2020-12-21

AI 百题训练班

参考答案

目录

机器学习基础

特征工程

为什么要对特征做归一化

注意：理解特征归一化所适用的模型场景

特征归一化是将所有特征都统一到一个大致相同的数值区间内，通常为[0, 1]。常用的特征归一化方法有：

1. Min-Max Scaling

对原始数据进行线性变换，使结果映射到[0, 1]的范围，实现对数据的等比例缩放。

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

其中 X_{min}，X_{max} 分别为数据的最小值和最大值

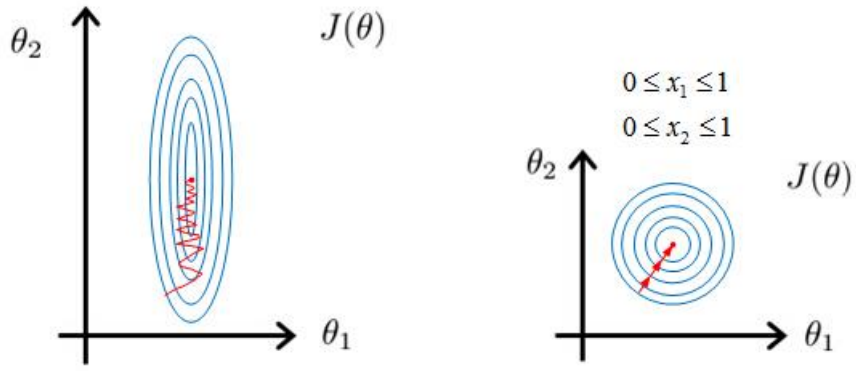
2. Z-Score Normalization

将原始数据映射到均值为 0，标准差为 1 的分布上。

$$X_{norm} = \frac{X - \mu}{\sigma}$$

其中μ为原始特征的均值，而σ为原始特征的标准差。

在采用基于梯度更新的学习方法（包括线性回归，逻辑回归，支持向量机，神经网络等）对模型求解的过程中，未归一化的数值特征在学习时，梯度下降较为抖动，模型难以收敛（通常需要较长的时间模型才能收敛）；而归一化之后的数值特征则可以使梯度下降较为稳定，进而减少梯度计算的次数，也更容易收敛。下图中，左边为特征未归一化时，模型的收敛过程；而右边是经过特征归一化之后模型的收敛过程。其中 $J(\theta)$ 表示损失函数，图中的圈代表损失函数的等值线；θ_1, θ_2分别是模型的两个参数，x_1, x_2是这两个模型参数对应的特征值。



什么是组合特征， 如何处理高维组合特征

注意：这里的特征组合主要指的是类别特征(Categorical Feature)之间的组合

狭义的组合特征即将类别特征（Categorical feature）两个或者多个特征组合（数学里面的组合概念）起来，构成高阶组合特征。

比如：假设 Mac 笔记本电脑的 CPU 型号和 SSD 大小对是否购买行为的影响用下面的表格表示

是否购买	CPU 型号		SSD 大小	
	Intel i5	Intel i7	256 GB	512GB
1	1	0	1	0
0	0	1	1	0
0	1	0	0	1
1	0	1	0	1

那么 CPU 型号和 SSD 大小的组合特征对是否购买行为的影响为：

是否购买	CPU 型号和 SSD 大小 组合特征			
	CPU = Intel i5 SSD = 256 GB	CPU = Intel i7 SSD = 256 GB	CPU = Intel i5 SSD = 512 GB	CPU = Intel i7 SSD = 512 GB
1	1	0	0	0
0	0	1	0	0
0	0	0	1	0
1	0	0	0	1

组合特征的不同取值的个数（number of unique values）为单个特征的不同取值的个数的乘积。假设数据的特征向量为 $X = (x_1, x_2, \ldots, x_k)$ 则， $|< x_i, x_j >| = |x_i| * |x_j|$ 其中$<x_i, x_j>$ 为特征x_i和特征x_j的组合特征，$|x_i|$ 表示特征x_i 不同取值的个数，$|x_j|$表示特征x_j不同取值的个数。

假设采用以线性模型为基础的模型来拟合特征时，比如以逻辑回归为例：

$$Y = sigmoid(\sum_i \sum_j w_{ij} < x_i, x_j >)$$

需要学习的参数w_{ij}的长度为$|< x_i, x_j >|$；如果$|x_i| = m, |x_j| = n$ ，则参数规模为 m*n。当 m 和 n 非常大时，经过特征组合后的模型就会变得非常复杂。一个可行的方法就是，特征的 Embedding，即将x_i，x_j分别用长度为 k 的低维向量表示（$k \ll m, k \ll n$）；那么学习参数的规模则变为 $m * k + n * k + k * k$.

One-hot 的作用是什么，为什么不直接使用数字作为表示

注意：理解 One-hot 编码和数字编码的特点

One-hot 主要用来编码类别特征，即采用哑变量(dummy variables) 对类别进行编码。它的作用是避免因将类别用数字作为表示而给函数带来抖动。直接使用数字会给将人工误差而导致的假设引入到类别特征中，比如类别之间的大小关系，以及差异关系等等。

什么是数据不平衡，如何解决

数据不平衡主要指的是在有监督机器学习任务中，样本标签值的分布不均匀。这将使得模型更倾向于将结果预测为样本标签分布较多的值，从而使得少数样本的预测性能下降。绝大多数常见的机器学习算法对于不平衡数据集都不能很好地工作。

解决方法：

1. 重新采样训练集
    a. 欠采样 –通过减少丰富类的大小来平衡数据集
    b. 过采样 – 增加稀有样本，通过使用重复，自举或合成少数类
2. 设计使用不平衡数据集的模型
    a. 在代价函数增大对稀有类别分类错误的惩罚权重。

对于树形结构为什么不需要归一化

决策树的学习过程本质上是选择合适的特征，分裂并构建树节点的过程；而分裂节点的标准是由树构建前后的目标增益（比如信息增益和信息增益率）决定的。这些指标与特征值之间的数值范围差异并无关系。

模型评估

请比较欧式距离与曼哈顿距离

欧式距离，即欧几里得距离，表示两个空间点之间的直线距离。

$$d = (\sum_{k=1}^{n} |a_k - b_k|^2)^{\frac{1}{2}}$$

曼哈顿距离，即所有维度距离绝对值之和。

$$d = \sum_{k=1}^{n} |a_k - b_k|$$

在基于地图，导航等应用中，欧式距离表现得理想化和现实上的距离相差较大；而曼哈顿距离就较为合适；另外欧式距离根据各个维度上的距离自动地给每个维度计算了一个“贡献权重”，这个权重会因为各个维度上距离的变化而动态的发生变化；而曼哈顿距离的每个维度对最终的距离都有同样的贡献权重。

为什么一些场景中使用余弦相似度而不是欧式距离

假设有 A 和 B 两个向量，其余弦相似度定义为 $\cos(A,B) = \frac{A*B}{\|A\|_2\|B\|_2}$，即两个向量夹角的余弦。它关注的是向量之间的角度关系，相对差异，而不关心它们的绝对大小；其取值范围在[-1, 1]之间；两个向量相同时为 1，正交时为 0，相反时为-1. 即在取值范围内，余弦距离值越大，两个向量越接近；余弦距离为向量之间的相似度量提供了一个稳定的指标，无论向量的维度多与少；特征的取值范围大与小。余弦距离的取值范围始终都能保持在[-1, 1]。余弦相似度广泛应用在文本，图像和视频领域。相比之下欧氏距离则受到维度多少，取值范围大小以及可解释性的限制。当特征的取值以及特征向量经过模长归一化之后，余弦距离和欧氏距离又存在以下的单调关系。

$$\|A - B\|_2^2 = \sqrt{2(1 - \cos(A,B)}$$

其中$\|A\|_2^2 = 1, \|B\|_2^2 = 1$

在模型评估过程中，过拟合和欠拟合具体指什么现象

过拟合是指模型对于训练数据拟合呈过当的情况，反映到评估指标上，就是模型在训练集上的表现好，但是在测试集和新数据上的表现较差。欠拟合指的是模型在训练和预测时表现都不好。用模型在数据上的偏差和方差指标来表示就是：欠拟合时候，偏差比较大；而过拟合时，偏差较小但方差较大。

降低过拟合和欠拟合的方法

降低过拟合的方法：

1. 特征 - 减少不必要的特征
    1) 根据特征的重要性，直接删除稀疏特征；
    2) 通过收集更多的数据，或者用数据增广的方法，产生更多的训练数据；从而阻止模型学习不相关的特征。
2. 模型复杂度 - 降低模型复杂度
    1) 神经网络，减少网络层数和神经元个数
    2) 决策树模型中降低树的深度，进行剪枝
3. 正则化 - 加入正则化项并提高正则化项的系数
    1) 对复杂模型和系数比较大的模型进行惩罚，使得算法倾向于训练简单的模型。
4. 多模型决策
    1) 采用 Bagging 或者 Stacking 的集成方法；将多个模型融合起来共同决策；以减少模型预测的 variance.
5. 模型训练
    1) 训练模型时采用早停策略或采用知识蒸馏方法进行训练
6. 数据目标 - 平滑目标
    1) 比如用于分类任务的标签平滑方法，即在 One-hot 表示的 ground true 标签里面，将值为 1 那一位上的一小部分值减掉，均分到其他值为 0 的位值上。

降低欠拟合的方法：

1. 特征 - 添加新特征

1）比如上下文特征,ID 类特征，组合特征等等

2. 模型复杂度 - 增加模型复杂度

1）比如在线性模型中添加高次项;

2）在神经网络模型中增加网络层数或者神经元个数。

3. 正则化 - 减少正则化项的系数

L1 和 L2 正则先验分别服从什么分布

L1 正则先验分布是 Laplace 分布，L2 正则先验分布是 Gaussian 分布。

Laplace 分布公式为：

$$f(x|\mu,\delta)=\frac{1}{2\delta}\exp\left(-\frac{|x-\mu|}{\delta}\right)$$

Gaussian 分布公式为：

$$f(x|\mu,\delta)=\frac{1}{\sqrt{2\pi}\delta}\exp\left(-\frac{(x-\mu)^2}{2\delta^2}\right)$$

接下来从最大后验概率的角度进行推导和分析。在机器学习建模中，我们知道了X和y以后,需要对参数进行建模。那么后验概率表达式如下。

$$P=\log P(y|X,w)\,P(w)=logP(y|X,w)+\log\ P(w)$$

可以看出来后验概率函数是在似然函数的基础上增加了$\log\ P(w)$，$P(w)$的意义是对权重系数w的概率分布的先验假设，在收集到训练样本X，y后，则可根据w在X，y下的后验概率对w进行修正，从而做出对w的更好地估计。若假设的w先验分布为 0 均值的高斯分布，即

$$P(w)=\frac{1}{\sqrt{2\pi}\delta}\exp\left(-\frac{w^2}{2\delta^2}\right)$$

则有

$$\log P(w)=log\prod_j P(w_j)=log\sum_j\left[\frac{1}{\sqrt{2\pi}\delta}\exp\left(-\frac{w_j^2}{2\delta^2}\right)\right]$$

$$=-\frac{1}{2\delta^2}\sum_j w_j^2+C$$

可以看到，在高斯分布下的效果等价于在代价函数中增加 L2 正则项。若假设服从均值为 0，参数为δ的拉普拉斯分布，即

$$P(w_j) = \frac{1}{2\delta}\exp\left(-\frac{|w|}{\delta}\right)$$

则有

$$\log P(w) = log\prod_j P(w_j) = log\sum_j\left[\frac{1}{2\delta}\exp\left(-\frac{|w_j|}{\delta}\right)\right]$$

$$= -\frac{1}{\delta}\sum_j|w_j| + C$$

可以看到，在拉普拉斯分布下的效果等价于在代价函数中增加 L1 正项。

机器学习模型

线性回归与逻辑回归

逻辑回归相比线性回归，有何异同

逻辑回归和线性回归之间既有区别又有联系。逻辑回归和线性回归最大的不同点是逻辑回归解决的是分类问题而线性回归则解决的是回归问题。逻辑回归又可以认为是广义线性回归的一种特殊形式，其特殊之处在于其目标(label/target)的取值服从二元分布。

所谓逻辑回归是一种特殊的广义线性回归，我们可以通过狭义线性回归到逻辑回归的转化过程来理解。狭义线性回归的表达式可表示为：

$$y = w * x + b$$

如果我们希望这个模型可以对二分类任务做出预测，即目标满足 0,1 分布。那么希望预测出来的值经过某种转换之后，大部分可以分布在 0,1 两个值附近。

我们发现 sigmoid 函数可以帮助我们做这样的转换，sigmoid 函数的表达式为：

$$\delta(z) = \frac{1}{1 + e^{-z}}$$

令：

$$y = \delta(z)$$

则：

$$\log\frac{y}{1-y} = z = w * x + b$$

可以看到，在线性回归框架中，狭义线性回归采用y作为预测结果，而逻辑回归则采用$\log\frac{y}{1-y}$作为预测结果。逻辑回归还可以表示为：

$$y = sigmoid(w * x + b)$$

通过以上的步骤推演，我们知道逻辑回归的求值计算其实就是在线性回归的基础上，再做一个 sigmoid 计算。所以它们都可以用相同的方法比如梯度下降来求解参数。

回归问题常用的性能度量指标

注：将误差归为以下三个类别是为了帮助记忆和理解，在学术研究中并没有这样公认的分类。

点对点误差

1. MSE (Mean Square Error) **均方误差** - 该统计参数是预测数据和原始数据对应误差的平方和的均值

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

2. RMSE (Root Mean Square Error) - 观测值与真值偏差的平方和与观测次数 n 比值的平方根，用来衡量观测值同真值之间的偏差

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2}$$

3. MAE (Mean Absolute Error) - 计算模型输出与真实值之间的平均绝对误差

$$MAE = \frac{1}{n}\sum_{i=0}^{n}|y_i - \hat{y_i}|$$

带归一化的误差求解方法

4. MAPE (Mean Absolute Percentage Error) - 不仅考虑预测值与真实值误差，还考虑误差与真实值之间的比例

$$MAPE = \frac{1}{n}\sum_{i=0}^{n}\frac{|y_i - \hat{y_i}|}{y_i}$$

5. MSPE (Mean Squared Percentage Error) - 平均平方百分比误差

$$MSPE = \frac{1}{n}\sum_{i=0}^{n}\left(\frac{y_i - \hat{y_i}}{y_i}\right)^2$$

点对面误差

6. R-Square **决定系数(*coeffcient of determination*)**

$$R-squared=1-\frac{RSS}{TSS}$$

其中 RSS (residual sum of squares)的表达式为：

$$RSS=\sum_{i}^{n}(y_i-\hat{y_i})^2$$

TSS (total sum of squares) 的表达式为：

$$TSS=\sum_{i=1}^{n}(y_i-\bar{y})^2$$

常见分类问题的度量指标

分类问题度量指标的基础是混淆矩阵。

	预测类别			
		Yes	No	总计
实际类别	Yes	TP	FN	P（实际为Yes）
	No	FP	TN	N（实际为No）
	总计	P'（被分为Yes）	N'（被分为No）	P+N

准确率 - 所有预测正确的样本（正样本预测为正，负样本预测为负）与所有样本的比值。

$$Accuracy=\frac{TP+TN}{TP+FN+FP+TN}$$

精确率 - **精确率**是针对我们**预测结果**而言的，它表示的是预测为正的样本中有多少是真正的正样本。那么预测为正就有两种可能了，一种就是把正类预测为正类(TP)，另一种就是把负类预测为正类(FP)，也就是

$$Precision=\frac{TP}{TP+FP}$$

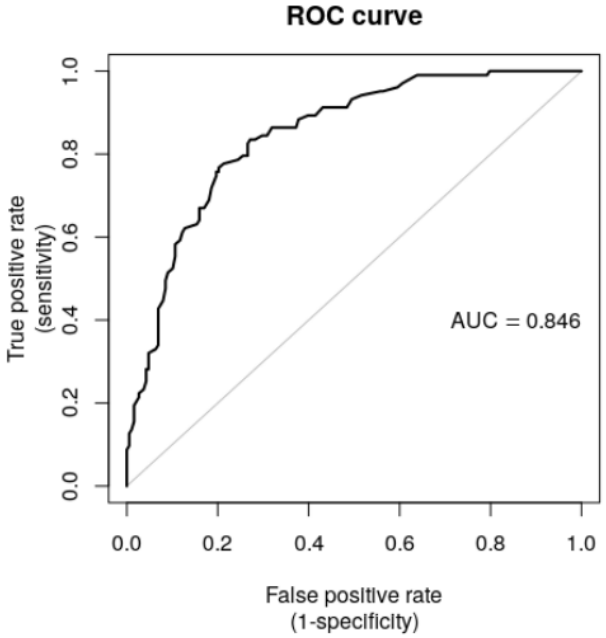
召回率-**召回率**是针对我们原来的样本而言的，它表示的是样本中的正例有多少被预测正确了。那也有两种可能，一种是把原来的正类预测成正类(TP)，另一种就是把原来的正类预测为负类(FN)。

$$Recall = \frac{TP}{TP + FN}$$

F1 值-**F1 值**是精确率和召回率的调和值

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

AUC (Area Under Curve)，曲线下的面积。这里的 Curve 指的的是 ROC (receiver operating characteristic curve) 接收者操作特征曲线，是反映敏感性(sensitivity)和特异性(1-specificity)连续变量的综合指标，ROC 曲线上每个点反映着对同一信号刺激的感受性。ROC 曲线是通过取不同的阈值来分别计算在每个阈值下，伪正类率 FPR（False Positive Rate）和真正类率 TPR （True Positive Rate）的值来绘制的。



