大厂 AI 面经

目录

第一篇 2021 年 7 月 5 日 - 7 月 11 日，京东 AI 岗位面试题 4 道!

1.1 介绍逻辑回归，逻辑回归是一个分类算法，那么它是在回归什么呢?

逻辑回归是在数据服从伯努利分布的假设下，通过极大似然的方法，运用梯度下降法来求解参数，从而达到将数据二分类的目的。

逻辑回归就是一种减小预测范围，将预测值限定为[0,1]间的一种广义线性回归模型，解决的是分类问题

1.2 编程题：颜色分类（leetcode 75）

思路一：单指针

对数组进行两次遍历，考虑使用单指针 ptr 进行遍历，第一次遍历中需要把所有的 0 交换到数组的头部，每交换一次，ptr 向右移动一位，直到遍历结束，此时 ptr 之前的元素都为 0；第二次遍历从 ptr 开始遍历，将所有的 1 交换到中间位置，每交换一次，ptr 向后移动一位，直到遍历结束，此时 ptr 之后（包括 ptr）的元素都为 2，排序完成。

代码：

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        ptr = 0
        for i in range(len(nums)):
            if nums[i] == 0:
                nums[i],nums[ptr] = nums[ptr],nums[i]
                ptr +=1
        for i in range(ptr,len(nums)):
            if nums[i] == 1:
                nums[i],nums[ptr] = nums[ptr],nums[i]
                ptr +=1
        return nums
```

时间复杂度：O(n)，其中 nn 是数组 nums 的长度。

空间复杂度：O(1)。

思路二：双指针

相比单指针只需要一次遍历即可完成。需要指针 p0 来交换等于 0 的元素，指针 p1 来交换等于 1 的元素，需要特别注意的如下：

先判断元素是否等于 1，满足等于 1 就进行交换，并将 p1 + 1，再判断是否等于 0 ，如果等于 0 也相应进行交换，另外需要判断 p0 和 p1 的关系，如果满足 p0 < p1，还需要再次进行交换，完成后将 p0 和 p1 同时 +1。

代码如下：

