅的决策树。

对于Bagging算法来说，由于我们会并行地训练很多不同的分类器的目的就是降低这个方差(variance) ,所以对于每个基分类器来说，目标就是如何降低这个偏差（bias),所以我们会采用深度很深甚至不剪枝的决策树。

随机森林和极端随机森林：

随机森林的优点：(1)随机森林能处理很高维数据（也就是很多特征的数据）(2)训练速度快，容易做成并行化方法(训练时，树与树之间是相互独立的) (3)随机森林抗过拟合能力比较强(增加树)

随机森林的缺点：(1)对于许多统计建模者来说，随机森林给人的感觉就像一个黑盒，你无法控制模型内部的运行。只能在不同的参数和随机种子之间进行尝试。(2)可能有很多相似的决策树，掩盖了真实的结果。(3)对于小数据或者低维数据（特征较少的数据），可能不能产生很好的分类。

随机森林优点:训练速度快，容易并行化计算。

Extremely randomized trees，极端随机森林

(1)randomForest应用的是Bagging模型, extraTree使用的所有的样本，只是特征是随机选取的。

(2)ExtraTrees算法多一层随机性，在对连续变量特征选取最优分裂值时，不会计算所有分裂值的效果，来选择分裂特征。而是对每一个特征，在它的特征取值范围内，随机生成一个split value，再计算看选取哪一个特征来进行分裂。方法比随机森林的随机性更强。对于某棵决策树，由于它的最佳分叉属性是随机选择的，因此用它的预测结果往往是不准确的，但多棵决策树组合在一起，就可以达到很好的预测效果。

GBDT：

GBDT中的树都是回归树， GBDT的核心就在于，每一棵树学的是之前所有树结论和的残差(负梯度)，这个残差就是一个加预测值后能得真实值的累加量。比如A的真实年龄是18岁，但第一棵树的预测年龄是12岁，那么在第二棵树里我们把A的年龄设为6岁去学习，如果第二棵树真的能把A分到6岁的叶子节点，那累加两棵树的结论就是A的真实年龄。每一步的残差计算其实变相地增大了分错instance的权重，而已经分对的instance则都趋向于0。这样后面的树就能越来越专注那些前面被分错的instance。

Adaboost是另一种boost方法，它按分类对错，分配不同的weight，计算cost function时使用这些weight，从而让“错分的样本权重越来越大，使它们更被重视”。

Gradient Boost优点：

(1)Decision Tree 可以很好的处理 missing feature，这是他的天然特性，因为决策树的每个节点只依赖一个 feature，如果某个 feature 不存在，这颗树依然可以拿来做决策，只是少一些路径。

Decision Tree 可以很好的处理各种类型的 feature，也是天然特性，很好理解。

对特征空间的 outlier 有鲁棒性，因为每个节点都是 x < 𝑇 的形式，至于大多少，小多少没有区别，outlier 不会有什么大的影响，同样逻辑回归和 SVM 没有这样的天然特性。

数据规模影响不大，因为我们对弱分类器的要求不高，作为弱分类器的决策树的深度一般设的比较小，即使是大数据量，也可以方便处理。像 SVM 这种数据规模大的时候训练会比较麻烦。

Gradient Boost缺点：

Boost是一个串行过程，不好并行化。还有就是xgb的优点。

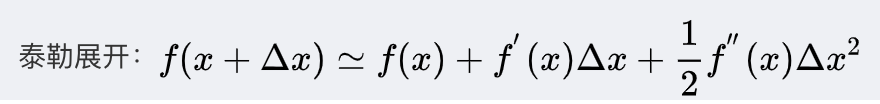
XGBoost

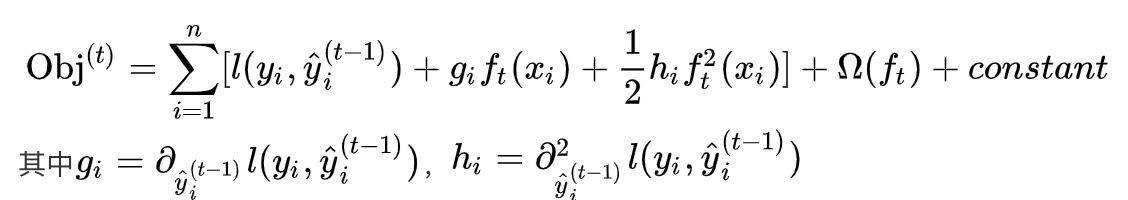
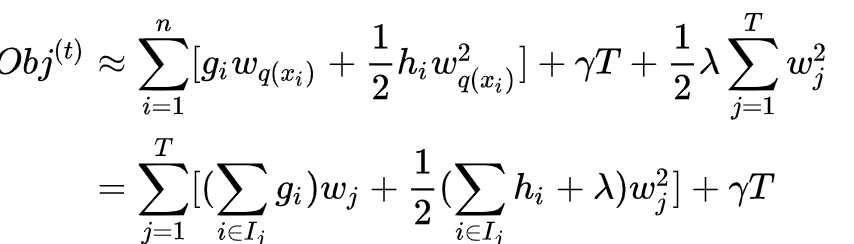
GBDT是一种基于集成思想下的Boosting学习器，并采用梯度提升的方法进行每一轮的迭代最终组建出强学习器，这样的话算法的运行往往要生成一定数量的树才能达到令我们满意的准确率。当数据集大且较为复杂时，运行一次极有可能需要几千次的迭代运算，这将对我们使用算法造成巨大的计算瓶颈。xgBoost最大的特点在于它能够自动利用CPU的多线程进行并行计算，同时在算法上加以改进提高了精度。

目标函数的形式为:损失函数+正则项

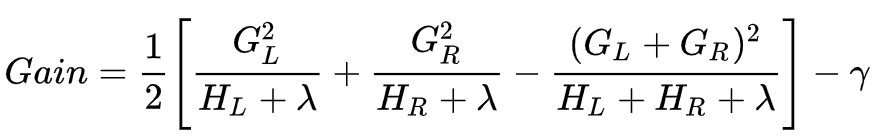
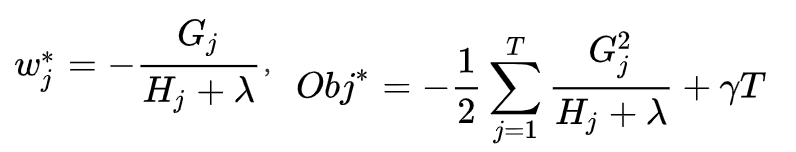
正则项，之所以要引入它是因为我们的目标是希望生成的模型能准确的预测新的样本（即应用于测试数据集），而不是简单的拟合训练集的结果（这样会导致过拟合）。所以需要在保证模型“简单”的基础上最小化训练误差，这样得到的参数才具有好的泛化性能。而正则项就是用于惩罚复杂模型，避免预测模型过分拟合训练数据。L1通过减少参数的数量来降低复杂度，L2通过减小参数值的大小来降低复杂度。

观察发现，如果目标函数中的损失函数权重过高，那么模型的预测精度则不尽人意，反之如果正则项的权重过高，所生成的模型则会出现过拟合情况，难以对新的数据集做出有效预测。只有平衡好两者之间的关系，控制好模型复杂度，并在此基础上对参数进行求解，生成的模型才会“简单有效”（这也是机器学习中的偏差方差均衡）



由此，我们将目标函数转换为一个一元二次方程求最小值的问题（在此式中，变量为w，函数本质上是关于的二次函数），略去求解步骤，最终结果如下所示：



xgb优点

传统GBDT在优化时只用到一阶导数信息，xgboost则对代价函数进行了二阶泰勒展开，同时用到了一阶和二阶导数。 xgboost工具支持自定义代价函数，只要函数可一阶和二阶求导。

xgboost在代价函数里加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数。正则项降低了模型的variance，使学习出来的模型更加简单，防止过拟合，这也是xgboost优于传统GBDT的一个特性。

列抽样（columnsubsampling）。xgboost借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算，这也是xgboost异于传统gbdt的一个特性。

缺失值的处理。对于特征的值有缺失的样本，xgboost可以自动学习出它的分裂方向。

支持并行。注意xgboost的并行不是tree粒度的并行，xgboost也是一次迭代完才能进行下一次迭代的。xgboost的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），xgboost在训练之前，预先对数据进行了排序，然后保存为block结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个block结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。(特征粒度上的并行，block结构，预排序)

xgb缺点：

(1)xgBoosting采用预排序，在迭代之前，对结点的特征做预排序，遍历选择最优分割点，数据量大时，贪心法耗时，LightGBM方法采用histogram算法，占用的内存低，数据分割的复杂度更低；

(2)xgBoosting采用level-wise生成决策树，见下

LightGBM

Level-wise一次可以同时分裂同一层的叶子: (1)容易进行多线程优化，(2)好控制模型复杂度，(3) 不容易过拟合。但实际上Level-wise是一种低效的算法，因为它不加区分的对待同一层的叶子，带来了很多没必要的开销，因为实际上很多叶子的分裂增益较低，没必要进行搜索和分裂。 Leaf-wise则是一种更为高效的策略，每次从当前所有叶子中，找到分裂增益最大的一个叶子，然后分裂，如此循环。因此同Level-wise相比，在分裂次数相同的情况下，Leaf-wise可以降低更多的误差，得到更好的精度。Leaf-wise的缺点是可能会长出比较深的决策树，产生过拟合。因此LightGBM在Leaf-wise之上增加了一个最大深度的限制，在保证高效率的同时防止过拟合。

直方图算法的基本思想是先把连续的浮点特征值离散化成k个整数，同时构造一个宽度为k的直方图。在遍历数据的时候，根据离散化后的值作为索引在直方图中累积统计量，当遍历一次数据后，直方图累积了需要的统计量，然后根据直方图的离散值，遍历寻找最优的分割点。我的理解: 就是连续特征离散化. 直方图算法不仅不需要额外存储预排序的结果，而且可以只保存特征离散化后的值，而这个值一般用8位整型存储就足够了，内存消耗可以降低为原来的1/8(有疑问)。由于特征被离散化后，找到的并不是很精确的分割点，所以会对结果产生影响。但是决策树本来就是弱模型，分割点是不是精确并不是太重要；较粗的分割点（粒度）也有正则化的效果。一个叶子的直方图可以由它的父亲节点的直方图与它兄弟的直方图做差得到。通常构造直方图，需要遍历该叶子上的所有数据，但直方图做差仅需遍历直方图的k个桶。利用这个方法，LightGBM可以在构造一个叶子的直方图后，可以用非常微小的代价得到它兄弟叶子的直方图，在速度上可以提升一倍。

直方图做差优化可以达到两倍的加速，可以观察到一个叶子节点上的直方图，可以由它的父亲节点直方图减去它兄弟节点的直方图来得到。根据这一点我们可以构造出来数据量比较小的叶子节点上的直方图，然后用直方图做差来得到数据量比较大的叶子节点上的直方图，从而达到加速的效果。

lightgbm缺点：

(1)Leaf-wise的缺点：可能会长出比较深的决策树，产生过拟合。因此LightGBM在Leaf-wise之上增加了一个最大深度限制，在保证高效率的同时防止过拟合。