分类任务的极大似然损失函数 - 参见题目（**逻辑回归的损失函数？**）

逻辑回归的损失函数

逻辑回归的预测结果服从 0-1 分布，假设预测为 1 的概率为 p，则

$$P(Y = 1|x) = p(x), P(Y = 0|x) = 1 - p(x)$$

写出似然函数来为：

$$L(w)=\prod_{i=1}^{n}p(x_i)^{y_i}(1-p(x_i))^{1-y_i}$$

其中n为样本数，y_i是 label 取值为 0 或 1。对上面的似然函数两边求对数可得:

$$logL(w)=\sum_{i=1}^{n}[y_i logp(x_i)+(1-y_i)\log(1-p(x_i))]$$

极大似然估计方法就是要求函数参数使得$logL(w)$最大，所以我们也可以将目标定为求函数参数使得$-logL(w)$最小。取平均之后，就得到最终的目标函数为：

$$J(w)=-\frac{1}{n}logL(w)=-\frac{1}{n}\sum_{i=1}^{n}[y_i logp(x_i)+(1-y_i)\log(1-p(x_i))]$$

逻辑回归处理多标签分类问题时，一般怎么做

如果y不是在$[0,1]$中取值，而是在 K 个类别中取值，那么这时问题就变为一个多分类问题。有两种方式可以处理该类问题：

1. 当 K 个类别不是互斥的时候，即每次对样本进行分类时，不需要考虑它是不是还可能是别的类别；那么我们可以为每个类别建立一个逻辑回归模型。用它来判断样本是否属于当前类别。
2. 当 K 个类别是互斥的时候，即当 $y=i$ 的时候意味着 y 不能取其他的值，这种情况下 Softmax 回归更合适一些。Softmax 回归是直接对逻辑回归在多分类的推广，相应的模型也可以叫做多元逻辑回归（Multinomial Logistic Regression）。模型通过 Softmax 函数来对概率建模，具体形式如下：

$$P=(y=i|x,\theta)=\frac{e^{\theta_i x}}{\sum_j^K e^{\theta_j x}}$$

决策函数为：

$$y^*=argmax_i P(y=i|x,\theta)$$

对应的损失函数为：

$$J(\theta)=-\frac{1}{N}\sum_{i}^{N}\sum_{j}^{K}1(y_i=j)log\frac{e^{\theta_i x}}{\sum_k^K e^{\theta_k x}}$$

朴素贝叶斯

写出全概率公式&贝叶斯公式

全概率公式

$$P(A) = \sum_{n} P(A|B_n)P(B_n)$$

贝叶斯公式

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

朴素贝叶斯为什么“朴素 naive”

朴素贝叶斯，(Navie Bayesian)中的朴素可以理解为是“简单、理想化”的意思，因为“朴素”是假设了样本特征之间是相互独立、没有相关关系。这个假设在现实世界中是很不真实的，属性之间并不是都是互相独立的，有些属性也会存在相关性，所以说朴素贝叶斯是一种很“朴素”的算法。

朴素贝叶斯的工作流程是怎样的

朴素贝叶斯的工作流程可以分为三个阶段进行，分别是准备阶段、分类器训练阶段和应用阶段。

准备阶段：这个阶段的任务是为朴素贝叶斯分类做必要的准备，主要工作是根据具体情况确定特征属性，并对每个特征属性进行适当划分，去除高度相关性的属性(如果两个属性具有高度相关性的话，那么该属性将会在模型中发挥了 2 次作用，会使得朴素贝叶斯所预测的结果向该属性所希望的方向偏离，导致分类出现偏差)，然后由人工对一部分待分类项进行分类，形成训练样本集合。这一阶段的输入是所有待分类数据，

输出是特征属性和训练样本。(这一阶段是整个朴素贝叶斯分类中唯一需要人工完成的阶段，其质量对整个过程将有重要影响。)

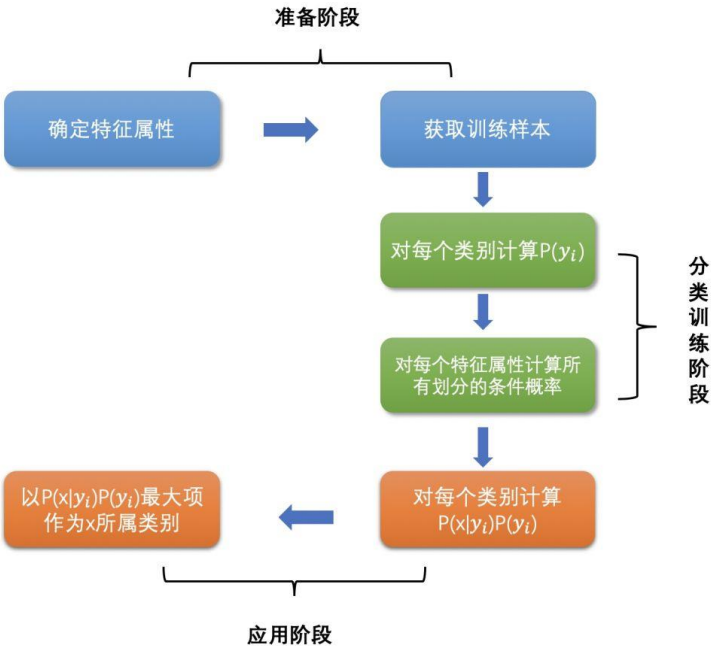
分类器训练阶段：这个阶段的任务就是生成分类器，主要工作是计算每个类别在训练样本中的出现频率及每个特征属性划分对每个类别的条件概率估计，并将结果记录。其输入是特征属性和训练样本，输出是分类器。从公式上理解，朴素贝叶斯分类器模型的训练目的就是要计算一个后验概率$P(c|x)$使得在给定特征的情况下，模型可以估计出每个类别出现的概率情况。

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)} = \frac{P(c)}{P(x)}\prod_{i=1}^{d} P(x_i|c)$$

因为$P(x)$是一个先验概率，它对所有类别来说是相同的；而我们在预测的时候会比较每个类别相对的概率情况，选取最大的那个作为输出值。所以我们可以不计算$P(x)$。贝叶斯学习的过程就是要根据训练数据统计计算先验概率$P(c)$和后验概率$P(x_i|c)$。

应用阶段：这个阶段的任务是使用分类器对待分类项进行分类，其输入是分类器和待分类项，输出是待分类项与类别的映射关系。并选出概率值最高所对应的类别；用公式表示即为：

$$h(x) = argmax_{c\in y} P(c)\prod_{i=1}^{d} P(x_i|c)$$



朴素贝叶斯有没有超参数可以调

基础朴素贝叶斯模型的训练过程，本质上是通过数学统计方法从训练数据中统计先验概率$P(c)$和后验概率$P(x_i|c)$；而这个过程是不需要超参数调节的。所以朴素贝叶斯模型没有可调节的超参数。虽然在实际应用中朴素贝叶斯会与拉普拉斯平滑修正（Laplacian Smoothing Correction）一起使用，而拉普拉斯平滑修正方法中有平滑系数这一超参数，但是这并不属于朴素贝叶斯模型本身的范畴。

朴素贝叶斯对异常值敏不敏感

基础的朴素贝叶斯模型的训练过程，本质上是通过数学统计方法从训练数据中统计先验概率$P(c)$和后验概率$P(x_i|c)$；少数的异常值，不会对统计结果造成比较大的影响。所以朴素贝叶斯模型对异常值不敏感。

K-Means

简述 Kmeans 流程

Kmeans 算法的基本流程如下：

1. 随机初始化 k 个中心点;
2. 计算所有样本分别到 k 个中心点的距离；
3. 比较每个样本到 k 个中心点的距离，并将样本分类到距离最近的中心点所在的类别中；
4. k 个类别组成的样本点重新计算中心点；
5. 重复 2-4，直到中心点不再变化。

其中，k 表示预设的类别个数；两点的距离由欧式距离的平方表示；中心点的计算方式为：在表示中心点向量的每一个方向上计算当前类所有样本的均值。

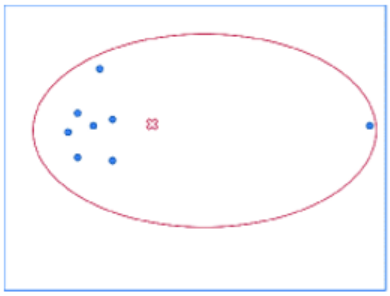
由于 Kmeans 的初始化中心点位置是随机的，而在 Kmeans 的优化过程中，每个中心点只在较小的距离范围内更新，所以算法的效果依赖于中心点初始化的效果。对于此问题的一个改进策略在 Kmeans++算法中提出。

Kmeans++ 的算法思想是使得**初始化的聚类中心点之间的距离尽可能地远**，目标是对 Kmeans 算法的初始化进行优化。其具体流程为:

1. 随机初始化一个中心
2. 对于每个样本 x，计算距离它最近的中心点的欧式距离 D(x)，每个样本被选为中心点的概率为 $\frac{D(x_i)^2}{\sum_{i=1}^{n} D(x_i)^2}$。按照轮盘赌选择法（roulette wheel selection）选择出下一个中心点；
3. 重复步骤 2，直到选出所有的中心点。

Kmeans 对异常值是否敏感,为何

Kmeans 对异常值较为敏感，因为一个集合内元素的均值易受到一个极大值的影响。如图中展示的结果，因为有了一个异常值，这个元素集合的中心点严重偏离了大多数点所在的位置区域；因此在有异常值影响的情况下，均值所计算出来的中心位置不能够反映真实的类别中心。



如何评估聚类效果

聚类往往不像分类一样有一个最优化目标和学习过程，而是一个统计方法，将相似的数据和不相似的数据分开。所以，评估聚类效果可以从以下维度入手：

聚类趋势(对数据进行评估)

霍普金斯统计量（Hopkins Statistic）评估给定数据集是否存在有意义的可聚类的非随机结构。如果一个数据集是由随机、均匀的点生成的，虽然也可以产生聚类结果，但该结果没有意义。聚类的前提需要数据是**非均匀分布**的。该值在区间[0, 1]之间，[0.01, 0.3]表示数据接近随机分布，该值为 0.5 时数据是均匀分布的，[0.7, 0.99]表示聚类趋势很强。

判定聚类的簇数是否为最佳

可用业务分析法，观察法，手肘法和 Gap Statistic 方法找到最佳的分类数,然后将这个分类树与实际簇数做比较 （见**超参数 k 如何选择？**）

聚类质量

因为没有标签，所以一般通过评估类的分离情况来决定聚类质量。类内部的样本距离越小，类之间的距离越大，则表示聚类效果越好。

超参数 k 如何选择

- **根据业务**，比如业务需要把用户分成高中低三种，则 k 选择为 3；
- **观察法**，对于低纬度数据适用；对数据进行可视化或者 PCA 降维可视化之后，可大致观测出数据分布自然区分的类别数；
- **手肘法**
  所有样本点到它所存在的聚类中心点的距离之和，作为模型好坏的衡量标准。
  具体步骤为：

- 计算$D_k = \sum_{i=1}^{k}\sum_{X\in C_i}\|X - M_i\|^2$（$k$越大$D_k$会越小）；
- 绘制D_k关于k的函数图像；
- 观察是否存在某个值，当k大于这个值之后，D_k的减小速度明显变缓或者基本不变，则这个值所在的点即为拐点，拐点处的 k 值就是要选择的最佳分类个数。

- Gap Statistic

  Gap Statistic 计算公式为：

  $$Gap(K) = E\left(log\widehat{D}_k\right) - logD_k$$

  采用均匀分布模型抽样产生和原始样本一样多的随机样本，对随机样本做 Kmeans 得到$\widehat{D}_k$，重复多次可以获得$E\left(log\widehat{D}_k\right)$，Gap statistic 取最大值所对应的 k 就是最佳的 k。Gap Statistic 方法借鉴了蒙特卡洛算法的思想。

Kmeans 算法的优缺点

优点

- 容易理解，聚类效果不错，虽然是局部最优， 但往往局部最优就能够给出一个不错的结果；
- 处理大数据集的时候，该算法可以保证较好的伸缩性 - 即稍作改进即可处理大数据集；
- 当数据簇近似高斯分布的时候，效果非常不错；
- 算法复杂度低 - 时间复杂度为$O(n * k * t)$ 其中n为样本数，k为类别数，t为迭代次数；而且通常来说k和t 相对于n来说很小。

缺点

- k 值需要人为设定，不同 k 值得到的结果不一样；
- 对初始的类别中心敏感，不同选取方式会得到不同的结果；
- 对异常值敏感；
- 样本只能归为一类，不适合多类别多标签分类任务；
- 不适合太离散的分类、样本类别不平衡的分类以及同类样本分布呈非凸形状的分类。

SVM

请简述 SVM 原理

SVM 是一种二类分类模型。它的基本模型是在特征空间中寻找间隔最大化的分离超平面的线性分类器。

- 当训练样本线性可分时，通过硬间隔最大化，学习一个线性分类器，即线性可分支持向量机；
- 当训练数据近似线性可分时，引入松弛变量，通过软间隔最大化，学习一个线性分类器，即线性支持向量机；
- 当训练数据线性不可分时，通过使用核技巧及软间隔最大化，学习非线性支持向量机。

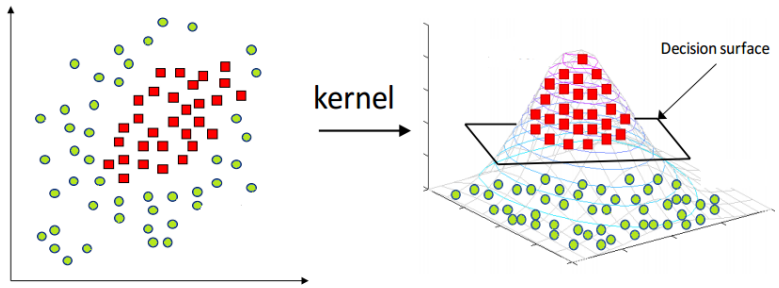
SVM 为什么采用间隔最大化

当训练数据线性可分时，存在无穷多个分离超平面可以将两类数据正确分开。感知机利用误分类最小策略，求得分离超平面，不过此时的解有无穷多个。线性可分支持向量机利用间隔最大化求得最优分离超平面，这时，解是唯一的。另一方面，此时的分隔超平面所产生的分类结果是鲁棒的，对未知实例的泛化能力最强。

SVM 为什么要引入核函数

当样本在原始空间线性不可分时，可将样本从原始空间映射到一个更高维的特征空间，使得样本在这个特征空间内线性可分。而引入这样的映射后，在所要求解的问题中，无需求解真正的映射函数，而只需要知道其核函数。核函数的定义为：$K(x,y) = <\phi(x),\phi(y)>$，即在特征空间的内积等于它们在原始样本空间中通过核函数 K 计算的结果。

SVM 引入核函数之后，一方面数据变成了高维空间中线性可分的数据；另一方面不需要求解具体的映射函数，只需要给定具体的核函数即可，这样使得求解的难度大大降低。



SVM 核函数之间的区别

SVM 常用的核函数有线性核函数，多项式核函数，高斯核（RBF），拉普拉斯核和 Sigmoid 核函数。其中多项式核函数，高斯核（RBF），拉普拉斯核和 Sigmoid 核函数通常用来处理线性不可分的情形。而一般选择核函数时通常考虑线性核和高斯核，也就是线性核与 RBF 核。 线性核：主要用于线性可分的情形，参数少，速度快，对于一般数据，分类效果已经很理想了。 RBF 核：主要用于线性不可分的情形，参数多，分类结果非常依赖于参数。有很多人是通过训练数据的交叉验证来寻找合适的参数，不过这个过程比较耗时。

为什么 SVM 对缺失数据和噪声数据敏感

这里说的缺失数据是指缺失某些特征数据，向量数据不完整。

SVM 没有处理缺失值的策略。而 SVM 希望样本在特征空间中线性可分，所以特征空间的好坏对 SVM 的性能很重要。缺失特征数据将影响训练结果的好坏。

另外，SVM 对噪声数据也较为敏感；原因是 SVM 的决策只基于少量的支持向量，若噪音样本出现在支持向量中，容易对决策造成影响，即影响目标函数中损失项的收敛，所以 SVM 对噪音敏感。

SVM 算法的优缺点

优点：

1. 可以有效解决高维特征的分类和回归问题；
2. 无需依赖全体样本，只依赖支持向量；
3. 有大量的核技巧可以使用，从而可以应对线性不可分问题；
4. 对于样本量中等偏小的情况，仍然有较好的效果。