```
1.  class Solution:
2.      def sortColors(self, nums: List[int]) -> None:
3.          """
4.          Do not return anything, modify nums in-place instead.
5.          """
6.          p0 = p1 = 0
7.          for i in range(len(nums)):
8.              if nums[i] == 1:
9.                  nums[i],nums[p1] = nums[p1],nums[i]
10.                 p1 +=1
11.             elif nums[i] == 0:
12.                 nums[i],nums[p0] = nums[p0],nums[i]
13.                 if p0 < p1:
14.                     nums[i],nums[p1] = nums[p1],nums[i]
15.                 p0 += 1
16.                 p1 += 1
17.         return nums
```

时间复杂度：O(n)，其中 nn 是数组 nums 的长度。

空间复杂度：O(1)。

1.3 GBDT 了解吗？基分类器用的什么？分类时也是用的那个吗？

GBDT 是梯度提升决策树，是一种基于 Boosting 的算法，采用以决策树为基学习器的加法模型，通过不断拟合上一个弱学习器的残差，最终实现分类或回归的模型。关键在于利用损失函数的负梯度在当前模型的值作为残差的近似值，从而拟合一个回归树。

GBDT 的基分类器用的是决策树，分类时也是用的决策树。

对于分类问题：常使用指数损失函数；对于回归问题：常使用平方误差损失函数（此时，其负梯度就是通常意义的残差），对于一般损失函数来说就是残差的近似。

1.4 XGBoost 相对 GBDT 原理上有哪些改进。

改进主要为以下方面：

传统的 GBDT 以 CART 树作为基学习器，XGBoost 还支持线性分类器，这个时候 XGBoost 相当于 L1 和 L2 正则化的逻辑斯蒂回归（分类）或者线性回归（回归）；

传统的 GBDT 在优化的时候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，得到一阶和二阶导数；

XGBoost 在代价函数中加入了正则项，用于控制模型的复杂度。从权衡方差偏差来看，它降低了模型的方差，使学习出来的模型更加简单，放置过拟合，这也是 XGBoost 优于传统 GBDT 的一个特性；

shrinkage（缩减），相当于学习速率（XGBoost 中的 eta）。XGBoost 在进行完一次迭代时，会将叶子节点的权值乘上该系数，主要是为了削弱每棵树的影响，让后面有更大的学习空间。（GBDT 也有学习速率）；

列抽样：XGBoost 借鉴了随机森林的做法, 支持列抽样, 不仅防止过 拟合，还能减少计算；

对缺失值的处理: 对于特征的值有缺失的样本，XGBoost 还可以自动 学习出它的分裂方向；

XGBoost 工具支持并行。Boosting 不是一种串行的结构吗?怎么并行 的？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（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开多线程进行。

第二篇 2021 年 7 月 5 日 - 7 月 11 日，阿里 AI 岗位面试题 6 道!

2.1 常见的损失函数，常见的激活函数，ELU 函数 了解吗?

常见的损失函数：0-1 损失函数，绝对值损失函数，log 对数损失函数，平方损失函数，指数损失函数，hinge 损失函数，交叉熵损失函数等。

0-1 损失函数

$$L(Y, f(X)) = \begin{cases} 1, Y \neq f(X) \\ 0, Y = f(X) \end{cases}$$

绝对值损失函数

$$L(Y, f(x)) = |Y - f(x)|$$

log 对数损失函数

$$L(Y, P(Y|X)) = -logP(Y|X)$$

平方损失函数

$$L(Y|f(X)) = \sum_{N}(Y - f(X))^2$$

指数损失函数

$$L(Y|f(X)) = exp[-yf(x)]$$

hinge 损失函数

$$L(y, f(x)) = max(0, 1 - yf(x))$$

交叉熵损失函数

$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)]$$

常见的激活函数有：Sigmoid、Tanh、ReLU、Leaky ReLU

Sigmoid 函数：

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

特点：

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：

缺点 1：在深度神经网络中梯度反向传递时导致梯度消失，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。

缺点 2：Sigmoid 的 output 不是 0 均值（即 zero-centered）。

缺点 3：其解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增加训练时间。

Tanh 函数：

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

特点：它解决了 Sigmoid 函数的不是 zero-centered 输出问题，收敛速度比 sigmoid 要快，然而，梯度消失（gradient vanishing）的问题和幂运算的问题仍然存在。