(2) 牺牲精度换取时间，与其说是、不足，倒不如说是lightGBM作者们构建新算法时着重瞄准的点。

项目内容1: Scrapy框架

爬取网站数据，提取结构性数据而编写的应用框架

爬虫是主要干活的, 用于从特定的网页中提取自己需要的信息, 即所谓的实体(Item)；用户也可以从中提取出链接,让Scrapy继续抓取下一个页面。

引擎从调度器中取出一个链接(URL)用于接下来的抓取

引擎把URL封装成一个请求(Request)传给下载器

下载器把资源下载下来，并封装成应答包(Response)

爬虫解析Response

解析出实体（Item）,则交给实体管道进行进一步的处理

解析出的是链接（URL）,则把URL交给调度器等待抓取

# Response返回的为网页html码为字符串，通过这种形式让html码结构化，方便调用某个模块(lxml为解析html代码的库)。

soup = bs4.BeautifulSoup(response.body.decode(response.encoding), 'lxml')

# 拿到table,对table进行操作

Tag = soup.table

通过正则表达式得到信息，

Info = soup.find('span', class\_='ctt') （结构）

UID = re.search("/u/[0-9]{10}", str(Tag.a)) # 这个关注人的ID

项目内容2: 分为两个步骤，检测和匹配。

检测是将其中的图片传入，截取人脸。匹配图片时，欧式距离越小，两张图片相似度越大（有阈值）。

什么是深度: 多个隐层，不需要过多地提取特征，在神经网络的每一层中，计算机都可以自动学习出特征

什么是TensorFlow:深度学习开源工具，可以使用它来搭建自己的神经网络。框架的主要目的是提供一个工具箱，是开发时能够简化代码，呈现出的模型尽可能简洁易懂。

设计理念: (1) 将图的定义和图的运行完全分开(符号式编程)（2） 涉及的运算都要放在图中，而图的运行只发生在会话中，会话提供了操作运行和tensor求值的环境，在调用Session对象的run方法来执行图时，传入一些tensor,这个过程叫填充。

编程模型:用数据流图做计算。输入、塑形、隐层、输出层、Softmax、，交叉熵、计算梯度、SGD训练，从上往下计算每一层的参数。

TensorFlow的数据流图是由节点和边组成的有向无环图。 TensorFlow由Tensor和Flow两部分组成，Tensor代表边，flow代表节点所做的操作。

TensorFlow的Session对象是支持多线程的，可以在同一个会话（Session）中创建多个线程，并行执行。在Session中的所有线程都必须能被同步终止，异常必须能被正确捕获并报告，会话终止的时候， 队列必须能被正确地关闭。TensorFlow提供了两个类来实现对Session中多线程的管理：tf.Coordinator和 tf.QueueRunner，这两个类往往一起使用。

Coordinator类用来管理在Session中的多个线程，可以用来同时停止多个工作线程并且向那个在等待所有工作线程终止的程序报告异常，该线程捕获到这个异常之后就会终止所有线程。

QueueRunner类用来启动tensor的入队线程，可以用来启动多个工作线程同时将多个tensor（训练数据）推送入文件名称队列中，具体执行函数是 tf.train.start\_queue\_runners ， 只有调用 tf.train.start\_queue\_runners 之后，才会真正把tensor推入内存序列中，供计算单元调用，否则会由于内存序列为空，数据流图会处于一直等待状态。

MTCNN:多任务卷积神经网络，可以同时完成人脸检测和人脸对齐两项任务。总体分为PNet、 Rnet和Onet三层网络结构。

构建图像金字塔:将原始图像缩放到不同尺度，形成图像金字塔。接着会对每个尺度的图片通过神经网络计算一遍。原因:原始图片中的人脸存在不同的尺度，比如有的人脸大，可以在缩小后的图片上检测，这样可以在统一尺度下检测人脸。

P(proposal network)： p-net的输入是一个12\*12\*3的图像，该网络判断这个12\*12的图像中是否含有人脸，并且给出人脸框和关键点的位置。输出由三部分:1）该图像是否是人脸（两个概率） 2）框的精确位置 3）5个关键点。基本构造是一个cnn。因为是将图片金字塔的各个尺度都使用p-net计算了一遍，因此形成了大小不同的人脸框。

R(refine): r-net的输入是一个24\*24的大小，图像基本构造是一个cnn，比p多一个全连接层，过滤大量效果比较差的候选框。

O(output): 基本构造是一个卷积神经网络,比R多一个卷积层，输出多个人脸面部特征点。

从P-Net到R-Net，最后再到O-Net，网络输入的图片越来越大，卷积层的通道数越来越多，内部的层数也越来越多， 因此它们识别人脸的准确率应该是越来越高的。同时， P-Net的运行速度是最快的， R-Net的速度其次，

O-Net的运行速度最慢。之所以要使用三个网络，是因为如果一开始直接对

图中的每个区域使用O-Net， 速度会非常慢。实际上P-Net先做了一遍过滤，将过滤后的结果再交给R-Net进行过滤，最后将过滤后的结果交给效果最好但速度较慢的O-Net进行判别。这样在每一步都提前减少了需要判别的数量，有效降低了处理时间。

Face-Net: 人脸识别方法。通过CNN将人脸映射到欧氏空间的特征向量上，计算不同图片人脸特征的距离。直接通过学习将特征变为欧式平面上的一个点，比较点之间的距离来进行判断。

1.前面部分采用一个CNN结构提取特征，

2.CNN之后接一个特征归一化（使其特征的||f(x)||2=1,这样子，所有图像的特征都会被映射到一个超球面上)，

3.再接入一个embedding层(嵌入函数)，嵌入过程可以表达为一个函数，即把图像x通过函数f映射到d维欧式空间。

4.此外，作者对嵌入函数f(x)的值，即值阈，做了限制。使得x的映射f(x)在一个超球面上。

5.接着，再去优化这些特征，而文章这里提出了一个新的损失函数，triplet损失函数(优化函数），而这也是文章最大的特点所在

什么是Triplet Loss呢？故名思意，也就是有三张图片输入的Loss（之前的都是Double Loss或者是SingleLoss）。

本文通过LDA思想训练分类模型，使得类内特征间隔小，类间特征间隔大。

  在上面中，如果严格的按照上式来进行学习的话，它的T(穷举所有的图像3元组)是非常大的。

    举个例子：在一个1000个人，每人有20张图片的情况下，其T=(1000\*20)\*19\*(20\*999)(总图片数\*每个图片类内组合\*每个图片类间组合),也就是O(T)=N^2 ，所以，穷举是不大现实的。那么，我们只能从这所有的N^2个中选择部分来进行训练。现在问题来了，怎么从这么多的图像中挑选呢？答案是选择最难区分的图像对。

    给定一张人脸图片，我们要挑选：

    1.一张hard positive：即在类内的另外19张图像中，跟它最不相似的图片。(正样本里面最差的样本)

    2.一张hard negative：即在类间的另外20\*999图像中，跟它最为相似的图片。(负样本里面最差的样本)

挑选hard positive 和hard negative有两种方法，offline和online方法，具体的差别只是在训练上。

Tensorflow：

请简要介绍下Tensorflow的计算图

什么是计算图？它实质上是一个全局数据结构：计算图是一个有向图，捕获有关计算方法的指令。对定义计算和计算的执行做了分离。

tensorflow的编程和以往接触的编程方式有很大差异。以前的编程，无论是编译类型语言还是动态解释型语言，变量计算后，就会得到结果，比如c=a+b，当执行完语句后，就会得到c的值。但， tensorflow不是！ 首先看一下，tensor是什么？它是一个 n 维数组： 0-d tensor: scalar （标量） 1-d tensor: vector （向量） 2-d tensor: matrix（矩阵）…

有了tensor, 那么tensorflow的计算流图就可以构建为下面这个样子，圆形节点代表tensor间执行的操作：Session对象封装了tensorflow的执行环境。记住：计算图只是定义了operations，operations只有在session里面执行才能真正得到计算图的结果。

会话

会话的作用是处理内存分配和优化，使我们能够实际执行由图形指定的计算。可以将计算图想象为我们想要执行的计算的“模板”：它列出了所有的步骤。为了使用这个图表，我们还需要发起一个会话，它使我们能够实际地完成任务。例如，遍历模板的所有节点来分配一组用于存储计算输出的存储器。为了使用 Tensorflow 进行各种计算，我们既需要图也需要会话。

注: 根据图的结构，我们不需要计算所有的节点也可以评估我们想要的节点！

变量:

要创建变量，请使用 tf.get\_variable ()。tf.get\_variable () 的前两个参数是必需的，其余是可选的。它们是 tf.get\_variable (name,shape）。name 是一个唯一标识这个变量对象的字符串。它在全局图中必须是唯一的，所以要确保不会出现重复的名称。shape 是一个与张量形状相对应的整数数组，它的语法很直观——每个维度对应一个整数，并按照排列。例如，一个 3×8 的矩阵可能具有形状 [3,8]。要创建标量，请使用空列表作为形状：[]。有两种主要方法可以用于给变量赋值：初始化器和 tf.assign ()。tf.global\_variables\_initializer () 将在其创建时查看全局图，自动将依赖关系添加到图中的每个 tf.initializer 上。

为什么初始化器不起作用

当我们调用 sess.run (init) 时，它会告诉每个初始化器完成它们的任务，初始化变量，这样在调用 sess.run (count\_variable) 时就不会出错。

Keras

在Keras中有两类主要的模型：Sequential顺序模型和使用函数式API的Model 类模型。

Keras的底层库使用Theano或TensorFlow，这两个库也称为Keras的后端。无论是Theano还是TensorFlow，都是一个“符号式”的库。

因此，这也使得Keras的编程与传统的Python代码有所差别。笼统的说，符号主义的计算首先定义各种变量，然后建立一个“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译以确定其内部细节，然而，此时的计算图还是一个“空壳子”，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值。

就像用管道搭建供水系统，当你在拼水管的时候，里面是没有水的。只有所有的管子都接完了，才能送水。

Keras的模型搭建形式就是这种方法，在你搭建Keras模型完毕后，你的模型就是一个空壳子，只有实际生成可调用的函数后（K.function），输入数据，才会形成真正的数据流。

深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式。

第一种，遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。这种方法每更新一次参数都要把数据集里的所有样本都看一遍，计算量开销大，计算速度慢，不支持在线学习，这称为Batch gradient descent，批梯度下降。

另一种，每看一个数据就算一下损失函数，然后求梯度更新参数，这个称为随机梯度下降，stochastic gradient descent。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近晃来晃去，hit不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。

为了克服两种方法的缺点，现在一般采用的是一种折中手段，mini-batch gradient decent，小批的梯度下降，这种方法把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大。