缺点：

1. 如果特征维度远大于样本个数，SVM 表现一般；
2. SVM 在样本巨大且使用核函数时，计算量很大；
3. 在应对非线性可分数据时，核函数的选择依旧没有统一的标准；
4. SVM 对缺失和噪声数据敏感；
5. 特征的多样性限制了 SVM 的使用，因为 SVM 本质上是属于一个几何模型，这个模型需要用 Kernel 定义样本之间的相似度（线性 SVM 中的内积），而我们无法预先为所有特征设定一个统一的相似度计算方法。这样的数学模型使得 SVM 更适合去处理“同性质”的特征。

SVM 的超参数 C 如何调节

在使用 SVM 库时候，通常有两个需要手工调节的关键参数

1. 参数 C
2. 核函数(Kernel)

由于 C 可以看做与正则化参数 λ 作用相反 （C 作用于损失项上，而 λ 作用于正则化项上，见**简述 SVM 软间隔推导过程** ），则对于 C 的调节：

- 低偏差，高方差，即遇到了过拟合时；减少 C 值。
- 高偏差，低方差，即遇到了欠拟合时：增大 C 值。

SVM 的核函数如何选择

- 当特征维度n较高，而样本规模m较小时，不宜使用核函数或者选用线性核的SVM，否则容易引起过拟合。
- 当特征维度n较低，而样本规模m足够大时，考虑使用高斯核函数。不过在使用高斯核函数前，需要进行特征缩放(feature scaling)。

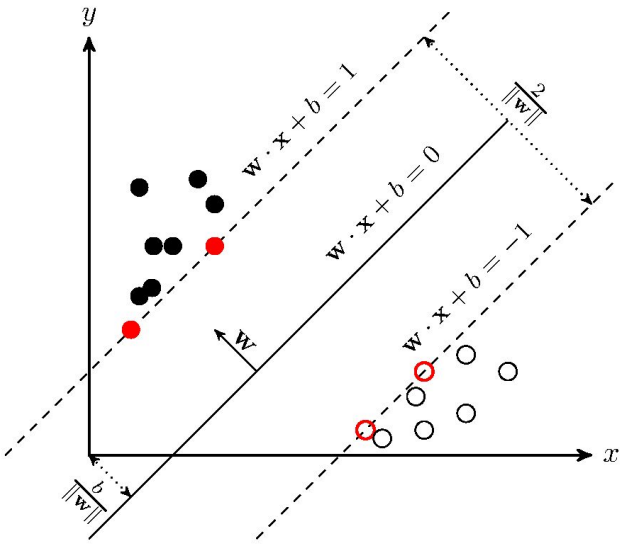
简述 SVM 硬间隔推导过程

假设空间超平面的方程为$w^T x + b = 0, w$为平面法向量，那么点到平面的距离为:

$$r = \frac{|w^T x + b|}{\|w\|}$$

我们令，如图所示：

$$\begin{cases} w^T x_i + b \geq +1, & y_i = +1 \\ w^T x_i + b \leq -1, & y_i = -1 \end{cases}$$



最大化间隔即为：

$$\max_{w,b} \frac{2}{\|w\|}$$

$$s.t. y_i(w^T x_i + b) \geq 1, \quad i = 1, 2, \ldots, N$$

而 $1/\|w\|$最大等价于$\|w\|$最小，所以问题等价于

$$\min_{w,b}\frac{1}{2}\|w\|^2$$

$$s.t.\quad y_i(w^T x_i+b)-1\geq 0,\quad i=1,2,\dots,N$$

定义拉格朗日函数为

$$L(w,b,\alpha)=\frac{1}{2}\|w\|^2+\sum_{i=1}^{N}\alpha_i(1-y_i(w^T x_i+b))$$

所以原问题的对偶问题为

$$\max_{\alpha}\min_{w,b}L(w,b,\alpha)$$

先求min令其对w和b求偏导，导数为 0 得出：

$$w=\sum_{i=1}^{N}\alpha_i y_i x_i$$

$$0=\sum_{i=1}^{N}a_i y_i$$

那么对偶问题变成了：

$$\max_{\alpha}\sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(x_i x_j)$$

$$s.t.\sum_{i=1}^{N}\alpha_i y_i=0$$

$$\alpha_i\geq 0, i=1,2,\dots,N$$

因为我们的原问题中有不等式约束，因此上述的过程满足 KKT 条件，即非拉格朗日参数的偏导数为 0；拉格朗日项为 0；拉格朗日乘子小于 0，拉格朗日参数大于 0，则：

$$\begin{cases}\alpha_i\geq 0\\ y_i f(x_i)-1\geq 0\\ \alpha_i(y_i f(x_i)-1)=0\end{cases}$$

那么对于任意的训练样本总满足$\alpha_i=0$或$y_i f(x_i)=1$，若$\alpha_i=0$,则样本不会被记入$f(x)$的表达式中（根据α_i对w的表达式得出），若$\alpha_i>0$，则必有$y_i f(x_i)=1$，所对应的样本点位于最大间隔边界上，是一个支持向量。**训练完成后，大部分的训练样本都不需要保留，最终模型仅与支持向量有关。**

对偶问题是一个二次规划问题，可以通过序列最小优化算法（Sequential Minimal Optimization, SMO）方法求出，这里不多做讨论。求出最佳的α_i^*之后，根据:

$$b^* = y_i - \sum_{i=1}^{N} \alpha_i^* y_i x_i^T x$$

所以超平面可表示为:

$$\sum_{i=1}^{N} \alpha_i^* y_i x_i^T x + b^* = 0$$

其中α_i^*为支持向量的拉格朗日参数。

我们复盘一下整个过程，可以将推导过程归纳为：

1. 通过点到平面的距离，构建我们要求解的带有约束条件的原始问题；
   将对原始问题的求解经拉格朗日转换后,变成求解它的对偶问题；通过对朗格朗日函数求偏导，并使得导数为 0，把求解的参数简化为一个α；
2. 通过 SMO 来求解对偶问题得到最佳的α；
3. 根据α与w和b的关系，得到最佳超平面的表达式；
4. 根据 KKT 条件，我们过滤掉了大部分的样本，只保留了支持向量来求解最佳超平面。

如果再进一步浓缩，大家可以记忆这个过程为：

1. 构建最大间隔问题，写出超平面表达式；
2. 通过拉格朗日函数，将原问题转化为它的对偶问题，并用 SMO 方法求解表达式的参数；
3. 采用 KKT 条件过滤样本，求出最终超平面表达式。

简述 SVM 软间隔推导过程

如果样本近似线性可分，允许少数分类错误，那么可以使用软间隔的 SVM 为每个样本引入一个松弛向量，使得样本可以在两个间隔平面中间，即

$$y_i(w^T x_i + b) \geq 1 - \varepsilon_i$$

那么目标函数变为了：

$$\min_{w,b,\varepsilon}\frac{1}{2}\|w\|^2+C\sum_{i=1}^{N}\varepsilon_i$$

$$s.t.\quad y_i(w^Tx_i+b)\geq 1-\varepsilon_i,\quad i=1,2,\dots,N$$

$$\varepsilon_i\geq 0, i=1,2,\dots,N$$

其中 C 为惩罚系数，平衡间隔最大和误差分类个数最少，其拉格朗日函数为：

$$L(w,b,\varepsilon,\alpha,\mu)=\frac{1}{2}\|w\|^2+C\sum_{i=1}^{N}\varepsilon_i-\sum_{i=1}^{N}\alpha_i(y_i(w^T\,x_i+b)-1+\varepsilon_i)-\sum_{i=1}^{N}\mu_i\varepsilon_i$$

同硬间隔问题类似，对w,b,ε 求偏导，使得导数为 0 可得：

$$w=\sum_{i=1}^{N}\alpha_i y_i x_i$$

$$0=\sum_{i=1}^{N}a_i y_i$$

$$C=\alpha_i+\mu_i$$

将结果带回到 L, 则得到原问题的对偶问题为：

$$\max_{\alpha}\sum_{i=1}^{N}\alpha_i-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j\left(x_i x_j\right)$$

$$s.t.\sum_{i=1}^{N}\alpha_i y_i=0$$

$$C-\alpha_i-\mu_i=0$$

$$\alpha_i\geq 0$$

$$\mu_i\geq 0, i=1,2,\dots,N$$

进一步得到，软间隔的 KKT 条件为：

$$\begin{cases}\alpha_i\geq 0,\mu_i\geq 0\\ y_i f(x_i)-1+\varepsilon_i\geq 0\\ \alpha_i(y_i f(x_i)-1+\varepsilon_i)=0\\ \varepsilon_i\geq 0\\ \mu_i\varepsilon_i=0\end{cases}$$

对于任意样本，总有$\alpha_i = 0$或$y_i f(x_i) = 1 - \varepsilon_i$，若$\alpha_i = 0$,则样本不会被记入$f(x)$的表达式中，若$\alpha_i > 0$，则必有$y_i f(x_i) = 1 - \varepsilon_i$，所对应的是一个支持向量；若$\alpha_i < C$,则$\mu_i > 0$,进而有$\varepsilon_i = 0$，即该样本落在最大间隔边界上；若$\alpha_i = C$，则有$\mu_i = 0$，此时若$\varepsilon_i \leq 1$，则样本落在最大间隔内部，若$\varepsilon_i \geq 1$则样本被错误分类。

树模型

简述决策树的构建过程

- 构建根节点，将所有训练数据都放在根节点；
- 选择一个最优特征，按照这一特征将训练数据集分割成子集，使得各个子集在当前条件下获得最好的分类；
- 如果子集非空，或未达到停机条件，递归 1，2 步骤，直到所有训练数据子集都被正确分类或没有合适的特征为止。

ID3 决策树与 C4.5 决策树的区别

按照题目**简述决策树的构建过程**中所描述的，构建决策树过程中一个关键步骤就是选择一个最优特征；而 ID3 决策树与 C4.5 决策树一个最大的区别就是在选择最优特征时所依赖的标准不同。

在 ID3 决策树中，选择最佳特征通过信息增益指标来选择，所谓信息增益可以定义为：

数据集对于某特征的信息增益 = 数据集的经验熵 - 这个特征对数据集的条件经验熵

$$g(D,A) = H(D) - H(D|A)$$

其中 D 为数据集，A 表示数据集样本上的某个特征，H（D）为数据集的经验熵；H（D|A）表示特征 A 对数据集 D 的经验条件熵。

$$H(D) = -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|}$$

其中 K 表示 D 中类别的数量，C_k表示第 k 个类别所包含样本的个数。

$$H(D|A) = -\sum_{i=1}^{n} \frac{|D_i|}{|D|} \sum_{k=1}^{K} \frac{|D_{ik}|}{|D|} \log_2 \frac{|D_{ik}|}{|D|}$$

其中 n 表示在特征 A 上不同取值的个数，D_{ik}表示在特征上取第 i 个值且类别为 k 时的样本个数。

在 C4.5 决策树中，为了解决信息增益偏向于选择取值较多的特征的问题，选择最佳特征通过信息增益熵来选择，所谓信息增益熵可以定义为：

$$g_R(D,A)=\frac{g(D,A)}{H_A(D)}$$

其中$H_A(D)$表示，训练数据集 D 关于特征 A 的值的熵。

$$H_A(D)=-\sum_{i=1}^{n}\frac{|D_i|}{|D|}\log_2\frac{|D_i|}{|D|}$$

其中n表示特征 A 上取得不同值的个数，D_i表示在特征 A 上取值为第i个时，样本的个数。

此外 1）ID3 只能处理离散型变量，而 C4.5 通过将连续值离散化来处理连续型变量。2）ID3 没有对缺失值的处理策略，而 C4.5 通过引入无缺失值样本对所有样本的比例来处理缺失值带来的“信息增益失真”的问题。

CART 回归树构建过程

在 CART 树的构建过程中，假设决策树是一颗二叉树；决策树的生成过程就是递归地构建二叉决策树的过程。对回归树来说用，我们这里讨论一个基本形式即为平方误差最小化准则，进行特征选择，生成二叉树。

输入：训练数据集 D；

输出：回归树 f(x)

在训练数据集所在的输入空间中，递归地将每个区域划分为两个子区域并决定每个子区域上的输出值，构建二叉树：

1）选择最优切分变量（特征）j 与切分点（特征值域上的值）s,求解

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_2\in R_2(j,s)}(y_i-c_2)^2]$$

2）用选定的(j,s)划分区域并决定相应的输出值，其中$R_1(j,s)$是划分后的左子区域，而$R_2(j,s)$是划分后的右子区域；c_1是左子区域上的预测值(此区域上所有样本真实值的均值)，而c_2是右子区域上的预测值(此区域上所有样本真实值的均值)。

$$R_1(j,s)=\{x|x^{(j)}\le s\},R_2(j,s)=\{x|x^{(j)}>s\}$$

$$\hat{c}_m=\frac{1}{N_m}\sum_{x_i\in R_m(j,s)}y_i,x\in R_m,m=1,2$$

3）递归地对两个子区域调用步骤 1），2）,直到满足停机条件。常用的停机条件有 a.树的深度 b.叶子区域的个数 c.叶子区域上样本的个数等。

4）将输入空间划分为 M 个区域 R1,R2，…,RM,生成决策树，划分的空间即为叶子节点上的空间：

$$f(x)=\sum_{m=1}^{M}\hat{c}_mI(x\in R_m)$$

大家可以对照 XGBoost 节点分裂的过程来学习 CART 回归树的生成过程。

决策树的优缺点

优点：

- 缺失值不敏感，对特征的宽容程度高 - 可缺失可连续可离散；
- 可以计算特征重要性，且可解释性强；
- 算法对数据没有强假设；
- 可以解决线性及非线性问题；
- 有特征选择等辅助功能。

缺点：

- 处理关联性数据比较薄弱 - 重要性强且关联的特征都得到了重视；
- 正负量级有偏样本的样本效果较差；
- 单棵树的拟合效果欠佳，容易过拟合。

决策树如何防止过拟合,说说具体方法

我们在讨论防止机器学习过拟合的时候，通过分类的方法，大致确立了这么几个改进方向，1）数据 2）模型 3）正则化 4）训练。在这个题目中，我们重点讨论如何通过改进决策树模型来防止过拟合，当然其他几个方向对防止决策树过拟合同样适用。

通过改进模型来防止过拟合的主要思路是简化模型，使得模型能够学习样本中的共同特征（即主要特征），而摈弃个性化的特征（即次要特征）。而对树模型进行简化的方法又可以分为预剪枝（在训练过程中进行剪枝），和后剪枝（在决策树构建完成之后进行剪枝）

预剪枝的主要方法有：

1） 限制树的深度 - 当树到达一定深度的时候，停止树的生长
2） 限制叶子节点的数量
3） 规定叶子区域内最少的样本数，未达到最少样本数的叶子区域不做分裂
4） 计算每次分裂对**测试集**的准确度提升

后剪枝的核心思想是让算法生成一颗完全生长的决策树，然后最底层向上计算是否剪枝。剪枝过程将子树删除，用一个叶子节点替代，该节点的类别同样按照多数投票的原则进行判断。同样地，后剪枝也可以通过在测试集上的准确率进行判断，如果剪枝过后准确率有所提升，则进行剪枝。相比于预剪枝，后剪枝方法通常可以得到泛化能力更强的决策树，但时间开销会更大。

集成学习

Bagging & Boosting

什么是集成学习算法

集成学习(Ensemble Learning)就是将多个机器学习模型组合起来，共同工作以达到优化算法的目的。集成学习的一般步骤是：1.生产一组 “个体学习器”(Individual leaner);2.用某种策略将它们结合起来。

个体学习器通常由一个学习算法通过训练数据训练产生。在同质集成（系统中个体学习器的类型相同）中，个体学习器又被称为“基学习器”而在异质集成（系统中个体学习的类型不同）中，个体学习器又被称为“组件学习器”(component learner)。

集成学习的思想类似我们俗话常说的“三个臭皮匠胜过一个诸葛亮”。

集成学习主要有哪几种框架,分别简述这几种集成学习框架的工作过程

集成学习主要的集成框架有，Bagging，Boosting 和 Stacking。其中 Bagging 和 Boosting 为同质集成，而 Stacking 为异质集成。

- **Bagging （Bootstrap Aggregating）**；Bagging 的核心思想为**并行地训练一系列各自独立的同类模型**，然后再将各个模型的输出结果按照某种策略进行聚合（例如分类中可采用投票策略，回归中可采用平均策略）。Bagging 方法主要分为两个阶段：
    - Bootstrap 阶段，即采用有放回的抽样方法，将训练集分为 n 个子样本集；并用基学习器对每组样本分别进行训练，得到 n 个基模型。
    - Aggregating 阶段，将上一阶段训练得到的 n 个基模型组合起来，共同做决策。在分类任务中，可采用投票法。比如相对多数投票法，即将结果预测为得票最多的类别。而在回归任务中可采用平均法，即将每个基模型预测得到的结果进行简单平均或者加权平均来获得最终的预测结果。