ReLU 函数：

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

特点：

1.ReLu 函数是利用阈值来进行因变量的输出，因此其计算复杂度会比剩下两个函数低（后两个函数都是进行指数运算）

2.ReLu 函数的非饱和性可以有效地解决梯度消失的问题，提供相对宽的激活边界。

3.ReLU 的单侧抑制提供了网络的稀疏表达能力。

ReLU 的局限性:在于其训练过程中会导致神经元死亡的问题。

这是由于函数 f(x)=max(0,x)导致负梯度在经过该 ReLU 单元时被置为 0，且在之后也不被任何数据激活，即流经该神经元的梯度永远为 0，不对任何数据产生响应。在实际训练中，如果学习率（Learning Rate）设置较大，会导致超过一定比例的神经元不可逆死亡，进而参数梯度无法更新，整个训练过程失败。

Leaky ReLu 函数：

$$f(x) = \begin{cases} x & (x > 0) \\ ax & (x \leqslant 0) \end{cases}$$

LReLU 与 ReLU 的区别在于， 当 z<0 时其值不为 0，而是一个斜率为 a 的线性函数，一般 a 为一个很小的正常数， 这样既实现了单侧抑制，又保留了部分负梯度信息以致不完全丢失。但另一方面，a 值的选择增加了问题难度，需要较强的人工先验或多次重复训练以确定合适的参数值。

基于此，参数化的 PReLU（Parametric ReLU）应运而生。它与 LReLU 的主要区别是将负轴部分斜率 a 作为网络中一个可学习的参数，进行反向传播训练，与其他含参数网络层联合优化。而另一个 LReLU 的变种增加了“随机化”机制，具体地，在训练过程中，斜率 a 作为一个满足某种分布的随机采样；测试时再固定下来。Random ReLU（RReLU）在一定程度上能起到正则化的作用。

ELU 函数：

$$f(x)=\begin{cases} x & ,x>0 \\ \alpha(e^{x}-1) & ,x\leq 0 \end{cases}$$

ELU 函数是针对 ReLU 函数的一个改进型，相比于 ReLU 函数，在输入为负数的情况下，是有一定的输出的，而且这部分输出还具有一定的抗干扰能力。这样可以消除 ReLU 死掉的问题，不过还是有梯度饱和和指数运算的问题。

2.2 分类问题为什么不用 MSE，而是要用交叉熵。

LR 的基本表达形式如下：

$$h_{\theta}(x)=g\left(\theta^{T}x\right)=\frac{1}{1+e^{-\theta^{T}x}}$$

使用交叉熵作为损失函数的梯度下降更新求导的结果如下：

首先得到损失函数如下：

$$C=\frac{1}{n}\sum[y\,\mathrm{In}\,\hat{y}+(1-y)\,\mathrm{In}(1-\hat{y})]$$

如果我们使用 MSE 作为损失函数的话，那损失函数以及求导的结果如下所示：

$$C=\frac{(y-\hat{y})^{2}}{2}$$

$$\frac{\partial C}{\partial w}=(\hat{y}-y)\sigma'(z)(x)$$

使用平方损失函数，会发现梯度更新的速度和 sigmod 函数本身的梯度是很相关的。sigmod 函数在它在定义域内的梯度都不大于 0.25。这样训练会非常的慢。使用交叉熵的话就不会出现这样的情况，它的导数就是一个差值，误差大的话更新的就快，误差小的话就更新的慢点，这正是我们想要的。

在使用 Sigmoid 函数作为正样本的概率时，同时将平方损失作为损失函数，这时所构造出来的损失函数是非凸的，不容易求解，容易得到其局部最优解。如果使用极大似然，其目标函数就是对数似然函数，该损失函数是关于未知参数的高阶连续可导的凸函数，便于求其全局最优解。（关于是否是凸函数，由凸函数的定义得，对于一元函数，其二阶导数总是非负，对于多元函数，其 Hessian 矩阵（Hessian 矩阵是由多元函数的二阶导数组成的方阵）的正定性来判断。如果 Hessian 矩阵是半正定矩阵）

2.3 F1score 的计算公式。

要计算 F1score，首先要计算 Precision 和 Recall，公式如下：

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

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

$$F\text{-}score = \frac{2Precision * Recall}{Precision + Recall}$$

2.4 FM vs SVM 的比较。

FM 和 SVM 最大的不同，在于特征组合时权重的计算方法

SVM 的二元特征交叉参数是独立的，而 FM 的二元特征交叉参数是两个 k 维的向量 v_i，v_j，交叉参数不是独立的，而是互相影响的

FM 可以在原始形式下进行优化学习，而基于 kernel 的非线性 SVM 通常需要在对偶形式下进行

FM 的模型预测与训练样本独立，而 SVM 则与部分训练样本有关，即支持向量.

FM 模型有两个优势：

在高度稀疏的情况下特征之间的交叉仍然能够估计，而且可以泛化到未被观察的交叉参数的学习和模型的预测的时间复杂度是线性的

2.5 随机森林的随机性。

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。有了这 2 个随机的保证，随机森林就不会产生过拟合的现象了。

2.6 编程题：跳跃游戏(leetcode55)

思路：贪心算法

依次遍历数组中的每一个位置，并实时维护最远可以到达的位置 rightMost，注意当前遍历到的位置 i 要在最远可以到达的位置范围内（也就是满足 i <= rightMost），当 rightMost 大于或等于 数组最后一个位置时，就表示可以到达，返回 True，否则就要返回 False。

代码如下：