中国可视化与可视分析大会 (ChinaVis) 中国图象图形学学会

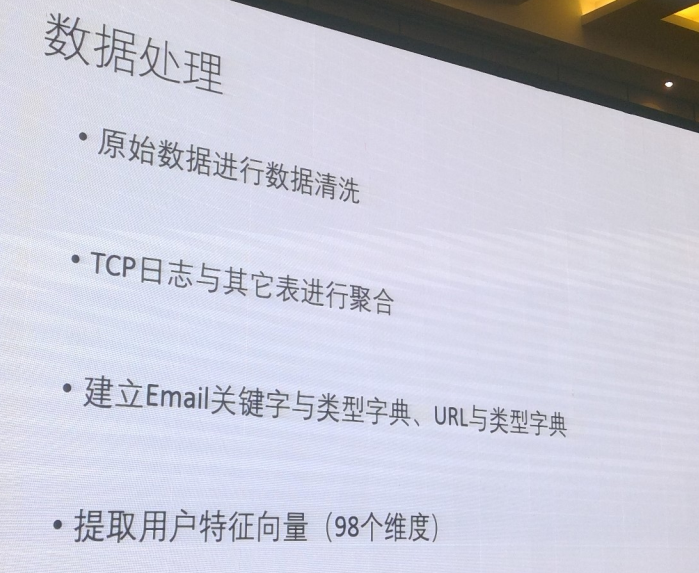
背景:一家互联网高科技公司正在研发一款重量级新产品，公司对内部发生的一切异常现象都非常敏感，将根据公司内部采集到的数据，分析并处置可能存在的各种安全威胁。

（1）分析公司内部员工所属部门及各部门的人员组织结构，给出公司员工的组织结构图

（2）分析该公司员工的日常工作行为，按部门总结和展示员工的正常工作模式

（3）找出至少5个异常事件，并分析这些事件之间可能存在的关联，总结你认为有价值的威胁情报，并简要说明你是如何利用可视分析方法找到这些威胁情报的

数据: 本次比赛提供了该公司内部2017年11月共30天的多种监控数据，包括登录日志、网页访问日志、邮件日志、打卡日志和TCP流量日志。





**Python方面**

**多态**:

新增一个子类，任何依赖父类作为参数的函数或者方法都可以不加修改地正常运行，就会自动调用实际类型的run()方法。对扩展开放，对修改封闭。

对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法。对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了。

**静态类型语言**是指在编译时变量的数据类型即可确定的语言(java)

**动态类型语言**是在运行时确定数据类型的语言。(python) 鸭子特性

**生成器**：通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，我们就可以在循环的过程中不断推算出后续的元素。这样就不必创建完整的list，从而节省大量的空间。在Python中，这种**一边循环一边计算的机制**，称为生成器：generator。

如果一个函数定义中包含**yield**关键字，那么这个函数就不再是一个普通函数，而是一个generator。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

**迭代器**: 可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。不过可以通过iter()函数获得一个Iterator对象。

垃圾回收

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

现在的高级语言如java，c#等，都采用了垃圾收集机制，而不再是c，c++里用户自己管理维护内存的方式。自己管理内存极其自由，可以任意申请内存，但如同一把双刃剑，为大量内存泄露，悬空指针等bug埋下隐患。

对于一个字符串、列表、类甚至数值都是对象，且定位简单易用的语言，自然不会让用户去处理如何分配回收内存的问题。

python里也同java一样采用了垃圾收集机制，不过不一样的是:

最关键的一句话：

python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

PyObject是每个对象必有的内容，其中ob\_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob\_refcnt就会增加，当引用它的对象被删除，它的ob\_refcnt就会减少.引用计数为0时，该对象生命就结束了。优点: 简单 实时性 缺点: 循环引用

导致引用计数+1的情况

对象被创建，例如a=23

对象被引用，例如b=a

对象被作为参数，传入到一个函数中，例如func(a)

对象作为一个元素，存储在容器中，例如list1=[a,a]

导致引用计数-1的情况

对象的别名被显式销毁，例如del a

对象的别名被赋予新的对象，例如a=24

一个对象离开它的作用域，例如f函数执行完毕时，func函数中的局部变量（全局变量不会）

对象所在的容器被销毁，或从容器中删除对象

什么是内存泄漏呢？

指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。导致程序运行速度减慢甚至系统崩溃等严重后果。

有 \_\_del\_\_() 函数的对象间的循环引用是导致内存泄漏的主凶。不使用一个对象时使用:del object 来删除一个对象的引用计数就可以有效防止内存泄漏问题。通过 Python 扩展模块 gc 来查看不能回收的对象的详细信息。可以通过 sys.getrefcount(obj) 来获取对象的引用计数，并根据返回值是否为 0 来判断是否内存

泄漏。

例:

list1 = []

list2 = []

list1.append(list2)

list2.append(list1)

list1与list2相互引用，如果不存在其他对象对它们的引用，list1与list2的引用计数也仍然为1，所占用的内存永远无法被回收，这将是致命的。

5、针对“循环引用”的解决办法

（1）标记清除技术——mark and sweep

（2）分代回收技术——generation collection

（3）手动使用gc模块

  『标记清除（Mark—Sweep）』算法是一种基于追踪回收（tracing GC）技术实现的垃圾回收算法。它分为两个阶段：第一阶段是标记阶段，GC会把所有的『活动对象』打上标记，第二阶段是把那些没有标记的对象『非活动对象』进行回收。那么GC又是如何判断哪些是活动对象哪些是非活动对象的呢？

  对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象（root object）出发，沿着有向边遍历对象，可达的（reachable）对象标记为活动对象，不可达的对象就是要被清除的非活动对象。根对象就是全局变量、调用栈、寄存器。

不过，这种简单粗暴的标记清除算法也有明显的缺点：清除非活动的对象前它必须顺序扫描整个堆内存，哪怕只剩下小部分活动对象也要扫描所有对象。

三、分代技术——generation collection

分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。

Python默认定义了三代对象集合，索引数越大，对象存活时间越长。

1. 分代技术是一种典型的以空间换时间的技术，这也正是java里的关键技术。这种思想简单点说就是：对象存在时间越长，越可能不是垃圾，应该越少去收集。
2. 这样的思想，可以减少标记-清除机制所带来的额外操作。分代就是将回收对象分成数个代，每个代就是一个链表（集合），代进行标记-清除的时间与代内对象
3. 存活时间成正比例关系。
4. 从上面代码可以看出python里一共有三代，每个代的threshold值表示该代最多容纳对象的个数。默认情况下，当0代超过700, 或1，2代超过10，垃圾回收机制将触发。
5. 0代触发将清理所有三代，1代触发会清理1,2代，2代触发后只会清理自己。

总结：

分代回收是一种以空间换时间的操作方式，Python将内存根据对象的存活时间划分为不同的集合，每个集合称为一个代，Python将内存分为了3“代”，分别为年轻代（第0代）、中年代（第1代）、老年代（第2代），他们对应的是3个链表，它们的垃圾收集频率与对象的存活时间的增大而减小。新创建的对象都会分配在年轻代，年轻代链表的总数达到上限时，Python垃圾收集机制就会被触发，把那些可以被回收的对象回收掉，而那些不会回收的对象就会被移到中年代去，依此类推，老年代中的对象是存活时间最久的对象，甚至是存活于整个系统的生命周期内。同时，分代回收是建立在标记清除技术基础之上。分代回收同样作为Python的辅助垃圾收集技术处理那些容器对象.

\*\*PYTHON是如何进行内存管理的？

1. Python是如何进行内存管理的？

答:Python的内存管理是由Python得解释器负责的，

从三个方面来说,一对象的引用计数机制,二垃圾回收机制,三内存池机制

一、对象的引用计数机制

Python内部使用引用计数，来保持追踪内存中的对象，所有对象都有引用计数。

二、垃圾回收

1，当一个对象的引用计数归零时，它将被垃圾收集机制处理掉。

2，当两个对象a和b相互引用时，del语句可以减少a和b的引用计数，并销毁用于引用底层对象的名称。然而由于每个对象都包含一个对其他对象的应用，因此引用计数不会归零，对象也不会销毁。（从而导致内存泄露）。为解决这一问题，解释器会定期执行一个循环检测器，搜索不可访问对象的循环并删除它们。

三、内存池机制

Python提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。

1，Pymalloc机制。为了加速Python的执行效率，Python引入了一个内存池机制，用于管理对小块内存的申请和释放。

2，Python中所有小于256个字节的对象都使用pymalloc实现的分配器，而大的对象则使用系统的malloc。

3，对于Python对象，如整数，浮点数和List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

TCP

三次握手

第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，并进入SYN\_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

四次握手

（1）客户端A发送一个FIN，用来关闭客户A到服务器B的数据传送。

（2）服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。

（3）服务器B关闭与客户端A的连接，发送一个FIN给客户端A（报文段6）。

（4）客户端A发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的.

Hadoop

大数据包括了以Hadoop和Spark为代表的基础大数据框架，还包括了数据挖掘和用机器算法进行预测分析等技术。

Hadoop两大核心：

1、HDFS分布式文件系统：存储是大数据技术的基础

2、Map Reduce编程模型：分布式计算是大数据应用的解决方案

Map

在map阶段，MapReduce会对要处理的数据进行分片（split）操作，为每一个分片分配一个MapTask任务。接下来map()函数会对每一个分片中的每一行数据进行处理得到键值对（key,value），其中key为偏移量，value为一行的内容。此时得到的键值对又叫做“中间结果”。此后便进入shuffle阶段，由此可以看出shuffle阶段的作用是处理“中间结果”。

shuffle

shuffle的本意是洗牌、混洗的意思，把一组有规则的数据尽量打乱成无规则的数据。而在MapReduce中，shuffle更像是洗牌的逆过程，指的是将map端的无规则输出按指定的规则“打乱”成具有一定规则的数据，以便reduce端接收处理。

因为频繁的磁盘I/O操作会严重的降低效率，因此“中间结果”不会立马写入磁盘，而是优先存储到map节点的“环形内存缓冲区”，在写入的过程中进行分区（partition），也就是对于每个键值对来说，都增加了一个partition属性值。当写入的数据量达到预先设置的阙值后便会启动溢写出线程将缓冲区中的那部分数据溢出写（spill）到磁盘的临时文件中，并在写入前根据key进行排序（sort）和合并（combine，可选操作）。当整个map任务完成溢出写后，会对磁盘中这个map任务产生的所有临时文件（spill文件）进行归并（merge）操作生成最终的正式输出文件，此时的归并是将所有spill文件中的相同partition合并到一起，并对各个partition中的数据再进行一次排序（sort），生成key和对应的value-list，至此，map端shuffle过程结束，接下来等待reduce task来拉取数据。对于reduce端的shuffle过程来说，reduce task在执行之前的工作就是不断地拉取当前job里每个map task的最终结果，然后对从不同地方拉取过来的数据不断地做merge最后合并成一个分区相同的大文件，然后对这个文件中的键值对按照key进行sort排序，排好序之后紧接着进行分组，分组完成后才将整个文件交给reduce task处理。