- **Stacking**：Stacking 的核心思想为**并行地训练一系列各自独立的不同类模型，**然后通过训练一个元模型（meta-model）来将各个模型的输出结果进行结合。也可以用两个阶段来描述 Stacking 算法：
    - 第一阶段，分别采用全部训练样本训练 n 个组件模型，要求这些个体学习器必须是异构的，也就是说采用的学习方法不同；比如可以分别是线性学习器，SVM，决策树模型和深度模型。
    - 第二阶段，训练一个元模型（meta-model）来将各个组件模型的输出结果进行结合。具体过程是，将各个学习器在训练集上得到的预测结果作为训练特征和训练集的真实结果组成新的训练集；用这个新组成的训练集来训练一个元模型。这个元模型可以是线性模型或者树模型。
- **Boosting**：Boosting 的核心思想为**串行地训练一系列前后依赖的同类模型，**即后一个模型用来对前一个模型的输出结果进行纠正。Boosting 算法是可将弱学习器提升为强学习的算法。学习过程是：先从初始训练集训练出一个基学习器，再根据基学习器的表现对训练样本进行调整，使得先前基学习器做错的训练样本在后续训练中受到更多关注，然后将当前基学习器集成到集成学习器中并基于调整后的样本分布来训练下一个基学习器；如此重复进行，直至基学习器数目达到事先指定的值 T。

Boosting 算法有哪两类，它们之间的区别是什么

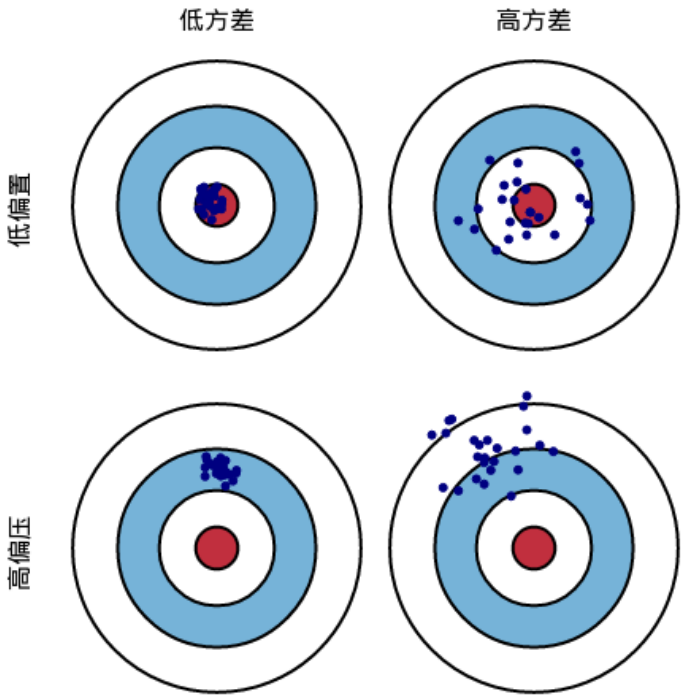
Boosting 算法主要有 AdaBoost（Adaptive Boost）自适应提升算法和 Gradient Boosting 梯度提升算法。最主要的区别在于两者如何识别和解决模型的问题。AdaBoost 用分错的数据样本来识别问题，通过调整分错的数据样本的权重来改进模型。Gradient Boosting 通过负梯度来识别问题，通过计算负梯度来改进模型。

什么是偏差和方差

偏差指的是预测值的期望与真实值之间的差距，偏差越大，预测值越偏离真实数据的标签。

方差描述的是预测值的变化范围，离散程度，也就是离预测值期望的距离，方差越大，数据的分布越分散。

可通过打靶射击的例子来做类比理解，我们假设一次射击就是一个机器学习模型对一个样本进行预测，射中红色靶心位置代表预测准确，偏离靶心越远代表预测误差越大。偏差则是衡量射击的蓝点离红圈的远近，射击位置即蓝点离红色靶心越近则偏差越小，蓝点离红色靶心越远则偏差越大；方差衡量的是射击是否稳定，即射击的位置蓝点是否聚集，蓝点越集中则方差越小，蓝点越分散则方差越大。



为什么说 Bagging 可以减少弱分类器的方差，而 Boosting 可以减少弱分类器的偏差

Bagging 算法对数据重采样，然后在每个样本集训练出来的模型上取平均值。假设有 n 个随机变量，方差记为δ^2，两两变量之间的相关性是$0 < \rho < 1$，则n个随机变量均值的方差为：

$$var\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{\delta^2}{n} + \frac{n-1}{n}\rho\delta^2$$

上式中随着n增大，第一项趋于0，第二项趋于$\rho\delta^2$，所以Bagging能够降低整体方差。在随机变量完全独立的情况下，n个随机变量的方差为$\frac{\delta^2}{n}$，n个随机变量的方差是原来的$1/n$。

Bagging 算法对 n 个独立不相关的模型的预测结果取平均，方差可以得到减少，如果模型之间相互独立，则集成后模型的方差可以降为原来的$\frac{1}{n}$，但是在现实情况下，模型不可能完全独立。为了追求模型的独立性，Bagging 的方法做了不同的改进；比如随机森林算法每次选取节点分裂属性时，会随机抽取一个属性子集，而不是从所有的属性中选取最优属性，这就为了避免弱分类器之间过强的关联性；通过训练集的重采样也能够带来弱分类器之间的一定独立性。这样多个模型学习数据，不会因为一个模型学习到数据某个特殊特性而造成方差过高。

设单模型的期望为μ，则 Bagging 的期望预测为：

$$E\left(\frac{1}{n}\sum_{i=1}^{n}X_i\right)=\frac{1}{n}E\left(\sum_{i=1}^{n}X_i\right)=E(X_i)\approx\mu$$

说明 Bagging 整体模型的期望近似于单模型的期望，这意味整体模型的偏差也与单模型的偏差近似。所以 Bagging 不能减少偏差。

在 Boosting 算法训练过程中，我们计算弱分类器的错误和残差，作为下一个分类器学习目标调整的依据；这个过程本身就在不断减小损失函数的值，其偏差自然逐步下降。但由于是采取这种串行和自适应的策略，各子模型之间是强相关的，于是子模型之和并不能显著降低方差。所以说 Boosting 主要还是靠降低偏差来提升模型性能。

随机森林

简述一下随机森林算法的原理

随机森林算法是 Bagging 集成框架下的一种算法，它同时对训练数据和特征采用随机抽样的方法来构建更加多样化的基模型。随机森林具体的算法步骤如下：

1. 假设有N个样本，则有放回的随机选择N个样本(每次随机选择一个样本，然后将该样本放回并继续选择)。采用选择好的N个样本用来训练一个决策树，作为决策树根节点处的样本。
2. 假设每个样本有M个属性，在决策树做节点分裂时，随机从这M个属性中选取m个属性，满足条件$m \ll M$。然后采用某种策略（比如信息增益最大化）从m个属性中选择一个最优属性作为该节点的分裂属性。
3. 决策树形成过程中重复步骤 2 来计算和分裂节点。一直到节点不能够再分裂，或者达到设置好的阈值（比如树的深度，叶子节点的数量等）为止。注意整个决策树形成过程中没有进行剪枝。
4. 重复步骤 1~3 建立大量的决策树，这样就构成了随机森林。

随机森林的随机性体现在哪里

随机森林的随机性体现在每颗树的训练样本是随机的，树中每个节点的分裂属性集合也是随机选择确定的。

1. **随机采样**：随机森林在计算每棵树时，从全部训练样本（样本数为 N）中选取一个**可能有重复的、大小同样为 N** 的数据集进行训练（即 Booststrap 采样）。
2. **特征选取的随机性**：在节点分裂计算时，随机地选取**所有特征的一个子集**，用来计算最佳分割方式。

随机森林算法的优缺点

优点

特征和数据的随机抽样

1. 它可以处理很高维度（特征很多）的数据，并且不用降维，无需做特征选择；
2. 如果有很大一部分的特征遗失，仍可以维持准确度；
3. 不容易过拟合；
4. 对于不平衡的数据集来说，它可以平衡误差；
5. 可以判断出不同特征之间的相互影响（类似于控制变量法）；

树模型的特性

6. 较好的解释性和鲁棒性；
7. 能够自动发现特征间的高阶关系；
8. 不需要对数据进行特殊的预处理如归一化；

算法结构

9. 训练速度比较快，容易做成并行方法；
10. 实现起来比较简单。

缺点

1. 随机森林已经被证明在某些噪音较大的分类或回归问题上会过拟合。（决策树的学习本质上进行的是决策节点的分裂，依赖于训练数据的空间分布）
2. 对于有不同取值的属性的数据，取值划分较多的属性会对随机森林产生更大的影响，所以随机森林在这种数据上产出的属性权值是不可信的。

随机森林为什么不能用全样本去训练 m 棵决策树

随机森林的基学习器是同构的，都是决策树，如果用全样本去训练 m 棵决策树的话；基模型之间的多样性减少，互相相关的程度增加，不能够有效起到减少方差的作用；对于模型的泛化能力是有害的。

随机森林和 GBDT 的区别

1. 随机森林采用的 Bagging 思想，而 GBDT 采用的 Boosting 思想。
2. 组成随机森林的树可以并行生成；而 GBDT 只能是串行生成。
3. 随机森林对异常值不敏感；GBDT 对异常值非常敏感。
4. 随机森林对训练集一视同仁；GBDT 对训练集中预测错误的样本给予了更多关注。
5. 随机森林是通过减少模型方差提高性能；GBDT 是通过减少模型偏差提高性能。
6. 对于最终的输出结果而言，随机森林采用多数投票等方法；而 GBDT 则是将所有结果累加起来，或者加权累加起来。
7. 组成随机森林的树可以是分类树，也可以是回归树；而 GBDT 只能由回归树组成。

GBDT

简述 GBDT 原理

梯度提升树的训练过程大致是这样的：

1. 根据训练集训练一颗初始决策树；
2. 计算之前所有树在此数据集上预测结果之和与真实结果的差值，又叫做残差。
3. 把残差作为当前树拟合的目标在训练集上训练。
4. 重复 2,3 步骤，直到达到设置的阈值（树的个数，早停策略等）

采用伪代码表示如下：

输入：训练数据集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$，其中$x_i \in x \subseteq R^n, y_i \in y \subseteq R$;

输出：提升树$f_M(x)$.

1）初始化$f_0(x) = 0$

2）对$m = 1, 2, \dots, M$

a. 计算残差

$$r_{mi} = y_i - f_{m-1}(x_i), \quad i = 1, 2, \dots, N$$

b. 拟合残差r_{mi}学习一个回归树，得到$T(x, \theta_m)$

c. 更新 $f_m(x) = f_{m-1}(x) + T(x, \theta_m)$

3）得到回归问题的提升树

$$f_M(x) = f_m(x) = \sum_{m=1}^{M} T(x, \theta_m)$$

上述的伪代码描述中，我们对 GBDT 算法做了以下三个简化：

1. 用残差来表示提升树的负梯度；
2. 假设所有树的贡献权重都相同；
3. 没有把回归树在叶子节点上的拟合信息体现出来。

接下来我们将简化的信息补全，得到下面 GBDT 算法的伪代码：

1）通过最小化损失函数优化初始模型：

$$F_0(x) = \arg\min_r(\sum_{n=1}^{N} L(y_i, \gamma))$$

2）对$m = 1, 2, \ldots, M$

a. 计算负梯度：

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1(x)}}, i = 1, 2, \ldots, n$$

b. 训练一个回归树去拟合目标值r_{im}，树的终端区域为$R_{jm}(j = 1, 2, \ldots, J_m)$

c. 对$j = 1, 2, \ldots, J_m$，计算步长γ_{jm}

$$\gamma_{jm} = \arg\min_{\gamma}(\sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma))$$

d. 更新模型:

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$$

3）输出$F_M(x)$.

其中每个树的终端区域代表当前树上叶子节点所包含的区域，步长γ_{jm}为第 m 棵树上第j个叶子节点的预测结果。

GBDT 如何用于分类

GBDT 在做分类任务时与回归任务类似，所不同的是损失函数的形式不同。我们以二分类的指数损失函数为例来说明：

我们定义损失函数为：

$$L(y, f(x)) = \exp(-yf(x))$$

其中$y_i \in y = \{-1, +1\}$，则此负梯度可表示为：

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1(x)}} = y_i \exp(-y_i f(x_i)), i = 1, 2, \ldots, n$$

对于各个叶子节点最佳的拟合值为：

$$\gamma_{jm} = \arg\min_{\gamma}(\sum_{x_i \in R_{jm}} \exp\left(-y_i(F_{m-1}(x_i) + \gamma)\right))$$

注意：通过与 GBDT 伪代码中步骤 b 和步骤 c 类比来理解。

GBDT 常用损失函数有哪些

回归问题：

MAE, MSE, RMSE（见**回归问题常用的性能度量指标？**）

补充

Huber Loss （MAE 和 MSE 结合）

1）在 0 附近可导

2）Huber 大部分情况下为 MAE，只在误差很小的时候变成了 MSE。

$$L_\delta\big(y, f(x)\big) = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\frac{1}{2}\big(y_i - f(x_i)\big)^2, & |y_i - f(x_i)| \le \delta \\ \delta|y_i - f(x_i)| - \frac{1}{2}\delta, & otherwise\end{cases}$$

分类问题：

对数似然损失函数，应对二分类和多分类问题（见**逻辑回归的损失函数？逻辑回归处理多标签分类问题时，一般怎么做**？）

补充

指数损失函数：$y_i \in y = \{-1, +1\}$

$$L\big(y, f(x)\big) = \frac{1}{N}\sum_{i=1}^{N}\exp\left(-y_i f(x_i)\right)$$

为什么 GBDT 不适合使用高维稀疏特征

高维稀疏特征会使树模型的训练变得极为低效，且容易过拟合。以 ID 类型特征为例进行说明：

- 树模型训练过程是一个贪婪选择特征的算法，要从候选特征集合中选择一个使分裂后收益函数增益最大的特征来分裂。按照高维的 ID 特征做分裂时，子树数量非常多，计算量会非常大，训练会非常慢。
- 同时，按 ID 分裂得到的子树的泛化性也比较弱，由于只包含了对应 ID 值的样本，样本稀疏时也很容易过拟合。

GBDT 算法的优缺点

优点：

- 预测阶段的计算速度快，树与树之间可并行化计算（注意预测时候可并行）。
- 在分布稠密的数据集上，泛化能力和表达能力都很好。
- 采用决策树作为弱分类器使得 GBDT 模型具有 1）较好的解释性和鲁棒性，2）能够自动发现特征间的高阶关系，并且也 3）不需要对数据进行特殊的预处理如归一化等。

缺点

- GDBT 在高维稀疏的数据集上，表现不佳（见**为什么 GBDT 不适合使用高维稀疏特征?**）
- 训练过程需要串行训练，只能在决策树内部采用一些局部并行的手段提高训练速度。

XGBoost

简述 XGBoost

XGBoost 是陈天奇新开发的 Boosting 库。它是一个大规模、分布式的通用 Gradient Boosting 库，它在 Gradient Boosting 框架下实现了 GBDT 和一些广义的线性机器学习算法。

在面试时，可以通过与 GBDT 的对比来介绍 XGBoost（见 **XGBoost 和 GBDT 有什么不同？**）

XGBoost 和 GBDT 有什么不同

- GBDT 是机器学习算法，XGBoost 是该算法的工程实现；
- 在使用 CART 作为基础分类器时，XGBoost 显示地加入了正则项来控制模型的复杂度，有利于防止过拟合，从而提高模型的泛化能力；
- GBDT 在模型训练时只使用了损失函数的一阶导数信息，XGBoost 对代价函数进行二阶泰克展开，可以同时使用一阶和二阶导数；
- 传统的 GBDT 采用 CART 作为基础分类器，XGBoost 支持多种类型的基础分类器，比如线性分类器；
- 传统的 GBDT 在每轮迭代时使用全部的数据，XGBoost 则支持对数据进行采样；
- 传统的 GBDT 没有涉及对缺失值进行处理，XGBoost 能够自动学习出缺失值的处理策略；
- XGBoost 还支持并行计算， XGBoost 的并行是基于特征计算的并行，将特征列排序后以 block 的形式存储在内存中，在后面的迭代中重复使用这个结构。

XGBoost 为什么可以并行训练

注意 XGBoost 的并行不是树粒度的并行，XGBoost 也是一次迭代完才能进行下一次迭代的（第 t 次迭代的代价函数里包含了前面 t-1 次迭代的预测值）。

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在多线程进行**。

树节点在进行分裂时，我们需要计算每个特征的每个分割节点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以 XGBoost 还提出了一种**可并行的直方图算法**，用于高效地**生成候选的分割点**。

XGBoost 防止过拟合的方法

1. 数据
    - 样本采样 - 在生成每颗树的时候可以对数据进行随机采样。
    - 特征采样 - 还可以在生成每棵树的时候，每棵树生成每一层子节点的时候，以及在每次做节点分裂的时候，选择是否对特征进行采样。