```
1.  class Solution:
2.      def canJump(self, nums: List[int]) -> bool:
3.          n = len(nums)
4.          rightMost = 0
5.          for i in range(n):
6.              if i <= rightMost:
7.                  rightMost = max(rightMost, i + nums[i])
8.                  if rightMost >= n-1:
9.                      return True
10.         return False
```

时间复杂度：O(n)，其中 n 为数组的大小。只需要访问 nums 数组一遍，共 n 个位置。

空间复杂度：O(1)，不需要额外的空间开销。

第三篇 2021 年 6 月 9 日，CVTE NLP 岗位面试题 9 道!

3.1 讲一下改进的 tf-idf

TF-IDF 中的 IDF 是一种试图抑制噪声的加权，单纯的以为文本频率小的单词就越重要，文本频率越大的单词就越无用，这一方式会在同类语料库中存在巨大弊端，一些同类文本的关键词容易被掩盖。例如：语料库 D 中教育类文章偏多，而文本 j 是一篇属于教育类的文化在那个，那么教育类相关词语的 IDF 值就会较小，使提取本文关键词的召回率更低。

改进方法：TF−IWF (Term Frequency-Inverse Word Frequency)

此处的TF与$TF-IDF$中意义一样，表示词频:

$$tf_{ij}=\frac{n_{i,j}}{\sum_k n_{k,j}}$$

上式中分子$n_{i,j}$表示词语t_i在文本j中的频数，分母$\sum_k n_{k,j}$表示文档j中所有词汇量总和，即是说:

$$TF_w=\frac{\text{给定词}w\text{在当前文章出现的次数}}{\text{当前文章中的总词量}}$$

不同之处在于IWF部分，定义为:

$$iwf_i=log\frac{\sum_{i=1}^m nt_i}{nt_i}$$

上式中分子 $\sum_{i=1}^m nt_i$ 表示语料库中所有词语的频数之和，分母 nt_i 表示词语 t_i 在语料库中的总频数，即:

$$IWF_i=\frac{\text{语料库中所有词语的频数}}{\text{给定词}w\text{在语料库中出现的频数和}}$$

因此，$TF-IWF$ 定义为:

$$TF-IWF_{i,j}\rightarrow tf_{i,j}\times iwf_i=\frac{n_{i,j}}{\sum_k n_{k,j}}\times log\frac{\sum_{i=1}^m nt_i}{nt_i}$$

3.2 讲一下 k-means 与谱聚类

聚类算法属于无监督的机器学习算法，即没有类别标签 y，需要根据数据特征将相似的数据分为一组。K-means 聚类算法即随机选取 k 个点作为聚类中心，计算其他点与中心点的距离，选择距离最近的中心并归类，归类完成后计算每类的新中心点，重新计算每个点与中心点的聚类并选择距离最近的归类，重复此过程，直到中心点不再变化。

谱聚类的思想是将样本看作顶点，样本间的相似度看作带权的边，从而将聚类问题转为图分割问题：找到一种图分割的方法使得连接不同组的边的权重尽可能低（这意味着组间相似度要尽可能低），组内的边的权重尽可能高（这意味着组内相似度要尽可能高），从而达到聚类的目的。

3.3 蒸馏的思想，为什么要蒸馏?

知识蒸馏就是将已经训练好的模型包含的知识，蒸馏到另一个模型中去。具体来说，知识蒸馏，可以将一个网络的知识转移到另一个网络，两个网络可以是同构或者异构。做法是先训练一个 teacher 网络，然后使用这个 teacher 网络的输出和数据的真实标签去训练 student 网络。

在训练过程中，我们需要使用复杂的模型，大量的计算资源，以便从非常大、高度冗余的数据集中提取出信息。在实验中，效果最好的模型往往规模很大，甚至由多个模型集成得到。而大模型不方便部署到服务中去，常见的瓶颈如下:

推断速度慢

对部署资源要求高(内存，显存等)，在部署时，我们对延迟以及计算资源都有着严格的限制。

因此，模型压缩（在保证性能的前提下减少模型的参数量）成为了一个重要的问题。而”模型蒸馏“属于模型压缩的一种方法。

3.4 有哪些蒸馏方式? 蒸馏中的学生模型是?

以 Bert 模型举例：

Logit Distillation

Beyond Logit Distillation：TinyBert

Curriculum Distillation：

Dynamic Early Exit：FastBert。

3.5 python 在内存上做了哪些优化?

python 通过内存池来减少内存碎片化，提高执行效率。主要通过引用计数来完成垃圾回收，通过标记-清除解决容器对象循环引用造成的问题，通过分代回收提高垃圾回收的效率。

3.6 怎么节省内存?

手动回收不需要用的变量；

将数值型数据转化为 32 位或 16 位（对数据类型进行限制）

代码示例如下：