纠正：分区好像是发生在溢出写过程之前，也就是当满足溢出写条件时，首先进行分区，然后分区内排序，并且选择性的combine，最后写出到磁盘

HDFS由一个NameNode和多个DataNode组成。

NameNode作用：(2.0版本是有两个namenode的，一个是主节点，一个是备用的，主节点挂了，就激活备用的)

1、管理文件系统的命名空间，存放文件元数据。

2、维护着文件系统的所有文件和目录，文件与数据块的映射。

3、记录每个文件中各个块所在数据节点的信息。HDFS写流程： 客户端向NameNode发起写数据请求，分块写入DataNode节点，DataNode自动完成副本备份.DataNode向NameNode汇报存储完成，NameNode通知客户端HDFS读流程： 客户端向NameNode发起读数据请求，NameNode找出最近的DataNode节点信息，客户端从DataNode分块下载文件。

YARN是Hadoop2.0之后的资源管理器1、ResourceManager：1）分配和调度资源2）启动并监控ApplicationMaster3）监控NodeManager2、ApplicationMaster：1）为MapReduce类型的程序申请资源，并分配给内部任务2）负责数据的切分3）监控任务的执行及容错3、NodeManager：1）管理单个节点的资源2）处理来自ResourceManager的命3）处理来自ApplicationMaster的命令

MapReduce编程模型

输入一个大文件，通过Split之后，将其分为多个分片；

每个文件分片有单独的机器去处理，这就是Map方法；

将各个机器计算的结果进行汇总并得到最终的结果，这就是Reduce方法

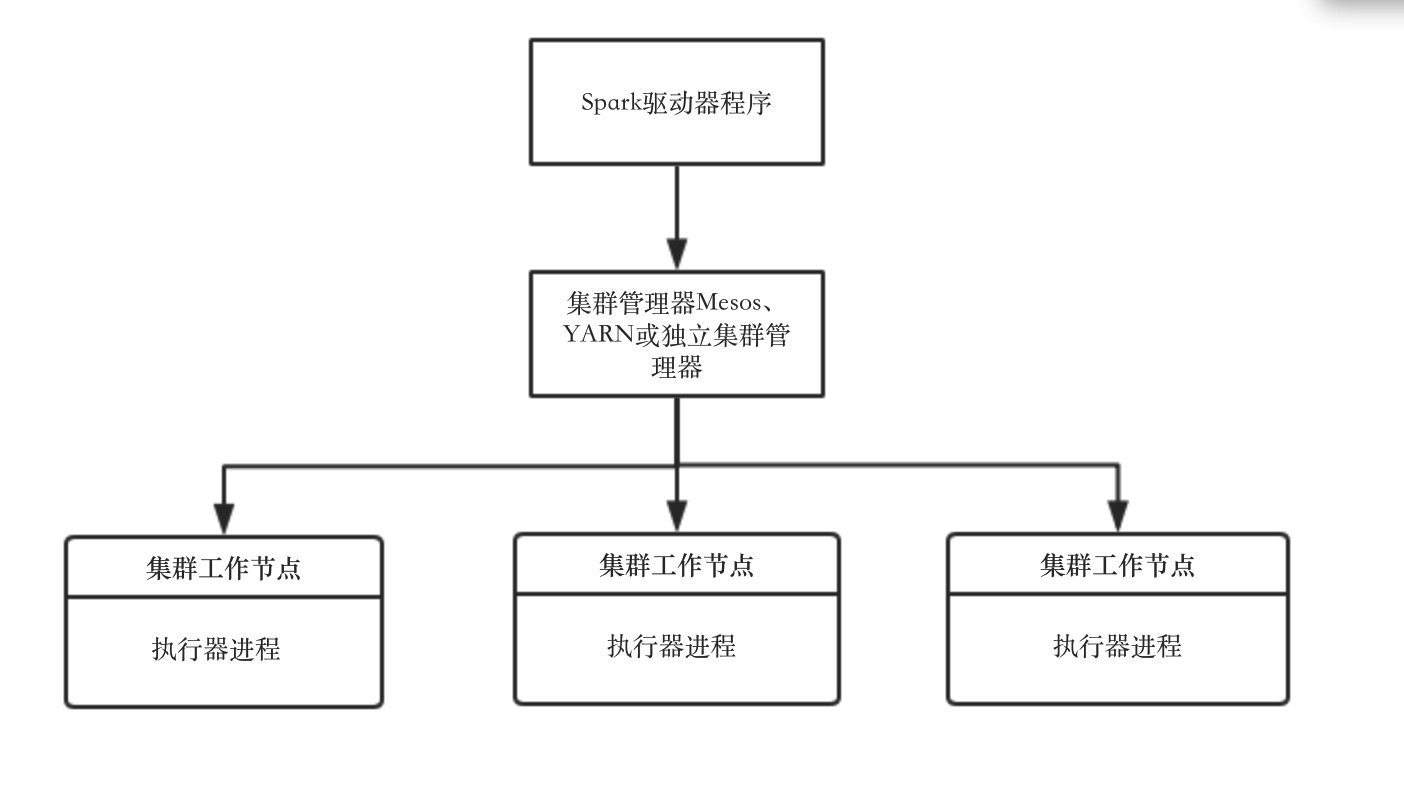
首先是 Mapreduce经过SplitInput 输入分片 决定map的个数在用Record记录 key value。

Shuffle：

   合并（merge）map输出时先输出到环形内存，当内存使用率达到60%时开始溢出写入到文件，溢出文件都是小文件，所以就要合并他们，在这个构成中就会排序，根据key值比较排序

Spark:

Spark是一个通用内存并行计算框架，简单来说Spark是内存迭代计算，每个算子将计算结果保存在内存中，其他算子，读取这个结果，继续计算。



如上图所示，在Spark集群中有一个节点负责中央协调，调度各个分布式工作节点。这个中央协调点叫“驱动器节点(Driver)”,与之对应的工作节点叫“执行器节点(executor)”。驱动器节点和所有的执行器节点被称为一个Spark应用(Application)。Spark应用通过一个“集群管理器(Cluster Manager)”的外部服务在集群中的机器上启动，其中它自带的集群管理器叫“独立集群管理器”。

Spark执行的流程：

Spark执行一个应用时，由作业、任务和步骤组成。先回顾一下：

任务：Spark的最小工作单位

步骤：由多个任务组成

作业：由一个或多个作业组成

用户定义RDD的有向无环图（DAG）

Action操作把有向无环图强制转译为执行计划：Spark调度器提交一个作业来计算所必要的RDD，这个作业包含一个或多个步骤，每个步骤就是一些并行执行的计算任务。

在集群中调度并执行任务：步骤是按顺序处理的，任务则独立启动来计算RDD的一部分。当作业的最后一个步骤结束时，一个Action操作也执行完了。

RDD：

RDD（Resilient Distributed Dataset） 弹性分布式数据集，其实就是分布式的元素集合。Python的基本内置的数据类型有整型、字符串、元祖、列表、字典，布尔类型等，而Spark的数据类型只有RDD这一种，在Spark里，对数据的所有操作，基本上就是围绕RDD来的，譬如创建、转换、求值等等。所有RDD的转换都是lazy(惰性求值)的，RDD的转换操作会生成新的RDD，新的RDD的数据依赖于原来的RDD的数据，每个RDD又包含多个分区。那么一段程序实际上就构造了一个由相互依赖的多个RDD组成的有向无环图(DAG)。并通过在RDD上执行动作将这个有向无环图作为一个Job提交给Spark执行。理解RDD后可以避免以后走很多弯路。关于RDD的特点，可以搜到很多资料，其实我们只需要理解两点就可以了：1.不可变 2.分布式

如果RDD不可变，那么在进行数据操作的时候，怎么改变它的值，怎么进行计算呢？其实RDD支持两种操作：

1.Tansformation（转化操作）：返回值还是一个RDD

2.Action（行动操作）：返回值不是一个RDD

第一种Transformation是返回一个新的RDD，如map(),filter()等。这种操作是lazy(惰性)的，即从一个RDD转换生成另一个RDD的操作不是马上执行，只是记录下来，只有等到有Action操作是才会真正启动计算，将生成的新RDD写到内存或hdfs里，不会对原有的RDD的值进行改变。而Action操作才会实际触发Spark计算，对RDD计算出一个结果，并把结果返回到内存或hdfs中，如count(),first()等。

还有一种情况，如果我们想多次使用同一个RDD，每次都对RDD进行Action操作的话，会极大的消耗Spark的内存，这种情况下，我们可以使用RDD.persist()把这个RDD缓存下来，在内存不足时，可以存储到磁盘(disk)里。

spark中的宽窄依赖：

窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，相应的，宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用。

宽依赖往往对应着shuffle操作，需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中，通常可以在一个节点内完成转换。

当RDD分区丢失时（某个节点故障），spark会对数据进行重算。

对于宽依赖，重算时候，的父RDD分区对应多个子RDD分区，这样实际上父RDD 中只有一部分的数据是被用于恢复这个丢失的子RDD分区的，另一部分对应子RDD的其它未丢失分区，这就造成了多余的计算；

窄依赖的函数有：map, filter, union, join(父RDD是hash-partitioned ), mapPartitions, mapValues

宽依赖的函数有：groupByKey, join(父RDD不是hash-partitioned ), partitionBy

spark中如何划分stage：

Spark中的Stage其实就是一组并行的任务，任务是一个个的task

Spark任务会根据RDD之间的依赖关系，形成一个DAG有向无环图， DAGScheduler会把DAG划分相互依赖的多个stage，划分依据就是宽窄依赖，遇到宽依赖就划分stage，每个stage包含一个或多个task， stage是由一组并行的task组成。spark程序中可以因为不同的action触发众多的job，一个程序中可以有很多的job，每一个job是由一个或者多个stage构成的，后面的stage依赖于前面的stage，也就是说只有前面依赖的stage计算完毕后，后面的stage才会运行

pipeline管道计算模式，piepeline只是一种计算思想，一种模式

spark的pipeline管道计算模式相当于执行了一个高阶函数，也就是说来一条数据然后计算一条数据，会把所有的逻辑走完，然后落地，而MapReduce是1+1=2，2+1=3这样的计算模式，也就是计算完落地，然后再计算，然后再落地到磁盘或者内存。管道计算模式完全基于内存计算，所以比MapReduce快的原因。

并行度

a.并行度过低时，会出现资源限制的情况。此时可以提高并行度来充分利用更多的计算core。

b.并行度过高时，每个分区产生的间接开销累计起来会更大。评价并行度是否过高可以看你的任务是不是在瞬间(毫秒级)完成的，或者任务是不是没有读写任何数据。

调优方法:对于任何已有的RDD进行重新分区来获取更多/更少的分区数。重新分区:repartition()；减少分区：coalesce()，比repartition()更高效。