2. 模型
    - 限制树的深度，树的深度越深模型越复杂。
    - 设置叶子节点上样本的最小数量，这个值越小则树的枝叶越多模型越复杂；相反如果这个值越大，则树的枝叶越少，模型越简单。这个值太小会导致过拟合，太大会导致欠拟合。
3. 正则化
    - L1 和 L2 正则化损失项的权重
    - 叶子节点数量惩罚项的权重值

XGBoost 为什么这么快

1) 同时采用了损失函数的一阶导数和二阶导数，使得目标函数收敛更快。
2) 在进行节点分类时采用的贪心算法和直方图算法，大大加速了节点分裂的计算过程。
3) 工程优化，使得模型训练时可进行并行计算。

深度学习基础

非线性

为什么必须在神经网络中引入非线性

如果神经网络中没有引入非线性层，那么神经网络就变成了线性层的堆叠。而多层线性网络的堆叠本质上还是一个线性层，我们以两层线性网络的堆叠为例：

我们用 $f(x)$表示第一层线性网络，$g(x)$表示第二层线性网络，则两层网络的堆叠表示为：

$$h(x) = g(f(x)) = {w_2}^T({w_1}^T x + b_1) + b_2 = {w_2}^T{w_1}^T x + {w_2}^T b_1 + b_2$$

我们令：

$${w_2}^T{w_1}^T = w^{*T}$$

$${w_2}^T b_1 + b_2 = b^*$$

那么原来的表达式就变为：

$$h(x) = w^{*T}x + b^*$$

所以$h(x)$ 还是一个线性函数。而我们知道线性函数的表现力是有限的，它只能表示特征与目标值之间比较简单的关系，相反带有非线性层的神经网络被证明可以表示任何函数。所以为了使得网络设计发挥作用，并且提高网络的表现力，必须要在神经网络中引入非线性。

ReLU 在零点不可导，那么在反向传播中怎么处理

ReLU 虽然在零点不可导，但是我们在做反向传播的计算时，对 Relu 这个函数的导数分情况讨论，即 ReLU 在零点时人为地给它赋予一个导数，比如 0 或者 1。例如在下面 ReLU 的反向传播函数实现中，将 ReLU 在零点位置的导数设置为 0。

```python
def relu(Z):
    """
    Numpy Relu activation implementation

    Arguments:
    Z - Output of the linear layer, of any shape

    Returns:
    A - Post-activation parameter, of the same shape as Z
    cache - a python dictionary containing "A"; stored for computing the backward pass efficiently
    """
    A = np.maximum(0,Z)
    cache = Z
    return A, cache

def relu_backward(dA, cache):
    """
    The backward propagation for a single RELU unit.

    Arguments:
    dA - post-activation gradient, of any shape
    cache - 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ - Gradient of the cost with respect to Z
    """
    Z = cache
    # just converting dz to a correct object.
    dZ = np.array(dA, copy=True)
    # When z <= 0, we should set dz to 0 as well.
    dZ[Z <= 0] = 0
    return dZ
```

ReLU 的优缺点

优点：

1. 使用 ReLU 的 SGD 算法的收敛速度比 sigmoid 和 tanh 快；
2. 在 x > 0 上，不会出现梯度饱和，梯度消失的问题。
3. 计算复杂度低，不需要进行指数运算，只要一个阈值（0）就可以得到激活值。

缺点：

1. ReLU 的输出不是 0 均值的，它将小于 0 的值都置为 0；使得所有参数的更新方向都相同，导致了 ZigZag 现象。
2. Dead ReLU Problem (ReLU 神经元坏死现象)：某些神经元可能永远不被激活，导致相应参数永远不会被更新(在负数部分，梯度为 0)
3. ReLU 不会对数据做幅度压缩，所以数据的幅度会随着模型层数的增加不断扩张。

注：ZigZag 现象指的是，模型中所有的参数在一次梯度更新的过程中，更新方向相同，即同为正或者同为负。这就导致了梯度更新图像呈现 Z 字形，进而导致梯度更新效率比较低。因为各个参数不能朝着总体梯度下降最快的方向更新，一些参数的更新需要“等待” 另外一些参数的更新。ReLU 激活函数导致 ZigZag 现象的推导过程如下：

$$f = \sum w_i x_i + b, \qquad x_i > 0 (x_i \textit{是通过激活函数得到的})$$

$$\frac{\partial f}{\partial w_i} = x_i > 0$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i} = \frac{\partial L}{\partial f} x_i$$

$$\because x_i > 0$$

$$\therefore sign\left(\frac{\partial f}{\partial w_i}\right) = sign\left(\frac{\partial L}{\partial f}\right)$$

其中$sign()$表示的是符号函数。通过以上推导我们可以得出，在同一个线性层中所有参数的导数符号都相同，而且只取决于 Loss 函数对当前层函数导数的符号。

激活函数有什么作用，常用的激活函数

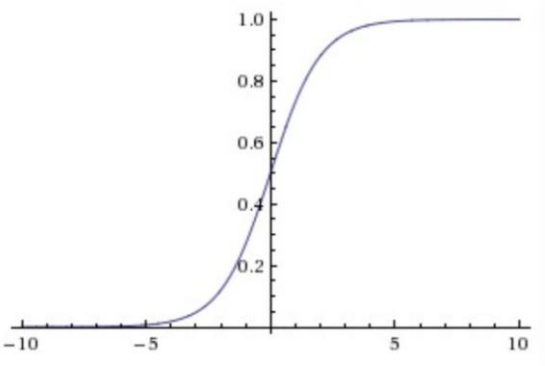
如果不用激活函数（其实相当于激活函数是 f(x) = x），在这种情况下你每一层节点的输入都是上层输出的线性函数，所以无论神经网络有多少层，输出都是输入的线性组合。这种情况就是最原始的感知机了，那么网络的表达能力就相当有限。正因为上面的原因，引入非线性函数作为激活函数之后，深层神经网络表达能力就很强大了（不再是输入的线性组合，而是几乎可以表示任意函数）。

常用的激活函数有：Sigmoid, Tanh, ReLU, Leaky ReLU,

Sigmoid 激活函数的公式是：

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

其图像为：



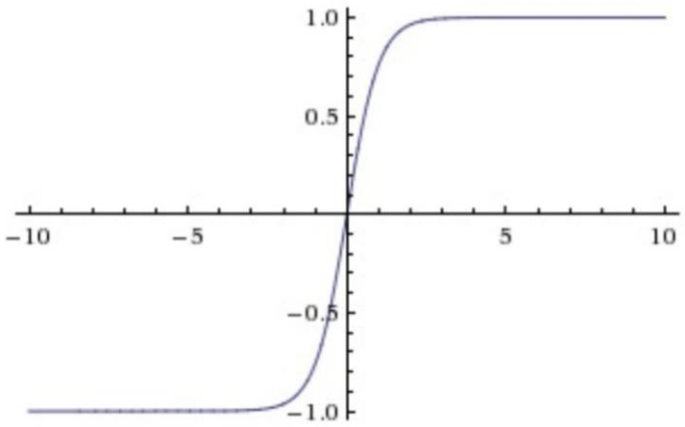
Sigmoid 将回归问题的结果转换到(0,1)的范围，图像成中间窄两边宽的结构；即大部分的时候 Sigmoid 的值集中的在两边。Sigmoid 函数提供了一个很好的适用于二分类的问题的输出结果，以及非线性表达。但是它的缺点也是非常明显的：

1. 前向传播和反向传播都要做指数计算，计算耗费资源较多。
2. Sigmoid 函数梯度饱和导致神经网络反向传播时出现梯度消失的现象。当神经元的激活在接近 0 或 1 处时会饱和，在这些区域梯度几乎为 0，这就会导致梯度消失，几乎就有没有信号通过传回上一层。
3. Sigmoid 函数的输出不是零中心的。因为如果输入神经元的数据总是正数，那么关于w的梯度在反向传播的过程中，将会要么全部是正数，要么全部是负数，这将会导致梯度下降权重更新时出现 Z 字型的下降（ZigZag 现象）。

Tanh 函数的数学公式是：

$$\tanh(x) = 2sigmoid(2x) - 1$$

其图像为：



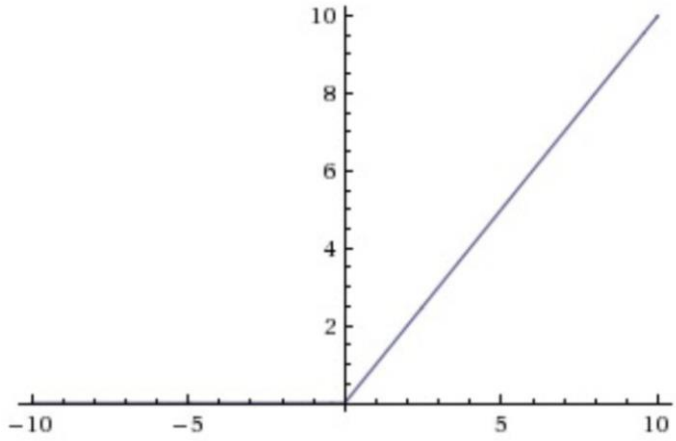
Tanh 激活函数的提出虽然解决了 sigmoid 函数的输出不是 0 中心的问题。但是它仍然存在两个主要问题：

1. 函数梯度饱和的问题；
2. 计算复杂的问题。

ReLU 函数的数学公式是：

$$f(x) = \max(0, x)$$

其图像为：



ReLU 激活函数相较于 Sigmoid 和 Tanh 函数来说，对于随机梯度下降收敛有巨大的作用，很好地解决了梯度消失的问题；同时函数的前向和反向传播计算计算效率也高。

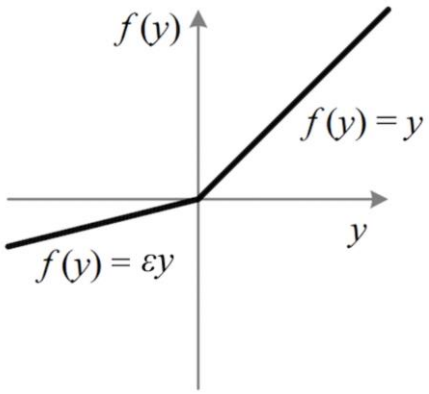
但是 ReLU 存在以下缺点：

1. ReLU 的输出不是零中心的，所以梯度下降时存在 ZigZag 现象。
2. ReLU 函数存在小于等于 0 一侧，神经元坏死的现象。

Leaky ReLU 函数的公式是：

$$f(y) = \max(\varepsilon y, y)$$

函数的图像为：



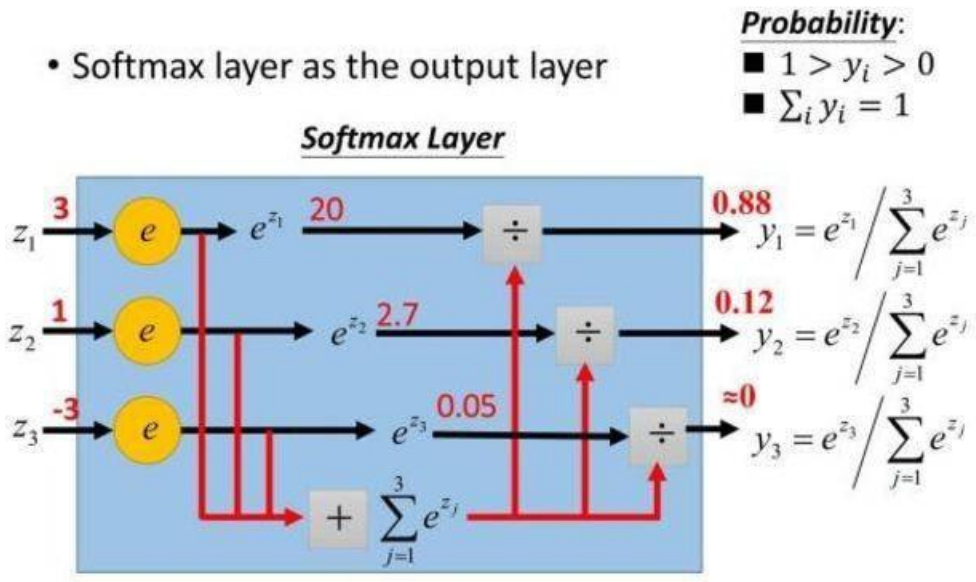
其中ε通常取很小的值比如 0.01，这样可以确保负数信息没有全部丢失。解决了 ReLU 神经元坏死的问题；并且一定程度上缓解了数据均值不为 0 导致的 ZigZag 现象。

Softmax 的原理是什么，有什么作用

Softmax 的函数公式是：

$$\delta(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Softmax 层的结构示意图如下：



它具有以下特性：

1. 经过 Softmax 之后，每一路的输出，在(0,1)范围内，即$0 < y_i < 1$
2. 所有路输出之和为 1，即$\sum_i \boldsymbol{y_i} = \mathbf{1}$
3. 经过 Softmax 之后，比较大的输入在输出时会变得相对更大(参考指数函数的性质来理解)。

基于以上三个特性，Softmax 函数经常被用在分类深度神经网络的输出头，做多分类输出。

梯度训练

BN 解决了什么问题

BN（Batch Normalization）的提出是为了解决 Internal Covariate Shift 的问题，即当深度神经网络经过多层的计算，输入到每一层的数据分布发生了严重的改变；这种输入数据上的抖动和变化，导致网络参数难以学习，随着网络的加深模型更加难以收敛。BN 层的引入，使得每一层的输入数据都归一化到近似于均值为 0 方差为 1 的分布上，从而大大缓解了 Internal Covariate Shift 的问题。此外，BN 还给深度神经网络带来了以下好处：

1. BN 可以解决每一层数据不稳定的问题 （上诉提到的问题）;
2. BN 可以解决梯度消失的问题;
    a. 例如在 Sigmoid 激活函数里面，当输入值非常大或者非常小的时候，梯度接近于 0；而将数据归一化到（0，1）之间，梯度可以很好地得到更新。
3. 网络的训练速度加快;
4. 引入 BN 层之后，使得激活函数的使用更加丰富和灵活;
5. 使得网络参数初始化和学习率的参数调节更加容易和灵活。

BN 的实现流程

BN 的过程可以总结为：

- 按 batch 进行期望和标准差计算
- 对整体数据进行标准化
- 对标准化的数据进行线性变换
- 其中变换系数需要学习

用公式描述为：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

用 python 程序实现的前向计算为：

```
def batchnorm_forward(x, gamma, beta, eps):

  N, D = x.shape

  #step1: calculate mean
  mu = 1./N * np.sum(x, axis = 0)

  #step2: subtract mean vector of every trainings example
  xmu = x - mu

  #step3: following the lower branch - calculation denominator
  sq = xmu ** 2

  #step4: calculate variance
  var = 1./N * np.sum(sq, axis = 0)

  #step5: add eps for numerical stability, then sqrt
  sqrtvar = np.sqrt(var + eps)

  #step6: invert sqrtwar
  ivar = 1./sqrtvar

  #step7: execute normalization
  xhat = xmu * ivar

  #step8: Nor the two transformation steps
  gammax = gamma * xhat

  #step9
  out = gammax + beta

  #store intermediate
  cache = (xhat,gamma,xmu,ivar,sqrtvar,var,eps)

  return out, cache
```

反向传播计算为：

```
def batchnorm_backward(dout, cache):

  #unfold the variables stored in cache
  xhat,gamma,xmu,ivar,sqrtvar,var,eps = cache

  #get the dimensions of the input/output
  N,D = dout.shape

  #step9
  dbeta = np.sum(dout, axis=0)
  dgammax = dout #not necessary, but more understandable

  #step8
  dgamma = np.sum(dgammax*xhat, axis=0)
  dxhat = dgammax * gamma

  #step7
  divar = np.sum(dxhat*xmu, axis=0)
  dxmu1 = dxhat * ivar

  #step6
  dsqrtvar = -1. /(sqrtvar**2) * divar

  #step5
  dvar = 0.5 * 1. /np.sqrt(var+eps) * dsqrtvar

  #step4
  dsq = 1. /N * np.ones((N,D)) * dvar

  #step3
  dxmu2 = 2 * xmu * dsq

  #step2
  dx1 = (dxmu1 + dxmu2)
  dmu = -1 * np.sum(dxmu1+dxmu2, axis=0)

  #step1
  dx2 = 1. /N * np.ones((N,D)) * dmu

  #step0
  dx = dx1 + dx2

  return dx, dgamma, dbeta
```

注：这里的程序实现只给出了一个朴素的版本，并不包含对全连接层以及 Conv 层结果的区分对待以及工程优化。正反向传播实现里面的 Step 有一一对应关系，大家可以把两个函数放在一起对照学习。

怎么解决梯度消失问题

在深度网络中，网络参数的学习是通过反向传播的链式求导法则来求 Loss 对某个参数的偏导数，然后进行参数更新的。因此造成梯度消失的原因主要有两个：1.当网络层数很深，而当前的参数所在层又靠近网络的输入时，求导链就会非常长；2.如果其中