```
def reduce_mem_usage(props):
    # 计算当前内存
    start_mem_usg = props.memory_usage().sum() / 1024 ** 2
    print("Memory usage of the dataframe is :", start_mem_usg, "MB")

    # 哪些列包含空值，空值用-999 填充。why：因为 np.nan 当做 float 处理
    NAlist = []
    for col in props.columns:
        # 这里只过滤了 object 格式，如果你的代码中还包含其他类型，请一并过滤
        if (props[col].dtypes != object):

            print("**************************")
            print("columns: ", col)
            print("dtype before", props[col].dtype)

            # 判断是否是 int 类型
            isInt = False
            mmax = props[col].max()
            mmin = props[col].min()

            # Integer does not support NA, therefore Na needs to be filled
            if not np.isfinite(props[col]).all():
                NAlist.append(col)
                props[col].fillna(-999, inplace=True) # 用-999 填充

            # test if column can be converted to an integer
            asint = props[col].fillna(0).astype(np.int64)
            result = np.fabs(props[col] - asint)
            result = result.sum()
            if result < 0.01: # 绝对误差和小于 0.01 认为可以转换的，要根据 task 修改

                isInt = True

            # make interger / unsigned Integer datatypes
            if isInt:
                if mmin >= 0: # 最小值大于 0，转换成无符号整型
                    if mmax <= 255:
                        props[col] = props[col].astype(np.uint8)
                    elif mmax <= 65535:
                        props[col] = props[col].astype(np.uint16)
                    elif mmax <= 4294967295:
                        props[col] = props[col].astype(np.uint32)
                    else:
                        props[col] = props[col].astype(np.uint64)
```

```
                else: # 转换成有符号整型
                    if mmin > np.iinfo(np.int8).min and mmax < np.iinfo(np.int
8).max:
                        props[col] = props[col].astype(np.int8)
                    elif mmin > np.iinfo(np.int16).min and mmax < np.iinfo(np.
int16).max:
                        props[col] = props[col].astype(np.int16)
                    elif mmin > np.iinfo(np.int32).min and mmax < np.iinfo(np.
int32).max:
                        props[col] = props[col].astype(np.int32)
                    elif mmin > np.iinfo(np.int64).min and mmax < np.iinfo(np.
int64).max:
                        props[col] = props[col].astype(np.int64)
            else: # 注意：这里对于 float 都转换成 float16，需要根据你的情况自己更
改
                props[col] = props[col].astype(np.float16)

            print("dtype after", props[col].dtype)
            print("********************************")
    print("___MEMORY USAGE AFTER COMPLETION:___")
    mem_usg = props.memory_usage().sum() / 1024**2
    print("Memory usage is: ",mem_usg," MB")
    print("This is ",100*mem_usg/start_mem_usg,"% of the initial size")
```

3.7 pandas 库怎么读取超大型文件?

可以采取分块读取数据的方式。

代码示例如下：

```
data_path = r'E:\python\Study\BiGData\demo.csv'
def read_bigfile(path):
    # 分块，每一块是一个 chunk，之后将 chunk 进行拼接
    df = pd.read_csv(path, engine='python', encoding='gbk', iterator=True)

    loop = True
    chunkSize = 10000
    chunks = []
    while loop:
        try:
            chunk = df.get_chunk(chunkSize)
            chunks.append(chunk)
        except StopIteration:
```

```
            loop = False
            print("Iteration is stopped.")
    df = pd.concat(chunks, ignore_index=True)
after_df = read_bigfile(path = data_path)
```

3.8 无重复字符的最长子串

该题为 leetcode-3，难度：【中等】

方法：双指针 + sliding window

定义两个指针 start 和 end 得到 sliding window；

start 初始为 0，用 end 线性遍历每个字符，用 recod 记录下每个字母最新出现的下标；

两种情况：一种是新字符没有在 record 中出现过，表示没有重复，一种是新字符 char 在 record 中出现过，说明 start 需要更新，取 start 和 record[char]+1 中的最大值作为新的 start。

需要注意的是：两种情况都要对 record 进行更新，因为是新字符没在 record 出现过的时候需要添加到 record 中，而对于出现过的情况，也需要把 record 中对应的 value 值更新为新的下标。

代码:

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        record = {}
        start,res = 0,0
        for end in range(len(s)):
            if s[end] in record:
                start = max(start, record[s[end]] + 1)
            record[s[end]] = end
            res = max(res, end - start + 1)
        return res
```

时间复杂度：O(n)

空间复杂度：O(1)

3.9 判断链表是否有环、链表环的入口

判断链表是否有环为 leetcode-141 题

提供两种解题方法，如下：

方法一：哈希表

遍历所有节点，每次遍历一个节点时，判断该节点此前是否被访问过。

如果被访问过，说明该链表是环形链表，并返回 True，如果没有，则将该节点加入到哈希表中，遍历完成即可。

代码如下：

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        seen = set()
        while head:
            if head in seen:
                return True
            seen.add(head)
            head = head.next
        return False
```

时间复杂度：O(n)

空间复杂度：O(n)

方法二：快慢指针

定义两个指针，一快一慢，满指针每一移动一步，快指针每次移动两步，由于快指针比慢指针慢，如果链表有环，则快指针一定会和慢指针相遇。

代码如下：

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        fast = slow = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if fast == slow:
                return True
        return False
```