Mapreduce和Spark

答：两者都是用mr模型来进行并行计算:

1. hadoop的一个作业称为job，job里面分为map task和reduce task，每个task都是在自己的进程中运行的，当task结束时，进程也会结束。 (对应stage)

2)hadoop的job只有map和reduce操作，表达能力比较欠缺而且在mr过程中会重复的读写hdfs，造成大量的io操作，多个job需要自己管理关系。 (对应内存)

hadoop和spark的shuffle相同和差异？

1）从 high-level 的角度来看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer。

2）从 low-level 的角度来看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。

3）从实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

基于MLlib的机器学习

　　一般我们常用的算法都是单机跑的，但是想要在集群上运行，不能把这些算法直接拿过来用。一是数据格式不同，单机上我们一般是离散型或者连续型的数据，数据类型一般为array、list、dataframe比较多，以txt、csv等格式存储，但是在spark上，数据是以RDD的形式存在的，如何把ndarray等转化为RDD是一个问题；此外，就算我们把数据转化成RDD格式，算法也会不一样。举个例子，你现在有一堆数据，存储为RDD格式，然后设置了分区，每个分区存储一些数据准备来跑算法，可以把每个分区看做是一个单机跑的程序，但是所有分区跑完以后呢？怎么把结果综合起来？直接求平均值？还是别的方式？所以说，在集群上跑的算法必须是专门写的分布式算法。而且有些算法是不能分布式的跑。Mllib中也只包含能够在集群上运行良好的并行算法。

基于ML的机器学习

该软件包公开了三个主要的抽象类：转换器、评估器和管道。

转换器类，如Binarizer（连续变量值二值化）、Bucketizer(连续变量多值化)，idf（逆文档频率）、OneHotEncoder(将标签列编码为二进制向量列)、PCA、StandarScaler(标准化列)、Word2Vec。VectorAssembler(将多个数字列合并为一列向量)

评估器，分类的有LR、DT、GBT、RF、NB、MLP、OneVsRest。

管道，表示从转换到评估（具有一系列的不同阶段）的端到端的过程。

使用ML预测婴儿生存几率:

1.加载数据

2.创建转换器。使用OneHotEncoder方法来对BIRTH\_PLACE列进行编码。但是，该方法不接受StringType列，首先将该列转换为IntegerType。使用VectorAssembler创建一个单一的列，它将所有特征整合在一起。

3.创建评估器，使用逻辑回归。

4.创建管道

5.拟合模型。首先划分训练集和测试集, 然后 model= pipeline.fit(births\_train),model.transform(births\_test)。

6.评估模型的性能

机器学习:

支持向量机（Support Vector Machine, SVM）

基本模型是在特征空间上找到最佳的分离超平面使得训练集上正负样本间隔最大。SVM是用来解决二分类问题的有监督学习算法，在引入了核方法之后SVM也可以用来解决非线性问题。

两个异类支持向量到超平面的距离之和称为间隔。为了最大化间隔，把这个问题转化为凸优化问题，引入拉格朗日乘子，转化为对偶问题，为每条约束添加拉格朗日乘子(样本权重)，对w和b求偏导，带入原式消除这两个参数。

精妙之处:对新样本的预测，转化为计算新样本与训练样本的内积，此外，没有内积就没有核函数。核函数只是用来计算低纬空间映射到高维空间之后的内积的一种简便方法，隐含着一个从低纬空间到高维空间的映射，而这个映射低纬空间中的线性的不可分变为线性可分。

序列最小优化算法（以下简称SMO算法）。SVM推导到最后，等效为一个二次规划问题（目标函数为二次函数,约束均用线性形式给出的非线性规划问题称为二次规划问题）。其基本思路就是一次迭代只优化两个变量而固定剩余的变量。直观地讲就是将一个大的优化问题分解为若干个小的优化问题，这些小的优化问题往往是易于求解的。第一步选取一对[clip_image015](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182042575630.png)和[clip_image017](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182042585531.png)，选取方法使用启发式方法（后面讲）。第二步，固定除[clip_image015[1]](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182042591528.png)和[clip_image017[1]](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182043003381.png)之外的其他参数，确定W极值条件下的[clip_image015[2]](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182043017742.png)，[clip_image017[2]](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182043017676.png)由[clip_image015[3]](http://images.cnblogs.com/cnblogs_com/jerrylead/201103/201103182043021722.png)表示。

SMO之所以高效就是因为在固定其他参数后，对一个参数优化过程很高效。

优点: 解决小样本、非线性及高维模式识别问题中表现出许多特有的优势

缺点: 对于每个高维空间在此空间的映射F,如何确定F也就是核函数,现在还没有合适的方法,所以对于一般的问题,SVM只是把高维空间的复杂性的困难转为了求核函数的困难.而且即使确定核函数以后,在求解问题分类时,要求解函数的二次规划,这就需要大量的存储空间.这也是SVM的一个问题.

逻辑回归（Support Vector Machine, SVM）

逻辑回归假设数据服从伯努利分布,通过极大化似然函数的方法，运用梯度下降来求解参数，来达到将数据二分类的目的。y值是一个连续的变量。划定一个阈值，y值大于这个阈值的是一类，y值小于这个阈值的是另外一类。阈值具体如何调整根据实际情况选择。一般会选择0.5做为阈值来划分。

最大似然估计的目的就是：利用已知的样本结果，反推最有可能（最大概率）导致这样结果的参数值。原理：极大似然估计是建立在极大似然原理的基础上的一个统计方法，是概率论在统计学中的应用。极大似然估计提供了一种给定观察数据来评估模型参数的方法，即：“模型已定，参数未知”。通过若干次试验，观察其结果，利用试验结果得到某个参数值能够使样本出现的概率为最大，则称为极大似然估计。

似然函数即是联合概率密度。

优点: **形式简单。**   
**训练速度较快。**分类的时候，计算量仅仅只和特征的数目相关。  
**方便输出结果调整。**逻辑回归可以很方便的得到最后的分类结果，因为输出的是每个样本的概率分数，我们可以很容易的对这些概率分数进行切阈值

缺点: **准确率并不是很高。**因为形式非常的简单(非常类似线性模型)，很难去拟合数据的真实分布。  
**很难处理数据不平衡的问题。**举个例子：如果我们对于一个正负样本非常不平衡的问题比如正负样本比 10000:1.我们把所有样本都预测为正也能使损失函数的值比较小。

逻辑回归和线性回归首先都是广义的线性回归，  
其次经典线性模型是平方损失函数，而逻辑回归则是似然函数，  
另外线性回归在整个实数域范围内进行预测，敏感度一致，而分类范围，需要在[0,1]。逻辑回归就是一种减小预测范围，将预测值限定为[0,1]间的一种回归模型，因而对于这类问题来说，逻辑回归的鲁棒性比线性回归的要好。

神经网络

神经网络学习的过程，就是根据训练数据来调整神经元之间的连接权。

感知机只有输出层神经单元进行激活处理，即只拥有一层功能神经元，其学习能力非常有限。

多层前馈神经网络：输入层神经单元仅接受输入，隐层与输出层神经元对信号进行加工，

BP神经网络是一种按误差反向传播(bp算法)训练的多层前馈网络，基于梯度下降策略，以目标的负梯度方向对参数进行调整。

基本BP算法包括信号的前向传播和误差的反向传播两个过程。即计算误差输出时按从输入到输出的方向进行，而调整权值和阈值则从输出到输入的方向进行。通过调整阈值，使误差沿梯度方向下降，经过反复学习训练，确定与最小误差相对应的网络参数(权值和阈值)，训练即告停止。

优点:

（1）能够自适应、自主学习。这是BP算法的根本以及其优势所在，BP算法根据预设的参数更新规则，不断地调整神经网络中的参数，以达到最符合期望的输出。（2）拥有较强的非线性映射能力。（3）严谨的推导过程。误差的反向传播过程，采用链式法测。

缺点：

（1）由于BP神经网络中的参数众多，每次都需要更新数量较多的阈值和权值，故会导致收敛速度过慢。（3）从数学角度看，BP算法是一种速度较快的梯度下降算法，很容易陷入局部最小值的问题。

CNN:

特点是隐层分为卷积层和池化层。卷积层通过一块块卷机核在原始图像上平移来提取特征。池化层通过汇聚特征后稀疏参数来减少要学习的参数，降低网格的复杂度。（不管是卷积还是池化，都是（W-F+2p）/s +1）

卷积运算的主要目的是使原信号特征增强，并且降低噪音

池化主要是防止过拟合。

权共享: 让一组神经元使用相同的连接权。

下图左：如果我们有1000x1000像素的图像，有1百万个隐层神经元，那么他们全连接的话（每个隐层神经元都连接图像的每一个像素点），就有1000x1000x1000000=10^12个连接，也就是10^12个权值参数。然而图像的空间联系是局部的，就像人是通过一个局部的感受野去感受外界图像一样，每一个神经元都不需要对全局图像做感受，每个神经元只感受局部的图像区域，然后在更高层，将这些感受不同局部的神经元综合起来就可以得到全局的信息了。这样，我们就可以减少连接的数目，也就是减少神经网络需要训练的权值参数的个数了。如下图右：假如局部感受野是10x10，隐层每个感受野只需要和这10x10的局部图像相连接，所以1百万个隐层神经元就只有一亿个连接，即10^8个参数。比原来减少了四个0（数量级），这样训练起来就没那么费力了，但还是感觉很多的啊，那还有啥办法没？

       好了，你就会想，这样提取特征也忒不靠谱吧，这样你只提取了一种特征啊？对了，真聪明，我们需要提取多种特征对不？假如一种滤波器（卷积核），也就是一种卷积核就是提出图像的一种特征，例如某个方向的边缘。那么我们需要提取不同的特征，怎么办，加多几种滤波器不就行了吗？对了。所以假设我们加到100种滤波器，每种滤波器的参数不一样，表示它提取输入图像的不同特征，例如不同的边缘。这样每种滤波器去卷积图像就得到对图像的不同特征的放映，我们称之为Feature Map。所以100种卷积核就有100个Feature Map。这100个Feature Map就组成了一层神经元。到这个时候明了了吧。我们这一层有多少个参数了？100种卷积核x每种卷积核共享100个参数=100x100=10K，也就是1万个参数。才1万个参数啊！亲！（又来了，受不了了！）见下图右：不同的颜色表达不同的滤波器。

深度学习另一个理解: 可看作是在对输入信号进行逐层加工，从而把初始的与输入目标之间不太密切的输入表示，转化成与输出目标更密切的表示。

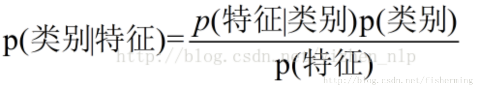
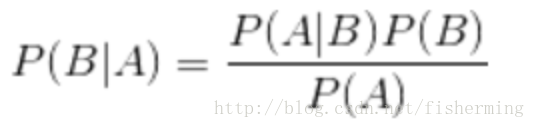
全连接层: 变为1维