的某些中间结果的值很小，并经过链式的累成作用，最终求得的梯度值就会接近于零，而导致参数得不到更新。

可通过以下方法解决梯度消失的问题：

1. 选用合适的激活函数。比如 ReLU 或者 Leaky ReLU。因为像 Sigmoid 和 Tanh 这样的激活函数，会出现比较大的梯度饱和区域，使得梯度的取值接近于 0。
2. 采用 Batch Normalization 层，对网络中计算得到中间值进行归一化，使得中间计算结果的取值在均值为 0，方差为 1 这样的分布内。那么此时，在 sigmoid 和 tanh 中，函数取值处于中间变化较大的部分，梯度取值也相对较大，从而可以防止过拟合。
3. 使用残差结构，残差结构相当于给靠近输入端的网络层提供了一个与靠近输出端层的直连操作。在反向传播计算时，减少了梯度传播的路径长度，以缓解梯度消失的问题。
4. 在 RNN 网络中，可以通过使用 LSTM（long-short term memory networks）长短时记忆网络，来解决信息遗忘和梯度传播的问题。

列举几个梯度下降的方法

梯度下降方法的演化通常可以从以下三个方面进行总结：

梯度更新计算涉及的样本数

- Batch Gradient Descent (BGD)
    - 批量梯度下降，对所有的样本计算梯度后求平均，并更新参数；
    - 下降速度很慢，受到内存大小的限制。
- Stochastic Gradient Descent (SGD)
    - 随机梯度下降，对每个样本计算梯度，并更新一次参数；
    - SGD 的梯度更新速度快；
    - 由于样本之间差异的影响，导致梯度下降的方向和大小出现剧烈的波动；
- Mini-batch Gradient Descent

- 小批量梯度下降，每一次梯度更新是基于对小批量样本计算梯度后求平均完成的。
- 综合了以上两种梯度下降方法的优点，梯度更新快且稳定。
- 在内存相对较小的情况下，也可以完成对大训练集的训练工作。

- 梯度下降方法的一般公式为：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

其中θ_t表示上一轮跟新后参数值，η表示学习率。

梯度下降方向的优化

- Momentum 动量算法

$$\theta_{t+1} = \theta_t - m_t$$

$$m_t = \gamma m_{t-1} + \eta \nabla_\theta J(\theta_t)$$

  - 其中m_t代表动量，它是之前t次梯度的带权累加；γ表示历史动量影响的权重；
  - 利用惯性，做动量更新的全局平衡，即每次更新时不仅要考虑当前梯度方向还要考虑历史累积的方向。当前梯度与上次梯度进行加权，如果方向一致，则累加导致更新步长变大；如果方向不同，则相互抵消中和导致更新趋向平衡；

- Nesterov Accelerated Gradient (NAG)

$$\theta_{t+1} = \theta_t - m_t$$

$$m_t = \gamma m_{t-1} + \eta \nabla_\theta J(\theta_t - \gamma m_{t-1})$$

  - 梯度不是根据当前位置θ_t计算出来的，而是在移动之后的位置$\theta_t - \gamma m_{t-1}$上计算梯度
  - 这个改进的目的就是为了提前看到前方的梯度，并做一定的补偿与平衡。如果前方的梯度和当前梯度目标一致，两者相加就获得比较大的更新； 如果前方梯度同当前梯度不一致，两者相加在数值大小上获得相减的效果，即得到比较小的更新。

梯度更新大小的优化

- Adagrad (Adaptive Subgradient)

$$\theta_{i,t+1} = \theta_{i,t} - \frac{\eta}{\sqrt{G_{i,t} + \varepsilon}} \nabla_{\theta_{i,t}} J(\theta)$$

ϵ 一般是一个极小值，作用是防止分母为 0 。$G_{i,t}$表示了前t 步参数θ_i梯度的平方累加，把沿路的 Gradient 的平方根，作为正则化项。

- 训练前期，梯度较小，使得正则化项很小；起到放大梯度的作用。
- 训练后期，梯度较大，使得正则化项很大；起到缩小梯度的作用。

- RMSprop

$$E[g^2]_{i,t} = \gamma E[g^2]_{i,t-1} + (1-\gamma) g_{i,t}^2$$

$$\theta_{i,t+1} = \theta_{i,t} - \frac{\eta}{\sqrt{E[g^2]i,_t + \varepsilon}} \nabla_{\theta_{i,t}} J(\theta)$$

  - RMSProp 算法不是像 AdaGrad 算法那样直接地累加平方梯度，而是加了一个衰减系数来控制获取多少历史信息

综合以上

- Adam
  - 同 RMSprop 相比，Adam 在对梯度平方（二阶）估计的基础上增加了对梯度(一阶)的估计，相当于采用了 Momentum 动量方法。

$$E[g]_{i,t} = \beta_1 E[g]_{i,t-1} + (1-\beta_1) g_{i,t}$$

$$E[g^2]_{i,t} = \beta_2 E[g^2]_{i,t-1} + (1-\beta_2) g_{i,t}^2$$

  - 由于 $E[g]_{i,t}$和 $E[g^2]_{i,t}$通常都初始化为 0， 因此在训练初期或者 β_1, β_2,接近 1 时，估计的期望往往偏向于 0, 为了解决这种偏置，增加了偏置矫正：

$$E[\hat{g}]_{i,t} = \frac{E[g]_{i,t}}{1-\beta_1}$$

$$E\left[\widehat{g^2}\right]_{i,t} = \frac{E[g^2]_{i,t}}{1-\beta_2}$$

  - 梯度更新公式为：

$$\theta_{i,t+1} = \theta_{i,t} - \frac{\eta}{\sqrt{E\left[\widehat{g^2}\right]_{i,t} + \varepsilon}} E[\hat{g}]_{i,t}$$

计算机视觉

CNN 经典模型

什么是端到端学习

在传统的机器学习系统中，我们通常需要对数据进行多阶段的处理，在每个阶段使用人工特征提取，数据标注或者独立的机器学习模型来完成特定的任务；而在端到端学习系统中，可以通过训练一个大的神经网络来完成从原始数据的输入到最终结果输出的整个任务。端到端的学习可以大大减少人工参与，简化解决方案部署流程。

现实中的任务通常非常复杂，将复杂任务拆分成简单的任务解决通常能够达到很好的效果；比如我们在做人脸识别任务时，通常会把任务拆分成三个子任务 1.人脸检测；2.人脸关键点对齐和矫正；3.经过切分、对其后人脸图像的人脸识别。如果要将这个任务变成端到端的任务，需要大量的从输入数据到输出结果的数据集；而现实中我们并没有这样大量的数据。所以在这个任务中端到端学习并不适用，而在有些我们已经有大量数据的任务中，比如语音识别并转换成文本的任务中，端到端学习就表现得很好。

CNN 的平移不变性是什么，如何实现的

平移不变性（translation invariant）指的是 CNN 对于同一张图及其平移后的版本，都能输出同样的结果。这对于图像分类（image classification）问题来说肯定是最理想的，因为对于一个物体的平移并不应该改变它的类别。

通常认为，CNN 网络的平移不变性是通过卷积+池化来实现的。

卷积：简单地说，图像经过平移，相应的特征图上的表达也是平移的。在神经网络中，卷积被定义为不同位置的特征检测器，也就意味着，无论目标出现在图像中的哪个位置，它同样都会检测到这些特征，输出相同的响应。

池化：比如最大池化，它返回感受野中的最大值，如果最大值被移动了，但是仍然在这个感受野中，那么池化层也仍然会输出相同的最大值。这两种操作共同提供了一些平移不变性，即使图像被平移，卷积保证仍然能检测到它的特征，池化则尽可能地保持一致的表达。

但是在近几年的研究中，人们发现，当深度网络加深之后，CNN 的平移不变性越来越难以保证。比如说，当图像中的物体发生小范围移动的话，网络预测的结果会出现较大的波动。其中一个主要原因来自下采样，比如 stride 为 2 的下采样，它是将网络人为的按照规则分块之后，然后再进行采样；这样被采样的对象经过微小平移之后，就会在采样块之间发生位置变化，导致每个采样块采样出来的结果与平移前的结果有所差别，而经过多层这样的采样之后，能够保持平移前结果不变的概率会越来越小。

AlexNet, VGG，GoogleNet，ResNet 等网络之间的区别是什么

AlexNet 相比传统的 CNN，主要改动包括 Data Augmentation（数据增强）、Dropout 方法、激活函数用 ReLU 代替了传统的 Tanh 或者 Sigmoid、采用 Local Response Normalization（LRN，实际就是利用临近的像素数据做归一化）、Overlapping Pooling（有重叠，即 Pooling 的步长比 Pooling Kernel 的对应边要小）、多 GPU 并行。

VGG 很好地继承了 AlexNet 的特点，采用了更小的卷积核堆叠来代替大的卷积核，并且网络更深。

GoogLeNet，网络更深，但主要的创新在于他的 Inception，这是一种网中网（Network In Network）的结构，即原来的节点也是一个网络。相比于前述几个网络，用了 Inception 之后整个网络结构的宽度和深度都可扩大，能够带来 2-3 倍的性能提升。

ResNet 在网络深度上有了进一步探索。但主要的创新在残差网络，网络的提出本质上还是要解决层次比较深的时候无法训练的问题。这种借鉴了 Highway Network 思想的网络相当于旁边专门开个通道使得输入可以直达输出，而优化的目标由原来的拟合输出 $H(x)$变成输出和输入的差$H(x)-x$，其中$H(x)$是某一层原始的期望映射输出，x是输入。

Pooling 层做的是什么

池化层（Pooling Layer）也叫子采样层（Subsampling Layer）它的一个很重要的作用就是做特征选择，减少特征数量，从而减少网络的参数数量。采用了 Pooling

Layer 之后，在不增加参数的情况下，可以增大网络对图像的感受野。虽然可以用带步长的卷积（步长大于 1）代替 Pooling layer 也可以实现下采样和增大感受野的作用，但是 Pooling Layer 的计算简单，同时也没有可学习的参数，在很多需要做特征图拼接的网络，比如 ShuffleNet 中是非常适用的。

Dropout 是否用在测试集上

在训练的时候 Dropout 打开，功能类似于每次将网络进行采样，只训练一部分网络。而训练结束，在测试集上做测试的时候，就要将这些每次训练好的“子网络”集成起来，一起做决策。所以要关闭 Dropout,即关闭网络采样功能。

Inception module 的优点是什么

Inception Module 基本组成结构有四个成分。1*1 卷积，3*3 卷积，5*5 卷积，3*3 最大池化。最后对四个成分运算结果进行通道上组合。这就是 Inception Module 的核心思想。**通过多个卷积核提取图像不同尺度的信息，最后进行融合，可以得到图像更好的表征**。

CNN 中的 1x1 卷积有什么作用

1. 通道的升降

卷积层参数量的计算公式为：

$$Number\_params = In\_channel * Kernel\_size * Kernel\_size * Out\_channel$$

如果要使用一个 3x3 的卷积做通道的升降，那么参数量将会是用 1x1 卷积做通道升降时的 9 倍。当输入通道和输出通道的维数很大的时候；这样参数量的增加是非常惊人的。

2. 增强了通道之间的交互和网络的表达能力

举个例子， 在一个网络中，如果我们要用一个 3x3 的卷积来对一个有 512 个 channel 的 feature map 做卷积操作；我们可以先采用一个 1x1 的卷积对这个 feature map 做通道的降维，降到 128 维；然后在做 3x3 的卷积操作，最后再用

1x1 卷积做通道的升维，升到 512 维。这个方案和我们直接用 3x3 的卷积进行卷积操作相比。 1）参数数量上大大减少，直接用 3x3 的卷积需要的参数量为 512*512*3*3 ≈ 2.35M，而采用 1x1 卷积的方案，参数量为 512*128 * 1*1*2 + 128*128*3*3 ≈ 0.28M；可以看到参数量几乎等于原来的 1/9。 2）通道之间的交互增强 - 在 1x1 卷积方案中，通道之间的交叉计算做了 3 次，而 3x3 原始方案中只进行了 1 次。 3）表达能力增强 - 通常在每次卷积操作之后会加一个非线性层；那么在 1x1 网络方案中非线性层的数量为 3 层，而原始 3x3 网络方案中非线性层只有 1 层。

Pytorch 实现 VGG16 的网络

我们可以从两个阶段来理解 VGG16 网络，第一阶段为特征提取，第二阶段为分类功能头。在特征提取阶段，VGG16 分为了 5 个 conv + maxpooling 网络块，并且所有的 conv 层都是 3x3 的卷积层，每个块内部卷积的输入与输出 channel 维度是一样的。在分类功能头部分，由三个全连接线性层加一个 Softmax 层做多分类输出。如下图所示：

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

基于此，我们的网络实现可如下进行（我将 PyTorch 官方的 vgg16 的实现做了一定的删减，只保留了 vgg16 的核心内容，以及可扩展的接口结构并对关键代码加入了注释帮助大家理解），注意：每个 conv 层和 FC 层之后都加入了一个非线性层；PyTorch 中用于分类的 nn.CrossEntropyLoss 损失函数，本身包含 Softmax 操作，所以在网络中没有加入 Softmax 层。

```
1.  import torch
2.  import torch.nn as nn
3.  from typing import Union, List, Dict, Any, cast
4.
5.
6.  class VGG(nn.Module):
7.
8.      def __init__(
9.          self,
10.         features: nn.Module,
11.         num_classes: int = 1000,
12.         init_weights: bool = True
13.     ) -> None:
14.         super(VGG, self).__init__()
15.         self.features = features # 从外部传入特征提取网络 - vgg 各个版本的分类功能头是一
    样的，而只是特征提取网络是不同的
16.         # 构建分类功能头
```

```
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            self._initialize_weights()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        网络结构 特征提取 -> feature map 扁平化 -> 分类功能头
        """
        x = self.features(x) # 特征提取
        x = torch.flatten(x, 1) # feature map 扁平化
        x = self.classifier(x) #分类功能头
        return x

    def _initialize_weights(self) -> None:
        for m in self.modules():
            if isinstance(m, nn.Conv2d): # 如果是卷积层则采用 Kaiming 初始化方法
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)


def make_layers(cfg: List[Union[str, int]]) -> nn.Sequential:

    """
    根据网络配置信息，构建特征提取网络
    """
    layers: List[nn.Module] = []
    in_channels = 3
    for v in cfg:
        if v == 'M': # 如果配置信息是'M'则构建 maxpooling layer
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else: # 如果是数字则代表构建 conv 层，并且数字代表的是 conv 层的输出 channel
            v = cast(int, v) # 把网络的输出维度信息转化成 int 型
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1) # 根据输出维度信息构建卷积层
            layers += [conv2d, nn.ReLU(inplace=True)] # 卷积层+ReLU 激活层
            in_channels = v # 将当前输出 channel 保存，作为下一个 conv 层的输入 channel
    return nn.Sequential(*layers)

# 配置不同 vgg 版本的特征提取网络
cfgs: Dict[str, List[Union[str, int]]] = {
    'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
}

```

```
def _vgg(cfg: str, **kwargs: Any) -> VGG:
    model = VGG(make_layers(cfgs[cfg]), **kwargs)
    return model

def vgg16(**kwargs: Any) -> VGG:

    return _vgg('D', **kwargs)
```

目标检测

ROI pooling 的不足是什么

ROI Pooling (Region of Interests) 在 Faster R-CNN 中提出，将经过 Backbone 之后的 feature map pooling 到同一个 size。这个方法有一个主要的缺点：需要进行 2 次量化取整的操作，会带来**精度的损失。**

第一次是经过 Region Proposal Network(RPN)预测得到的 Bounding Box 边界框是个 float 数值，而不是整数，这样就涉及到一次取整的操作，取整后才可以输入 ROI Pooling。第二次是 ROI Pooling 需要对提取的 ROI 做 pooling 操作，输出一个 7*7 大小的 feature map，即当前的 ROI 中的像素需要划分到 7*7 个桶（区域）中。这样 ROI 的长度或者宽度除以 7，得到的也是一个 float 数值，而非整数，这个时候又涉及到一次量化取整的操作。

ROI align 的具体做法是什么

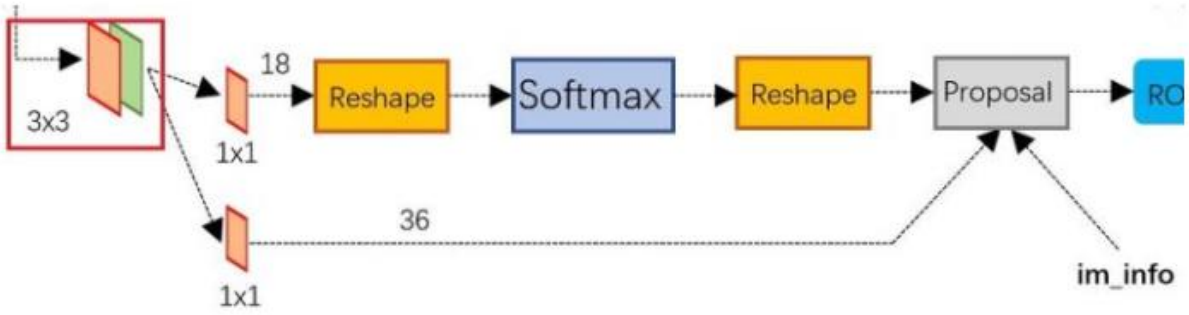
ROI Align 的计算流程如下：

1. 由输入的 ground truth box 大小，经过 backbone 的缩放（比如 32），得到输出的 shared layer 中 ground truth box 大小（浮点数而不是整数）；
2. 对于缩放后的 ground truth box，将其平均分为（n，n）大小的小块（这里的 n 即为 pooling 后的 anchor 的边长）；
3. 对于每个小块，假设我们的采样率为 2，那么在每个小块内，要采样$2x2 = 4$个点，四个点的位置按小块内的$1/4,3/4$位置取；
4. 按双线性插值计算 4 个点的像素值，取 4 个点中的像素值 max 作为这个小块的像素值。

Faster RCNN 中 RPN 相比之前做了什么优化

采用经典的检测方法生成检测框时，非常耗时，如 OpenCV Adaboost 使用滑动窗口+图像金字塔生成检测框；或如 R-CNN 和 Fast R-CNN 使用 SS(Selective Search)方法生成检测框。而 Faster RCNN 则抛弃了传统的滑动窗口和 SS 方法，直接使用

RPN(Region Proposal Network)生成检测框，这也是 Faster R-CNN 的巨大优势，能极大提升检测框的生成速度。



上图展示了 RPN 网络的具体结构。可以看到 RPN 网络实际分为两个分支，上面一个分支通过 Softmax 分类 anchors 获得 positive 和 negative 分类，下面一个分支用于计算对于 anchors 的 bounding box regression 偏移量，以获得精确的 proposal。而最后的 Proposal 层则负责综合 positive anchors 和对应 bounding box regression 偏移量获取 proposals，同时剔除太小和超出边界的 proposals。

YOLO v3 进行了几次下采样

这道题目是对 YOLOv3 网络结构的考察。YOLOv3 网络对原始的输入数据进行了 5 次下采样，采样得到的最小特征图示对原图像进行 32 倍缩放后的结果；并分别在 8 倍，16 倍和 32 倍下采样之后进行了 bbox 的分类和回归定位，如下图所示。

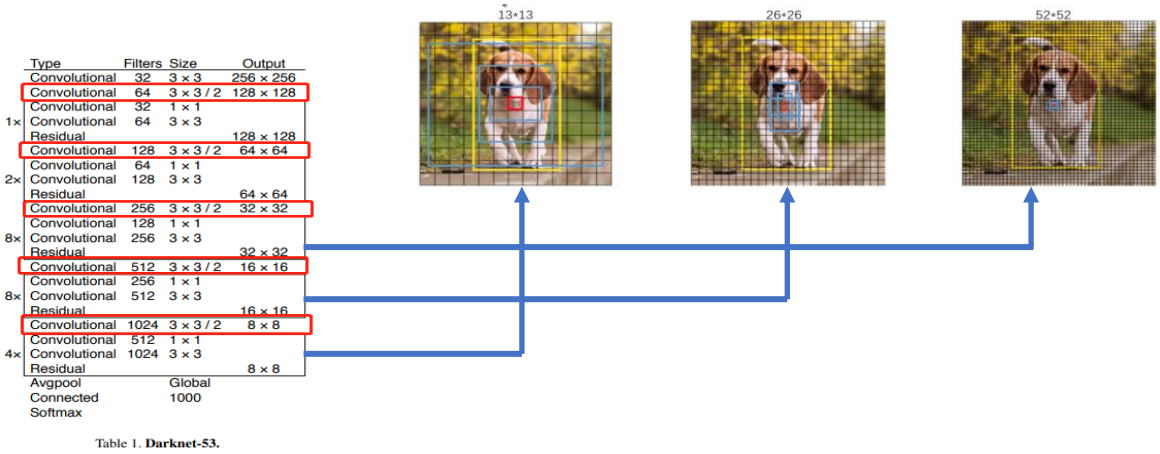


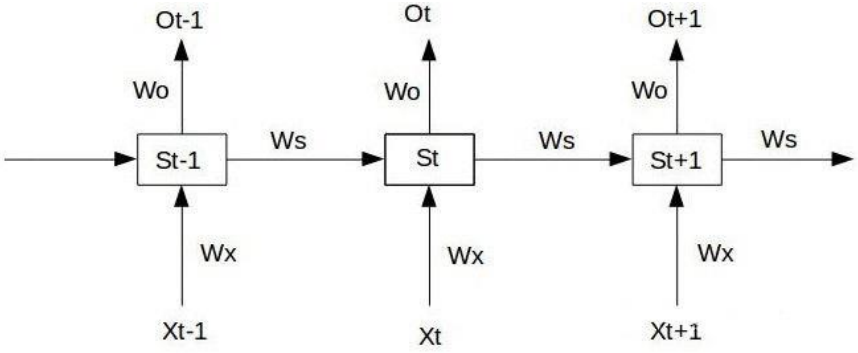
Table 1. **Darknet-53.**

自然语言处理

RNN&CNN

RNN 发生梯度消失的原因是什么

RNN 发生梯度消失的主要原因来自于**长程依赖**问题。



我们假设最左端的输入 S_0 为给定值， 且神经元中没有激活函数， 则前向传播过程如下：

$$S_1 = \tanh(W_x X_1 + W_s S_0 + b_1),\ O_1 = W_0 S_1 + b_2$$

$$S_2 = \tanh(W_x X_2 + W_s S_1 + b_1),\ O_2 = W_0 S_2 + b_2$$

$$S_3 = \tanh(W_x X_3 + W_s S_2 + b_1),\ O_3 = W_0 S_3 + b_2$$

假设在$t = 3$ 时刻，损失函数为$L_3 = \frac{1}{2}(Y_3 - O_3)^2$，那么如果我们要训练 RNN 时，实际上就是对$W_x, W_s, W_0, b_1, b_2$求偏导，并不断调整它们以使得 L_3 尽可能达到最小。那么我们得到反向传播的公式为（在这里我们着重关注权重项 W，而忽略偏置项 b）：

$$\frac{\partial L_3}{\partial W_0} = \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial W_0}$$

$$\frac{\partial L_3}{\partial W_x} = \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial W_x} + \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial S_2}\frac{\partial S_2}{\partial W_x} + \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial S_2}\frac{\partial S_3}{\partial S_1}\frac{\partial S_1}{\partial W_x}$$

$$\frac{\partial L_3}{\partial W_s} = \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial W_s} + \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial S_2}\frac{\partial S_2}{\partial W_s} + \frac{\partial L_3}{\partial O_3}\frac{\partial O_3}{\partial S_3}\frac{\partial S_3}{\partial S_2}\frac{\partial S_3}{\partial S_1}\frac{\partial S_1}{\partial W_s}$$

我们发现，神经网络层数的加深对W_0而言并没有什么影响， 而对W_x， W_s会随着时间序列的拉长产生梯度消失和梯度爆炸问题。根据上诉分析整理一下公式可得，在任意时刻t对W_x, W_s求偏导的公式为：

$$\frac{\partial L_t}{\partial W_x} = \sum_{k=0}^{t} \frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial S_t}\left(\prod_{j=k+1}^{t} \frac{\partial S_j}{\partial S_{j-1}}\right)\frac{\partial S_t}{\partial W_x}$$

$$\frac{\partial L_t}{\partial W_s} = \sum_{k=0}^{t} \frac{\partial L_t}{\partial O_t}\frac{\partial O_t}{\partial S_t}\left(\prod_{j=k+1}^{t} \frac{\partial S_j}{\partial S_{j-1}}\right)\frac{\partial S_t}{\partial W_s}$$

从上面的式子可以看出，导致梯度消失和爆炸的原因就在于 $\prod_{j=k+1}^{t}\frac{\partial S_j}{\partial S_{j-1}}$。 将激活函数展开得到：

$$\prod_{j=k+1}^{t} \frac{\partial S_j}{\partial S_{j-1}} = \prod_{j=k+1}^{t} tanh' W_s$$

而在这个公式中，tanh 的导数总是小于 1 的，如果W_s也是一个大于 0 小于 1 的值，那么随着t的增大，上述公式的值越来越趋近于 0，这就导致了梯度消失问题。(梯度爆炸问题的原理类似)

RNN 中使用 ReLU 可以解决梯度消失问题吗

虽然 ReLU 在值大于 0 处，导数值恒为 1，可以使得激活函数的导数不至于很小，产生梯度饱和的问题；但是在 RNN 中，ReLU 并不能很好的解决梯度消失的问题，理由如下。

参照题目(**RNN 发生梯度消失的原因是什么？**)的推导过程，我们现将

$$S_t = \tanh(W_x X_1 + W_s S_{t-1} + b_1)$$

替换为：

$$S_t = \mathrm{ReLU}(W_x X_1 + W_s S_{t-1} + b_1)$$

那么

$$\prod_{j=k+1}^{t} \frac{\partial S_j}{\partial S_{j-1}} = \begin{cases} \prod_{j=k+1}^{t} W_s, W_x X_1 + W_s S_{t-1} + b_1 \text{恒大于} 0 \\ 0 \end{cases}$$

所以，即使采用了 ReLU 作为激活函数，如果W_s是个大于 0 小于 1 的值，也会导致梯度消失的问题。

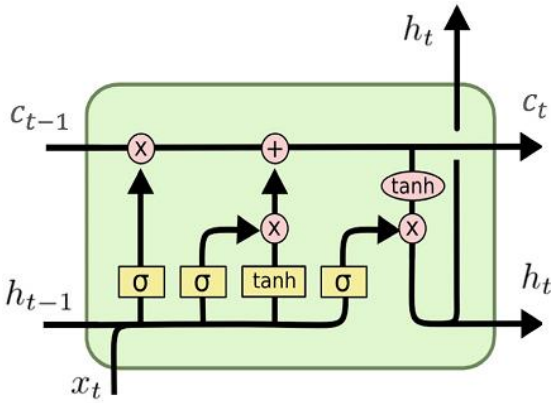
LSTM 为什么能解决梯度消失/爆炸的问题

LSTM 可以通过门控机制来解决梯度消失和爆炸的问题。

从题目(**RNN 发生梯度消失的原因是什么？**)的分析中，我们知道，RNN 产生梯度消失和梯度爆炸的原因在于长程依赖。而 LSTM 使用门控函数，有选择地让一部分信息通过。门控由一个 Sigmoid 单元和一个逐点乘积操作组成；Sigmoid 单元输出 1 或 0，用来判断通过还是阻止。LSTM 可以训练门的组合结果。所以，当门是打开的（梯度接近于 1），梯度就不会消失。并且 sigmoid 不超过 1，那么梯度也不会爆炸。

GRU 和 LSTM 的区别

LSTM 的结构和公式如下：



遗忘门：$f_t = \delta(W_f \cdot [h_{t-1}, x_t] + b_f)$

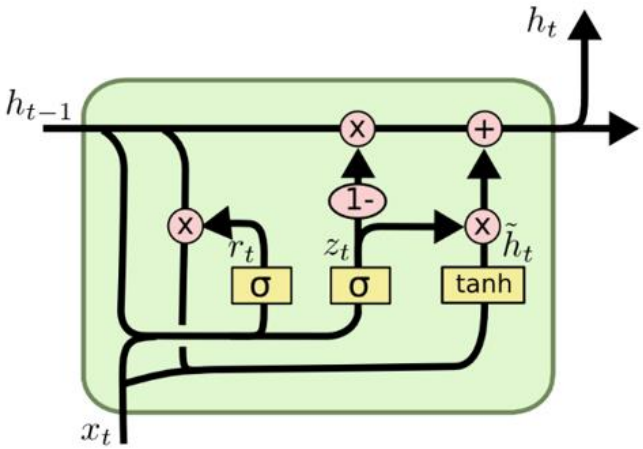
输入门：$i_t = \delta(W_i \cdot [h_{t-1}, x_t] + b_i)$

输出们：$o_t = \delta(W_o \cdot [h_{t-1}, x_t] + b_o)$

当前单元状态：$c_t = f_t * c_{t-1} + i_t * \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

当前时刻的隐层输出：$h_t = o_t * \tanh(c_t)$

而 GRU 的结构和公式如下：



重置们：$r_t = \delta(W_r \cdot [h_{t-1}, x_t])$

更新门：$z_t = \delta(W_z \cdot [h_{t-1}, x_t])$

当前记忆内容：$\widetilde{h_t} = \tanh(W \cdot [r_t * h_{t-1}, x_t])$

当前时刻的隐层输出：$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$

两者的区别有：

1. GRU 没有输出门；
2. 它将隐藏状态和 cell 的记忆状态合并在一起；
3. GRU 的参数更少，因而训练稍快或需要更少的数据来泛化；
4. 如果有足够的数据，LSTM 的强大表达能力可能会产生更好的结果。

LSTM 的不足之处

1. LSTM 模型的门控机制，有点像 ResNet 模型；可以绕过单元（减小反向传播链路长度）从而记住更长的时间步骤（LSTM 之所可以解决梯度消失问题，是因为它避免了长时间序列的连乘，而是将相乘和相加操作结合在一起。这和 ResNet 的网络设计有异曲同工之妙）。但如果面对更长序列的记忆和训练任务，LSTM 仍然会暴露出梯度消失的问题。
2. 计算费时。每一个 LSTM 的 cell 里面都有 4 个全连接层(参照 LSTM 的公式),如果 LSTM 的时间跨度很大，并且网络又很深，这个计算量会很大，很耗时。
3. 容易过拟合，并且 Dropout 方法通常不能给 LSTM 模型带来泛化能力的大幅提升。

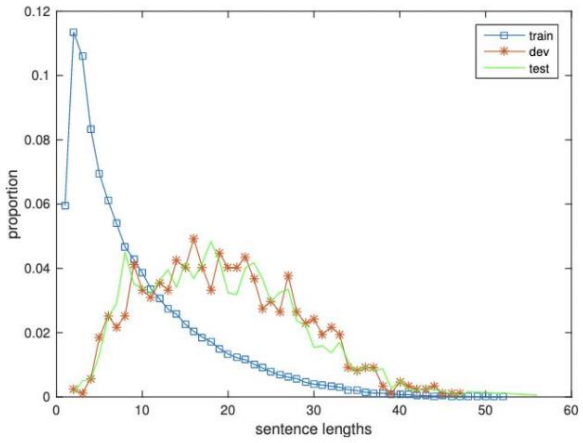
CNN 和 RNN 在 NLP 应用中各自的优缺点

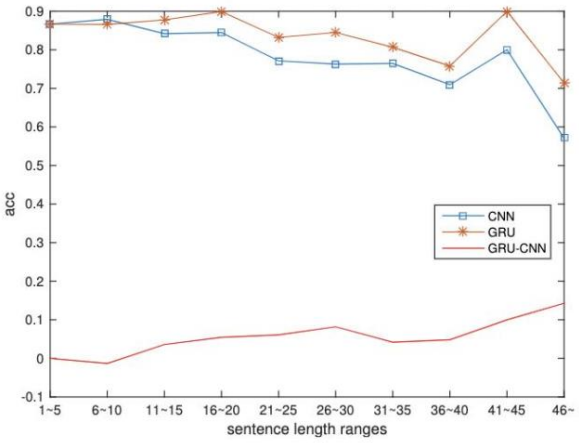
从结构上来分析， CNN 对样本进行逐层抽取，不断扩大感受野，可以应用文本的分类任务；而 RNN 网络具有记忆功能，在序列任务上具有优势。

在 IBM 2017 年的一篇论文中 Comparative Study of CNN and RNN for Natural Language Processing 发现， CNN 对句子匹配任务上， 较 RNN 网络具有一定的优势；在某些文本分类任务上，具有和大致 RNN 相当的能力；而对于序列任务和上下文依赖的任务， RNN 具有明显的优势。

			performance	lr	hidden	batch	sentLen	filter_size	margin
TextC	SentiC (acc)	CNN	82.38	0.2	20	5	60	3	–
		GRU	**86.32**	0.1	30	50	60	–	–
		LSTM	84.51	0.2	20	40	60	–	–
	RC (F1)	CNN	68.02	0.12	70	10	20	3	–
		GRU	**68.56**	0.12	80	100	20	–	–
		LSTM	66.45	0.1	80	20	20	–	–
SemMatch	TE (acc)	CNN	77.13	0.1	70	50	50	3	–
		GRU	**78.78**	0.1	50	80	65	–	–
		LSTM	77.85	0.1	80	50	50	–	–
	AS (MAP & MRR)	CNN	**(63.69,65.01)**	0.01	30	60	40	3	0.3
		GRU	(62.58,63.59)	0.1	80	150	40	–	0.3
		LSTM	(62.00,63.26)	0.1	60	150	45	–	0.1
	QRM (acc)	CNN	**71.50**	0.125	400	50	17	5	0.01
		GRU	69.80	1.0	400	50	17	-	0.01
		LSTM	71.44	1.0	200	50	17	-	0.01
SeqOrder	PQA (hit@10)	CNN	54.42	0.01	250	50	5	3	0.4
		GRU	**55.67**	0.1	250	50	5	–	0.3
		LSTM	55.39	0.1	300	50	5	–	0.3
ContextDep	POS tagging (acc)	CNN	94.18	0.1	100	10	60	5	–
		GRU	93.15	0.1	50	50	60	–	–
		LSTM	93.18	0.1	200	70	60	–	–
		Bi-GRU	94.26	0.1	50	50	60	–	–
		Bi-LSTM	**94.35**	0.1	150	5	60	–	–