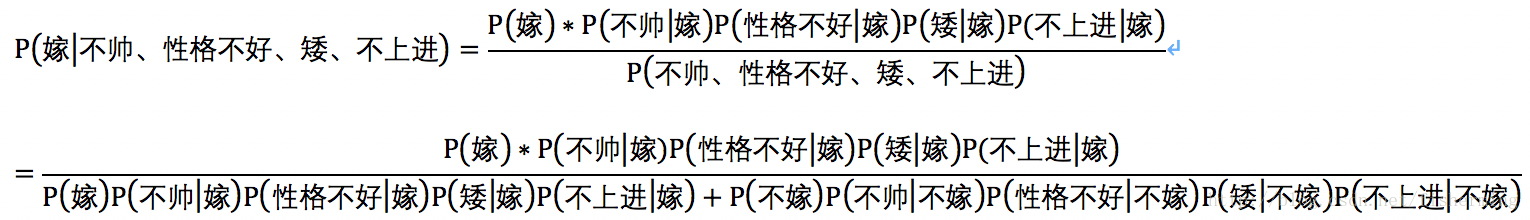
最后变为softmax

激活函数:解决非线性问题。Relu(瑞撸函数)，max(x,0)大于0时梯度不衰减，从而缓解梯度不消失问题。

朴素贝叶斯

朴素贝叶斯中的朴素一词的来源就是假设各特征之间相互独立。这一假设使得朴素贝叶斯算法变得简单，但有时会牺牲一定的分类准确率。对于目标求解为不同的类别，贝叶斯公式的分母总是相同的。所以，只求解分子即可。对于分母，条件独立性假设是假定“输入参数的各个特征”对“输出参数的值”的影响相互独立，而不是假设特征本身相互独立。 所以“P(不帅、性格不好、矮、不上进)”并不等于“P(不帅)P(性格不好)P(矮)P(不上进)”。





优点：

算法逻辑简单,易于实现(算法思路很简单，只要使用贝叶斯公式转化即可）

缺点：

（1）朴素贝叶斯假设属性之间相互独立，这种假设在实际过程中往往是不成立的。在属性之间相关性越大，分类误差也就越大。

（2）需要知道先验概率，且先验概率很多时候取决于假设，假设的模型可以有很多种，因此在某些时候会由于假设的先验模型的原因导致预测效果不佳。

# 激活函数

引入非线性因素。在我们面对线性可分的数据集的时候，简单的用线性分类器即可解决分类问题。但是现实生活中的数据往往不是线性可分的，面对这样的数据，一般有两个方法：引入非线性函数、线性变换。

在神经网络中，为了避免单纯的线性组合，我们在每一层的输出后面都添加一个激活函数（sigmoid、tanh、ReLu等等）

1）深度学习往往需要大量时间来处理大量数据，模型的收敛速度是尤为重要的。所以，总体上来讲，训练深度学习网络尽量使用zero-centered数据 (可以经过数据预处理实现) 和zero-centered输出。所以要尽量选择输出具有zero-centered特点的激活函数以加快模型的收敛速度。

2）如果使用 ReLU，那么一定要小心设置 learning rate，而且要注意不要让网络出现很多 “dead” 神经元，如果这个问题不好解决，那么可以试试 Leaky ReLU、PReLU 或者 Maxout.

3）最好不要用 sigmoid，你可以试试 tanh，不过可以预期它的效果会比不上 ReLU 和 Maxout.

Sigmoid

它能够把输入的连续实值变换为0和1之间的输出，特别的，如果是非常大的负数，那么输出就是0；如果是非常大的正数，输出就是1.

缺点1：在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。如果我们初始化神经网络的权值为 [0,1]之间的随机值，由反向传播算法的数学推导可知，梯度从后向前传播时，每传递一层梯度值都会减小为原来的0.25倍，如果神经网络隐层特别多，那么梯度在穿过多层后将变得非常小接近于0，即出现梯度消失现象；当网络权值初始化为 (1,+∞) 区间内的值，则会出现梯度爆炸情况。

缺点2：Sigmoid 的 output 不是0均值(即zero-centered），产生的一个结果就是：如x>0, 那么对w求局部梯度则都为正，这样在反向传播的过程中w要么都往正方向更新，要么都往负方向更新，导致有一种捆绑的效果，使得收敛缓慢。

Tanh tanh(x)=2\*sigmoid(2\*x)-1

sigmoid在输入处于[-1,1]之间时，函数值变化敏感，一旦接近或者超出区间就失去敏感性，处于饱和状态，影响神经网络预测的精度值。tanh的输出和输入能够保持非线性单调上升和下降关系，渐进于0、1，比sigmoid函数延迟了饱和期,

解决了Sigmoid函数的不是zero-centered输出问题，

缺点： 梯度消失（gradient vanishing）的问题和幂运算的问题仍然存在。

Relu 修正线性单元ReLU（rectified linear unit）

优点：

1） 解决了梯度消失、爆炸的问题

2）计算方便，计算速度快

3）加快了网络的训练

缺点

1）ReLU的输出不是zero-centered

2）由于负数部分恒为0，会导致一些神经元无法激活

ELU (Exponential Linear Units)

融合了sigmoid和ReLU，左侧具有软饱和性，右侧无饱和性。右侧线性部分使得ELU能够缓解梯度消失，而左侧软饱能够让ELU对输入变化或噪声更鲁棒。

不会有Dead ReLU问题

输出的均值接近0，zero-centered

Maxout

对上图做个说明，第i层有3个节点，红点表示，而第（i+1）层有4个结点，用彩色点表示，此时在第（i+1）层采用maxout（k=3）。我们看到第（i+1）层的每个节点的激活值都有3个值，3次计算的最大值才是对应点的最终激活值。我举这个例子主要是为了说明，决定结点的激活值的时候并不是以层为单位，仍然以节点为单位。

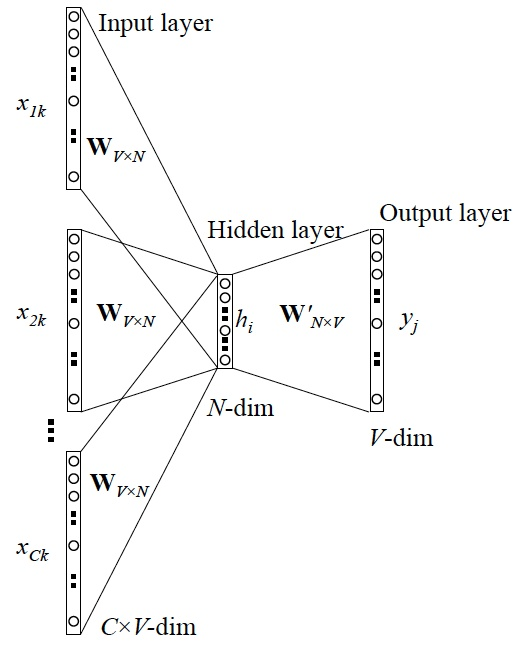
这样 Maxout 神经元就拥有 ReLU 单元的所有优点（线性和不饱和），而没有它的缺点（死亡的 ReLU 单元）。然而和 ReLU 对比，它每个神经元的参数数量增加了一倍，这就导致整体参数的数量激增。

Word2vec

单词如果使用独热表示，那么每一步输入的向量维数会非常大，独热表示实际上完全平等看待了单词表中的所有单词，忽略了单词之间的联系。

Word2vec指学习一个映射f，它可以将单词变成向量表示。主要有Skip-Gram和CBOW两种模型。 Skip-Gram是给定input word来预测上下文。而CBOW是给定上下文，来预测input word。

CBOW(连续词袋模型)



(1) 输入层：上下文单词的onehot. {假设单词向量空间dim为V，上下文单词个数为C}.

(2) 所有onehot分别乘以共享的输入权重矩阵W. {VN矩阵，N为自己设定的数，初始化权重矩阵W}.

(3) 所得的向量 {因为是onehot所以为向量} 相加求平均作为隐层向量, size为1N.

(4) 乘以输出权重矩阵W' {NV},得到向量 {1V} 激活函数处理得到V-dim概率分布

(5) 概率最大的index所指示的单词为预测出的中间词（target word）与true label的onehot做比较，误差越小越好

**负采样**

由于训练词向量模型的目标不是为了得到一个多么精准的语言模型，而是为了获得它的副产物——词向量。所以要做到的不是在几十万个token中艰难计算下一词，而只需能做到在几个词中找到对的那个词就行。

缺点是上下文无关(static）

因而为了让句子有一个整体含义(context)，会在下游具体的NLP任务中基与词向量的序列做encoding操作。

GloVe

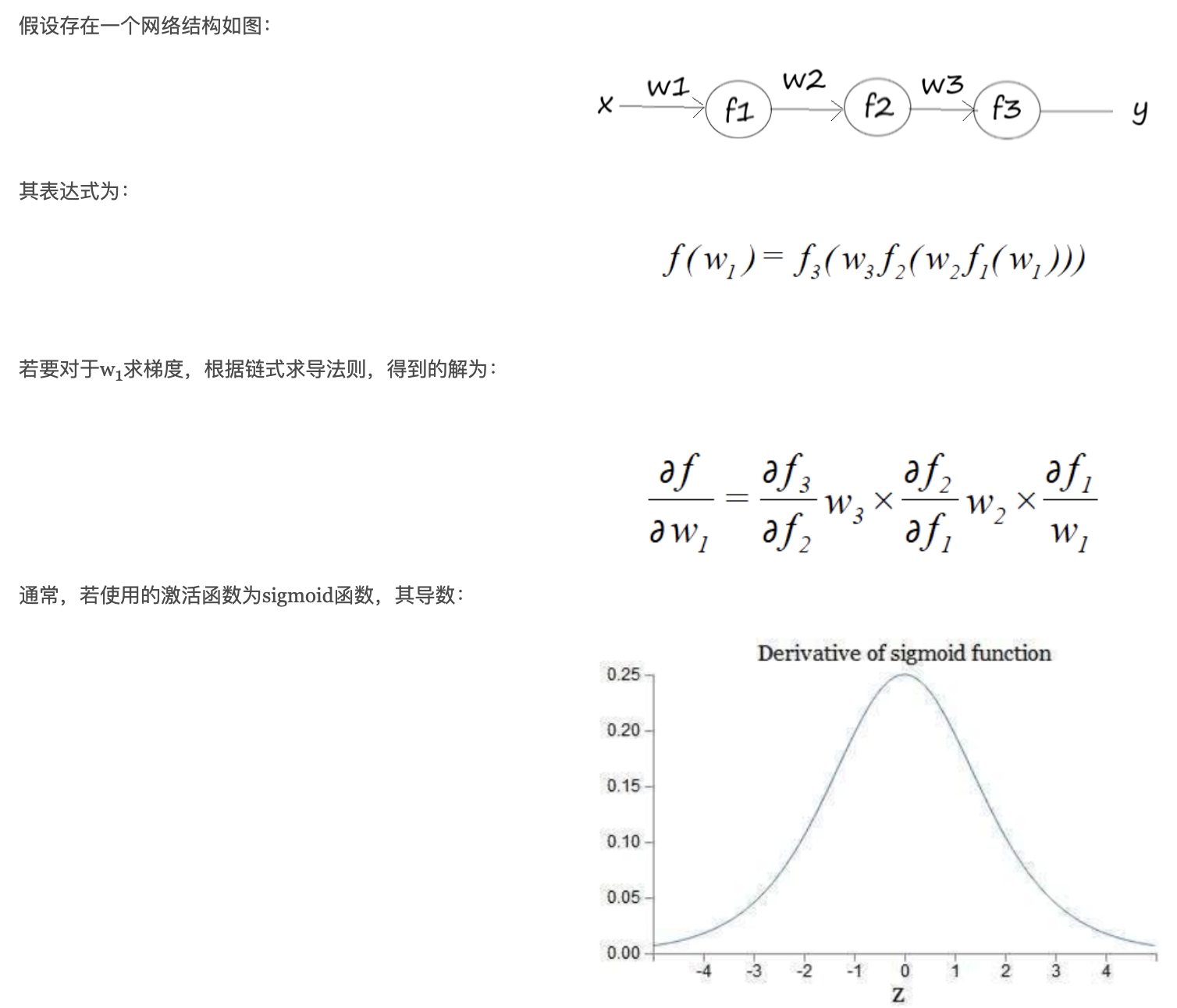
设共现矩阵为X，其元素为Xi,j。X i,j的意义为：在整个语料库中，单词i和单词j共同出现在一个窗口中的次数。(如5 love but you love him i)。GloVe模型没有使用神经网络的方法。

Glove和skip-gram、CBOW模型对比

我的理解是skip-gram、CBOW每次都是用一个窗口中的信息更新出词向量，但是Glove则是用了全局的信息（共线矩阵），也就是多个窗口进行更新.

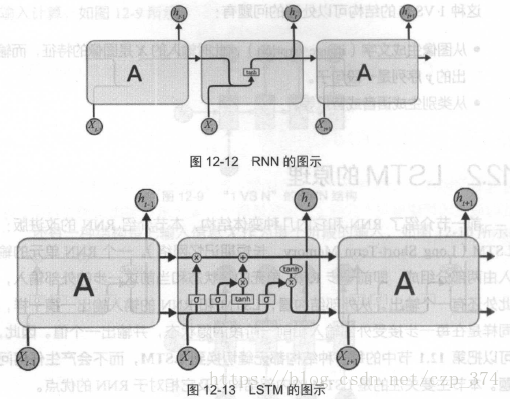
LSTM:（Long Short-Term Memory，长短期记忆网络）

梯度爆炸和梯度消失 Derivative(倒数)

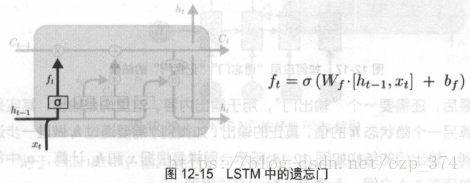


这样可以看到，如果我们使用标准化初始w，那么各个层次的相乘都是0-1之间的小数，而激活函数f的导数也是0-1之间的数，其连乘后，结果会变的很小，导致梯度消失。梯度消失、爆炸，其根本原因在于反向传播训练法则，属于先天不足。

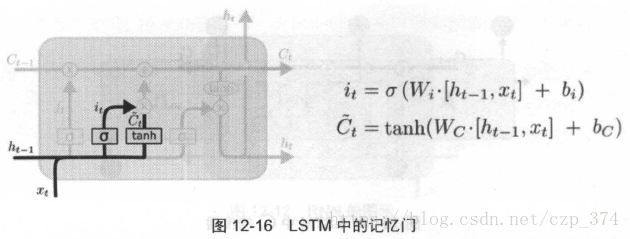
回顾RNN的公式ht=f(Uxt+Wht−1+b)。从这个式子中可以看出，RNN每一层的隐状态都由前一层的隐状态经过变换和激活函数得到， 反向传播求导时最终得到的导数会包含每一步梯度的连乘，这会引起梯度爆炸或梯度消失，所以RNN很难处理“长程依赖”问题，即无法学到序列中蕴含的间隔时间较长的规律。LSTM在隐状态计算时以加法代替了这里的迭代变换，可以避免梯度消失的问题,使网络学到长程的规律。



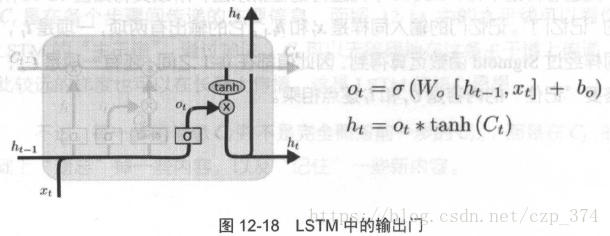
和RNN有所不同，LSTM的隐状态有两部分**，**一部分是h，一部分是C。C是在各个步骤传递的主要信息，而图12-14中的水平线可以看作是LSTM的“主干道”（上面那条水平线）。通过加法，Ct可以无障碍地在这条主干道上传递，因此较远的梯度也可以在长程上传播。这是LSTM的核心思想。 不过，每一步的信息Ct并不是完全照搬前一步的Ct−1，而是在Ct−1的基础上“遗忘”掉一些内容，以及“记住”一些新内容。



LSTM的每一个单元都有一个“遗忘门”，用来控制遗忘掉Ct−1的那些部分。遗忘门的结构如图12-15所示。σ是Sigmoid激活函数，它的输出在0-1之间。最终遗忘门的输出是和Ct−1相同形状的矩阵，这个矩阵会和Ct−1逐点相乘，决定遗忘哪些东西。显然，遗忘门输出接近0的位置的内容是要遗忘的，而接近1的部分是要保留的。遗忘门的输入是xt和ht−1。



光遗忘肯定是不行，LSTM单元还得记住新东西。所以又有如图12-16所示的“记忆门”。记忆门的输入同样是xt和ht−1，它的输出有两项，一项是It，It同样经过Sigmoid函数运算得到，因此值都在0-1之间，还有一项C̃ t，最终要“记住”的内容是C̃ t和it逐点相乘。



最后，还需要一个“输出门”，用于输出内容。这里说是输出，其实是去计算另一个隐状态ht的值，真正的输出（如类别）需要通过ht做进一步运算得到。输出门的结构如图12-18所示。同样是根据xt和ht−1计算，Ot中每一个数值在0~1之间，ht通过ot∗tanh(Ct)得到。

LSTM的缺点:

lstm只能缓解梯度消失，并不能完全解决；因为最新的状态单元可以是过去的一个线性组合，过去的信息传递到现在主要是由每个时间步的遗忘门参数连乘决定的

只能记住100个量级的序列，而不是1000个量级，或者更长的序列。

见Bi-LSTM

Bi-LSTM

但是利用LSTM对句子进行建模还存在一个问题：无法编码从后到前的信息。在更细粒度的分类时，如对于强程度的褒义、弱程度的褒义、中性、弱程度的贬义、强程度的贬义的五分类任务需要注意情感词、程度词、否定词之间的交互。举一个例子，“这个餐厅脏得不行，没有隔壁好”，这里的“不行”是对“脏”的程度的一种修饰，通过BiLSTM可以更好的捕捉双向的语义依赖，即Bi-LSTM实现上下文相关（context）。

从前到后和后到前分别做一遍LSTM的encoding操作，从而获得两个方向的token(词)联系，进而获得句子的context。

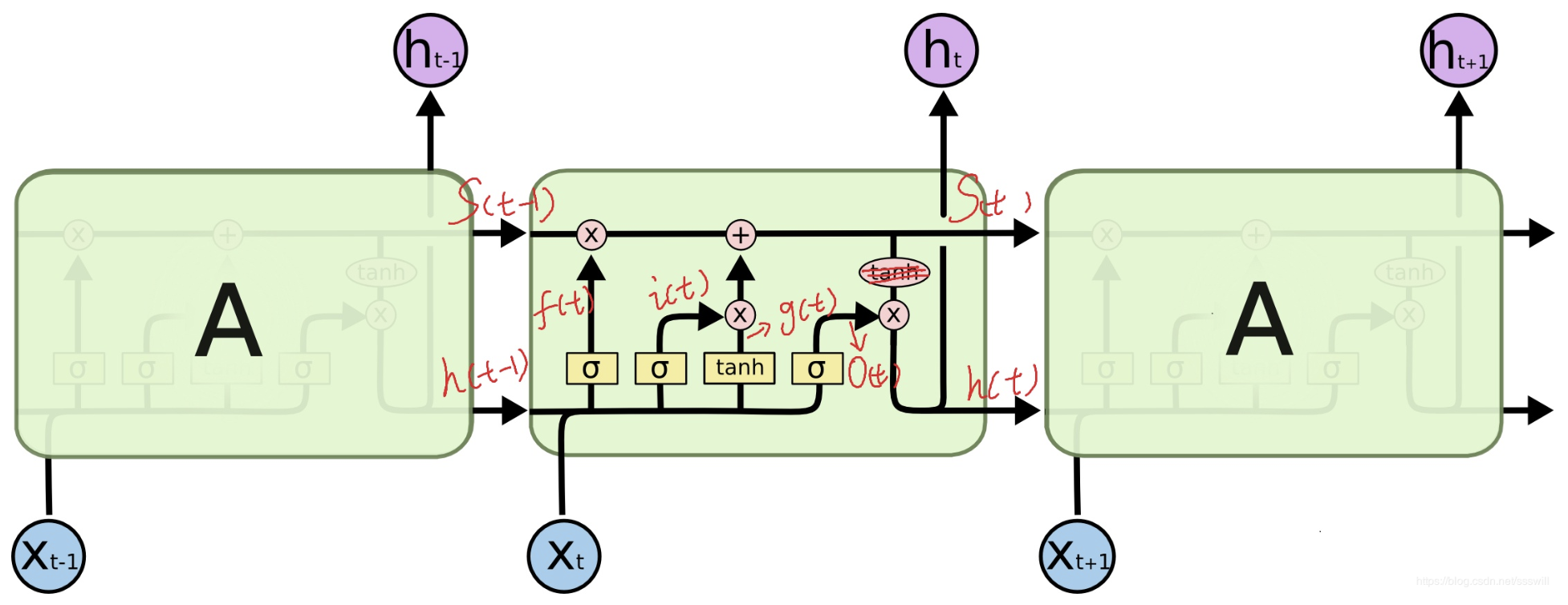
不完全双向

模型的前向和后向LSTM两个模型是分别训练的，最后得到的隐层向量直接拼接得到结果向量，并且在最后的Loss function中也是前向和后向的loss function直接相加，并非完全同时的双向计算。

自己看见自己

经过两层的双向操作，每个位置上的输出就已经带有了原本这个位置上的词的信息了。这样的“窥探”会导致模型预测词的任务变得失去意义，因为模型已经看到每个位置上是什么词了。