另外，当句子长度变长之后， CNN 的表现明显不如 RNN (如下图)。 直观来看，在短句长的任务上， CNN 由于其卷积的功能对句子的整体结构有一个总揽的能力，但在长句长时， CNN 只能处理其窗口内的信息，相邻窗口的信息只能借助后一层的卷积层来达到信息的融合，这对卷积窗口和移动的步长等等参数依赖是很大的，因此 CNN 处理 NLP 任务实际上是具有建模容易、调参难的特点。而 RNN 则训练时间会相对长很多。





论文地址： https://arxiv.org/abs/1702.01923

Transformer

写出 Self-Attention 的公式，Self-Attention 机制里面的 Q,K,V 分别代表什么

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention 机制中，Q,K,V 分别指的是 Query， Key 和 Value；每个输入向量的 Q,K,V 向量是由该输入向量分别与权重矩阵W_q, W_k, W_v做相乘计算得到。通过当前位置的 Q 与其他位置的 K 做相乘计算，可得到当前位置对目标位置的 attention 权重；再将这个 attention 权重（经过 Softmax 后）与目标位置的 Value 相乘就得到了当前位置对目标位置的 attention 编码。将当前位置对所有位置（包括它自己）attention 编码进行相加就得到了当前位置的输出编码。

Transformer 中使用多头注意力的好处是什么

将模型分成多个头，形成多个子空间，可以让模型去关注不同方面的信息，从而捕捉更加丰富的特征。

Attention 中 self-attention 的时间复杂度

假设输入的序列长度为n，d为 Embedding 的维度。那么 Self-attention 的时间复杂度是$O(n^2 \cdot d)$。因为 Self-attention 的公式为：

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

包括三个步骤：相似度的计算，Softmax 和加权平均

- 对于相似度的计算是 Q，K 做点乘，计算的时间复杂度为$O(n^2 \cdot d)$，得到一个大小为(n,n)的矩阵。
- Softmax 的计算，时间复杂度为$O(n^2)$。
- 加权平均可以看作大小为(n,n)和(n,d)的两个矩阵相乘：时间复杂度为 $O(n^2 \cdot d)$，得到一个大小为(n,d)的矩阵

- 综上三步计算，可得到 Self-attention 的时间复杂度$O(n^2 \cdot d)$。

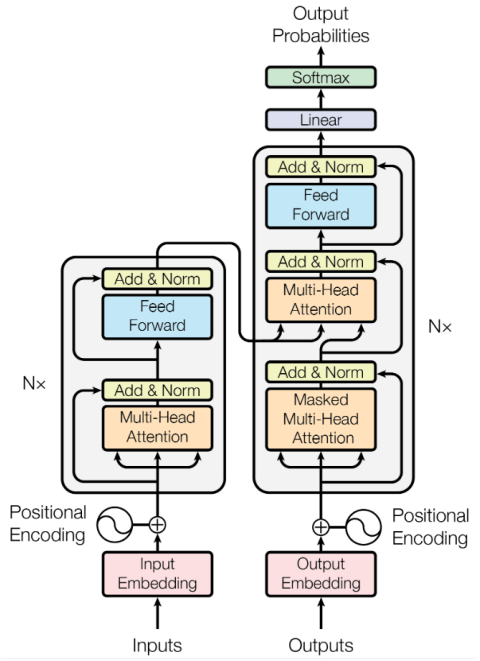
Transformer 中的 Encoder 和 Decoder 的异同点

相同点

1. 在 Transformer 网络中，Encoder 与 Decoder 部分所包含的基础模块的数量是一样，都为 6 个。
2. 每个 Encoder 或者 Decoder 模块都采用 Sub-Layer + Layer Normalization 构成基础层。而 Sub-Layer 都分别用了 Feed Forward Layer 和 Multi-head Attention Layer。

不同点

1. 在 Decoder 模块中，Multi-head Attention Layer 有两层，一层为 Masked Multi-Head Attention 另外一层为 Encoder-Decoder Multi-Head Attention。而 Encoder 模块中只有一层 Multi-Head Attention。
2. 在 Masked Multi-Head Attention 中做了 Mask 操作，即解码序列未到达的位置上的值将会通过 Mask 屏蔽掉。
3. 在 Encoder-Decoder Multi-Head Attention 中，来自最后一层 Encoder 的 K,V 也会被加入计算中。



为什么 transformer 块使用 LayerNorm 而不是 BatchNorm

BatchNorm 操作的是同一个神经元在 Batch 的不同的样本之间做 Normalization；做完 Normalization 之后，网络的学习可以反映 mini-batch 样本的整体分布情况；进而平衡样本之间的差异，使得网络学习更加稳定。而在 Transformer 中，每个输入样本是一个句子，句子有长有短； Transformer 网络中的元素与样本中单个词有对应关系。由于句子同一个 Batch 中，句子长短通常是不一样的，通常较短的句子还会有 padding 补零操作；而且即使是同一个位置，每个词在句子中充当的角色也有很大的差异；所以如果采用 Batch normalization 容易引入比较大的噪声。相比之下 Layer Normalization 中对同一个样本中，同一层网络中的元素进行 normalization 会更加合适。此外，Layer Normalization 也不容易受 Batch size 大小的影响；同时在训练过程中不需要保存同一个神经元在不同样本之间的统计值，大大提高了内存利用率和训练效率。

Self-Attention 中的计算为什么要使用根号 d_k缩放

假设两个 d_k 维向量每个分量都是一个相互独立的服从标准正态分布的随机变量，那么他们的点乘结果会变得很大，并且服从均值为 0，方差为 d_k 的分布， 而很大的点乘会让 Softmax 函数处在梯度很小的区域（我们在讲 Softmax 和 Sigmoid 关系的时候讲过，面对二分类情况的时候 Softmax 就退化成了 Sigmoid 函数；我们知道 Sigmoid 函数在输入数值较大的区域存在梯度饱和的现象），对每一个分量除以 $sqrt(d_k)$ 可以让点乘的方差变成 1。

Bert&GPT

Bert 和 GPT-2 的异同点

共同点：

1. Bert 和 GPT-2 都采用 Transformer 作为底层结构
2. Bert 和 GPT-2 都是通用的语言模型，在很多任务上达到了惊人的效果。

不同点：

1. **模型** - Bert 和 GPT-2 虽然都采用 Transformer，但是 Bert 使用的是 Transformer 的 Encoder，即：Self-Attention，是双向的语言模型；而 GPT-2 用的是 Transformer 中去掉中间 Encoder-Decoder Attention 层的 Decoder，即：Marked Self-Attention，是单向语言模型。
2. **使用** - Bert 是 pre-training + fine-tuning，针对具体的任务时需要做 fine-tuning，而在 GPT-2 中只需要做 pre-training 就可以直接使用。
3. **输入向量** - GPT-2 是 Token Embedding + Position Encoding；Bert 是 Token Embedding + Position Encoding + Segment Encoding.
4. **参数量** - Bert 包含 3 亿个参数；而 GPT-2 有 15 亿个参数。
5. **模型训练** - Bert 在训练时引入 Marked LM 和 Next Sentence Prediction；而 GPT-2 只是单纯用单向语言模型进行训练。
6. **适用任务** - Bert 不能做生成式任务，而 GPT-2 可以。

为什么 Self-attention Model 在长距离序列中如此强大

1）没有因梯度消失而导致的长序列信息“遗忘”的问题；
2）采用全局编码的方式来避免类似于 CNN 所产生的局部依赖的问题；
3）利用 Word Embedding + Position Encoding 相结合的方式来处理文本中的序列信息；
4）利用注意力机制来“动态”地生成不同连接的权重，从而处理变长的信息序列。

Bert 类模型中的绝对位置 embedding 和 相对位置 embedding 怎么理解

绝对位置 Embedding - Learned Positional Embedding

直接对不一样的位置随机初始化一个 Postion embedding，加到 Word Embedding 上输入模型，做为参数进行训练。

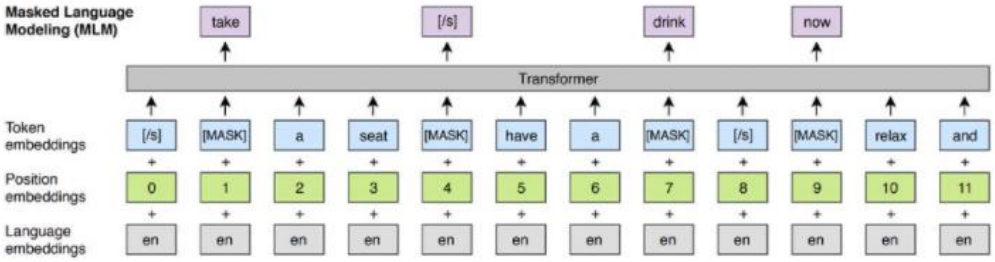
相对位置 Embedding - Sinusoidal Positional Encoding

使用正余弦函数表示绝对位置，经过二者乘积获得相对位置。

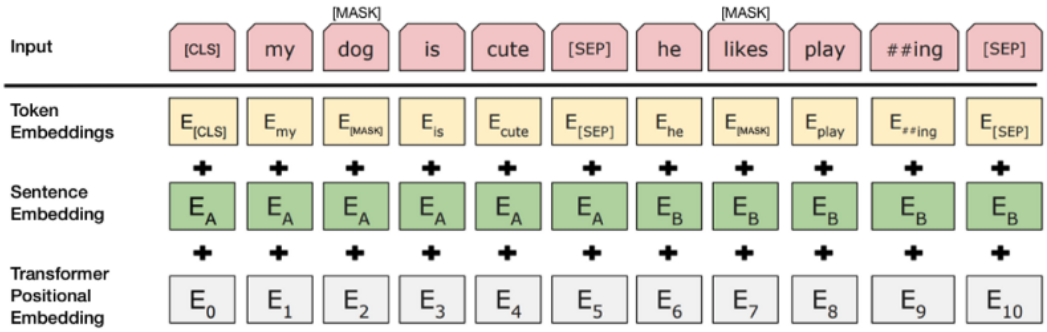
$$PE_{(i,2j)} = \sin(i/10000^{2j/d_{model}})$$
$$PE_{(i,2j+1)} = \cos(i/10000^{2j/d_{model}})$$

Bert 的预训练任务有哪些，各自的作用是什么

Bert 预训练任务有两个 Masked Language Model 和 Next Sentence Prediction。



Masked Language Model (MLM)：在一句话中 mask 掉几个单词然后对 mask 掉的单词做预测。随机将输入中 15%的词遮蔽起来，通过其他词预测被遮盖的词（这就是典型的语言模型），通过迭代训练，可以学习到词的上下文特征、语法结构特征、句法特征等，保证了特征提取的全面性，这对于任何一项 NLP 任务都是尤为重要。



Next Sentence Prediction （NSP）：判断两句话是否为上下文的关系。输入句子 A 和句子 B，判断句子 B 是否是句子 A 的下一句，通过迭代训练，可以学习到句子间的关系，这对于文本匹配类任务显得尤为重要。

Roberta、Albert 分别对 Bert 做了哪些改进

Roberta 的改进：

1）在模型的规模、算力和数据上:
   a. 更多的训练数据，16G-> 160G 更长的训练时间
   b. 更大的 mini-Batch 256 ->8K
2）在训练方法上：
   a. 去掉 NSP 任务 - 使得 Bert 可以学习更长的依赖关系，因为 NSP 任务限制了输入的数据长度只能是两个句子。
   b. 动态 Mask - 对 Mask 的内容进行了自助抽样，丰富了训练数据。

ALBERT 的改进：

1）两种减少参数方法：矩阵分解、参数共享
矩阵分解:在两个大维度之间加入一个小维度，从$O(V*H)$变为$O(V*E+E*H)$，其中$H \gg E$，以达到降维的作用
参数共享：交叉层参数共享
2）Sentence-Order Prediction (SOP)代替 Next Sentence Prediction
NSP 将主题预测（topic prediction）和连贯性预测(coherence prediction)融合起来学习比较困难，而 SOP 将负样本换成了同一篇文章中的两个逆序的句子，进而消除主题预测。通过调整正负样本的获取方式来去除主题识别的影响，使预训练更关注于句子关系一致性预测。
3）移除 Dropout

XLNet 如何实现 Permutation Language Model

PLM (Permutation Language Model) 是 XLNet 的核心思想，首先将句子的 token 随机排列，然后采用 AR（Autoregressive Language Model） 的方式预测句子末尾的单

词，这样 XLNet 即可同时拥有 AE 和 AR(Autoencoder LM) 的优势。它按照从左到右的顺序输入数据，也就是说只能看到预测单词的上文，而我们希望在看到的上文中能够出现下文的单词，这样就能在只考虑上文的情况下实现双向编码，为了到达这样的效果，**XLNet 将句子中的单词随机打乱顺序，这样的话对于单词x_i，它原先的上下文单词就都有可能出现在当前的上文中了**，如下图所示，对于单词 x3，改变原先 1、2、3、4 的排列组合，它的上文中就可能出现 x4，这是虽然模型只考虑上文，但是却包含了原先上下文的信息。不过在实际微调中我们不能直接去改变原始输入，输入的顺序还应该是 1234，所以顺序的改变应该发生在模型内部，这就要用到 Attention mask 机制，通过乘以一个 mask 矩阵的方式来改变序列中单词的顺序。

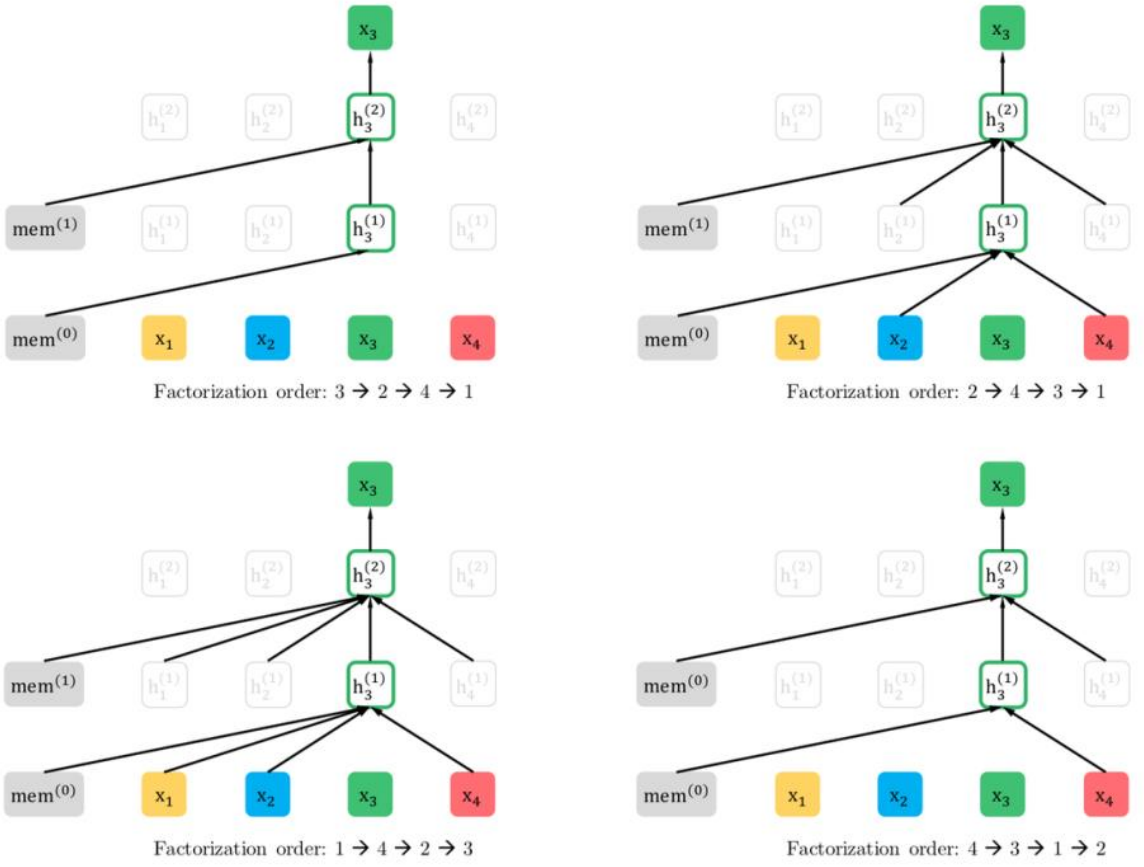


Figure 1: Illustration of the permutation language modeling objective for predicting x_3 given the same input sequence x but with different factorization orders.