LSTM参数理解:



中间的 cell 里面有四个黄色小框，每一个小黄框代表一个前馈网络层，num\_units就是这个层的隐藏神经元个数，就这么简单。其中1、2、4的激活函数是 sigmoid，第三个的激活函数是 tanh。此外：

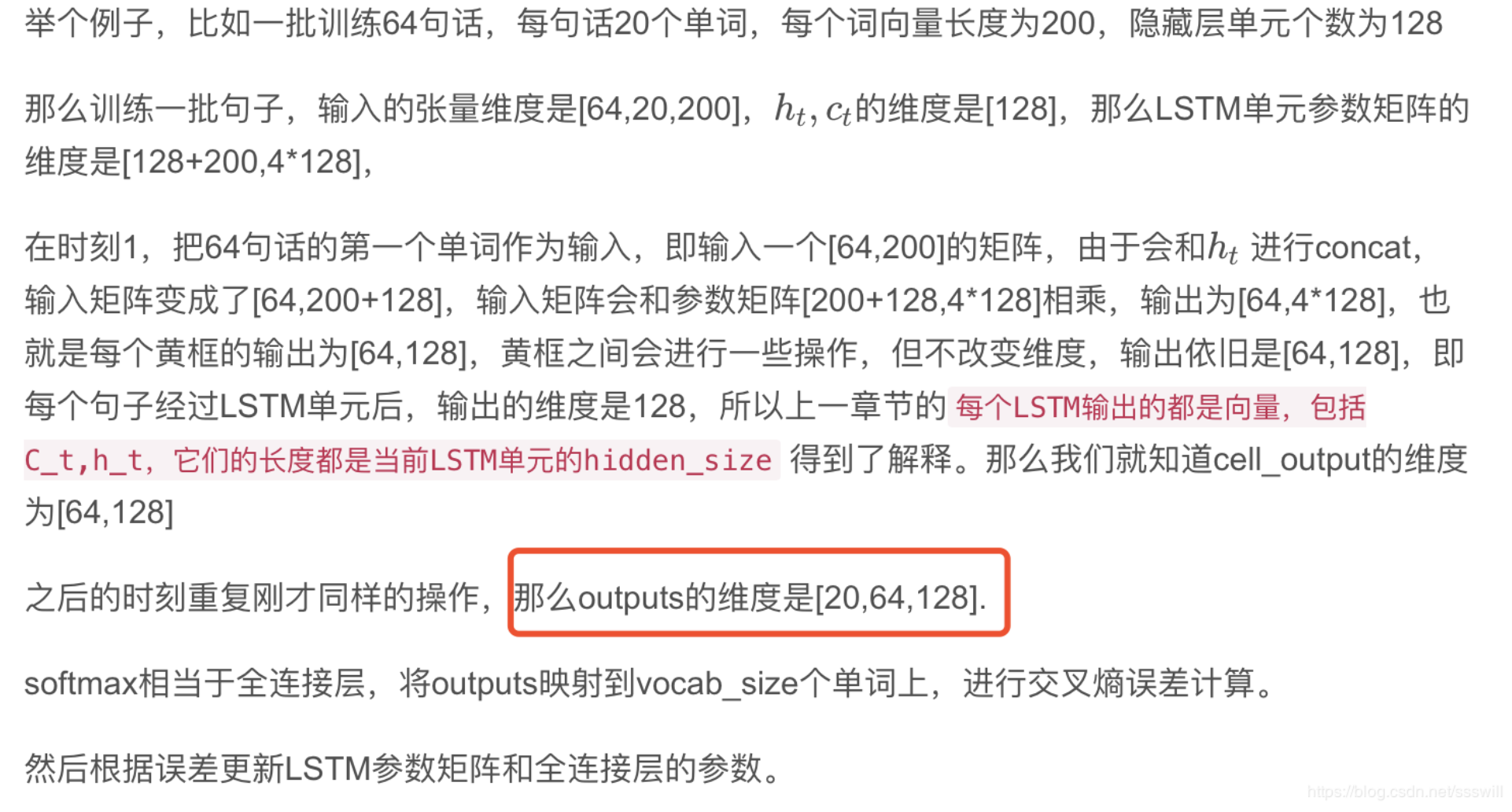
(1) cell 的状态是一个向量，是有多个值的

(2)上一次的状态 h(t-1)是怎么和下一次的输入 x(t) 结合起来的，直白的说就是把二者直接拼起来（concat），比如 x是28位的向量，h(t-1)是128位的，那么拼起来就是156位的向量。

(3)cell 的权重是共享的，这是什么意思呢？这是指这张图片上有三个绿色的大框，代表三个 cell 对吧，但是实际上，它只是代表了一个 cell 在不同时序时候的状态，所有的数据只会通过一个 cell，然后不断更新它的权重。

4、计算：那么一层的 LSTM 的参数有多少个？根据第 3 点的说明，我们知道参数的数量是由 cell 的数量决定的，这里只有一个 cell，所以参数的数量就是这个 cell 里面用到的参数个数。假设 num\_units 是128，输入是28位的，那么根据上面的第 2 点，可以得到，四个小黄框的参数一共有 （128+28）\*128\*4，也就是156 \* 512，不过还要加上输出的时候的激活函数的参数，假设是10个类的话，就是128\*10的 W 参数和10个bias 参数

LSTM的输入和输出



NLP

TF−IDF=TF(词频)∗IDF(逆文档频率)

词频 (TF) 是一词语出现的次数除以该文件的总词语数。假如一篇文件的总词语数是100个，而词语“母牛”出现了3次，那么“母牛”一词在该文件中的词频就是3/100=0.03。一个计算文件频率 (IDF) 的方法是文件集里包含的文件总数除以测定有多少份文件出现过“母牛”一词。所以，如果“母牛”一词在1,000份文件出现过，而文件总数是10,000,000份的话，其逆向文件频率就是 lg(10,000,000 / 1,000)=4。最后的TF-IDF的分数为0.03 \* 4=0.12。

BERT（Bidirectional Encoder Representation from Transformers）

Transformer舍弃了RNN的循环式网络结构，完全基于注意力机制来对一段文本进行建模。Transformer所使用的注意力机制的核心思想是去计算一句话中的每个词对于这句话中所有词的相互关系，然后认为这些词与词之间的相互关系在一定程度上反应了这句话中不同词之间的关联性以及重要程度。因此再利用这些相互关系来调整每个词的重要性（权重）就可以获得每个词新的表达。这个新的表征不但蕴含了该词本身，还蕴含了其他词与这个词的关系，因此和单纯的词向量相比是一个更加全局的表达。

**Masked Language Model**

在输入一句话的时候，随机地选一些要预测的词，然后用一个特殊的符号来代替它们。尽管模型最终还是会看到所有位置上的输入信息，但由于需要预测的词已经被特殊符号代替，所以模型无法事先知道这些位置上是什么词，这样就可以让模型根据所给的标签去学习这些地方该填的词了。

**sentence-level representation**

预测输入BERT的两端文本是否为连续的文本，作者指出引入这个任务可以更好地让模型学到连续的文本片段之间的关系。类似word2vec的单词级负采样。

**位置编码**

Transformer模型并没有捕捉顺序序列的能力，也就是说无论句子的结构怎么打乱，Transformer都会得到类似的结果。

为了解决这个问题，论文中在编码词向量时引入了位置编码（Position Embedding）的特征。具体地说，位置编码会在词向量中加入了单词的位置信息。

**Transformer —— attention is all you need**

Transformer中抛弃了传统的CNN和RNN，整个网络结构完全是由Attention机制组成。更准确地讲，Transformer由且仅由self-Attenion和Feed Forward Neural Network组成。

对比RNN

作者采用Attention机制的原因是考虑到RNN（或者LSTM，GRU等）的计算限制为是顺序的，也就是说RNN相关算法只能从左向右依次计算或者从右向左依次计算，这种机制带来了两个问题：

(1) 时间片 t 的计算依赖 t-1 时刻的计算结果，这样限制了模型的并行能力；

(2) 顺序计算的过程中信息会丢失，尽管LSTM等门机制的结构一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象,LSTM依旧无能为力。

基本结构

transformer的本质上是一个Encoder-Decoder的结构。

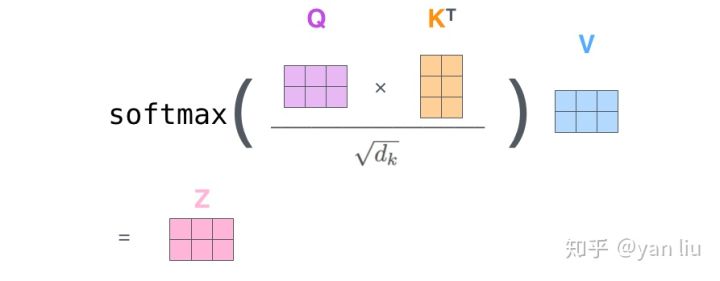
首先通过Word2Vec等词嵌入方法将输入语料转化成特征向量，从编码器输入的句子首先会经过一个自注意力（self-attention）层，这层帮助编码器在对每个单词编码时关注输入句子的其他单词。自注意力层的输出会传递到前馈（feed-forward）神经网络中。每个位置的单词对应的前馈神经网络都完全一样。

接下来我们看看Transformer的一个核心特性，在这里输入序列中每个位置的单词都有自己独特的路径流入编码器。在自注意力层中，这些路径之间存在依赖关系。而前馈（feed-forward）层没有这些依赖关系。因此在前馈（feed-forward）层时可以并行执行各种路径。

self-attention(自注意力机制)

其核心内容是为输入向量的每个单词学习一个权重。比如当模型处理这个单词“it”的时候，自注意力机制会允许“it”与“animal”建立联系。当我们在编码器#5（栈中最上层编码器）中编码“it”这个单词的时，注意力机制的部分会去关注“The Animal”，将它的表示的一部分编入“it”的编码中。

计算自注意力的第一步就是从每个编码器的输入向量（每个单词的词向量）中生成三个向量。也就是说对于每个单词，我们创造查询向量、键向量和值向量。它们是通过3个不同的权值矩阵由嵌入向量 X 乘以三个不同的权值矩阵 W^Q W^K， W^V 得到的。



multi-headed attention(多头注意力机制)

Multi-Head Attention相当于 h个不同的self-attention的集成（ensemble）。

如，当编码“it”一词时，一个注意力头集中在“animal”上，而另一个则集中在“tired”上，从某种意义上说，模型对“it”一词的表达在某种程度上是“animal”和“tired”的代表。

优点

(1)设计创新，因为其抛弃了在NLP中最根本的RNN或者CNN并且取得了非常不错的效果。

(2)算法的并行性非常好，符合目前的硬件（主要指GPU）环境。

缺点: 粗暴的抛弃RNN和CNN虽然非常炫技，但是它也使模型丧失了捕捉局部特征的能力