时间复杂度：O(N)

空间复杂度：O(1)

链表环的入口 为 leetcode-142 题

解题思路：使用快慢指针

首先，我们要做以下设定，设链表共有 a + b 个节点，其中链表头部到链表入口（不计链表入口节点）有 a 个节点，链表环有 b 个节点，同时设快指针走了 f 步，慢指针走了 s 步；

显然，由于快指针是慢指针的两倍，因此有：f = 2s

如果两个指针第一次相遇，则满足：f = s + nb（n 表示 n 个环），也就是 fast 比 slow 多走了 n 个环的长度；

由上可得：当两个指针第一次相遇，有 s = nb, f = 2nb

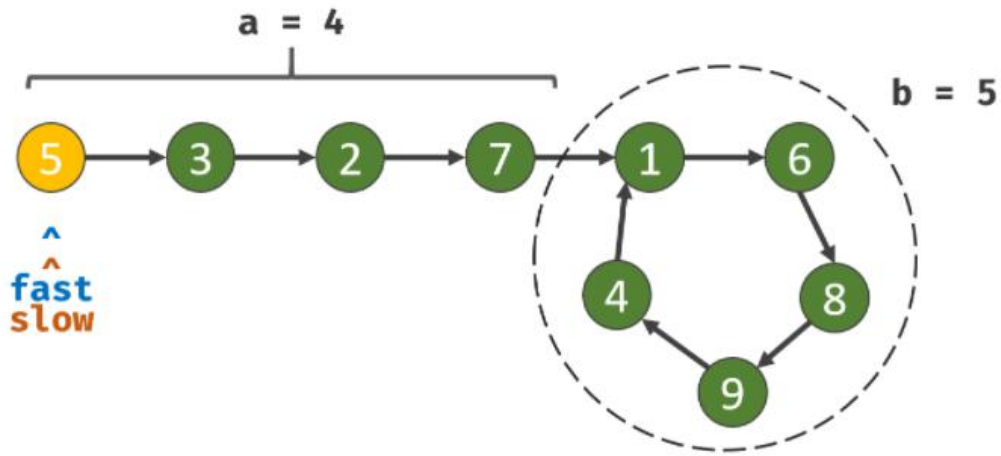
现在需要解决的问题，我们如何知道 a 的大小?

假设现在两指针第一次相遇：

对慢指针来说，如果让慢指针走到链表入口节点，需要 a + nb 步，而在相遇时，已经走了 nb 步了，因此，再让慢指针走 a 步就可以走到链表入口节点了；

对快指针来说，如果让快指针从头指针开始走 a 步，快指针也正好走到链表入口节点；

由此可见，两指针发生第二次相遇的时候，刚好都在链表入口处。



代码如下：

```
1.  class Solution:
2.      def detectCycle(self, head: ListNode) -> ListNode:
3.          fast = slow = head
4.          while True:
5.              if not (fast and fast.next):return
6.              fast = fast.next.next
7.              slow = slow.next
8.              if fast == slow:break
9.          fast = head
10.         while fast != slow:
11.             fast = fast.next
12.             slow = slow.next
13.         return fast
```

时间复杂度 O(N)

空间复杂度 O(1)

第四篇 2021 年 6 月 22 日，vivo2022 提前批数据挖掘面试题 2 道!

4.1、输入一个数组，统计数组中有多少个数是 7 的倍数或者含有数字 7

判断一个数是否是 7 的倍数可以直接用取余的方法，判断一个数中是否含有数字 7，这里提供两种方法：一种是将数字转换成字符串，用 in 进行判断；另一种是将数字转换成字符串，用 find 方法，如果不包含会返回 -1。

代码如下：

方法一：

```
def get_num_1(nums):
    res = 0
    for i in nums:
        if i % 7 ==0 or str(i).find('7') != -1:
            res +=1
    return res
nums = [7,21,34,777,666,888,63]
print(get_num_1(nums))
```

方法二：

```
def get_num_2(nums):
    res = 0
    for i in nums:
        if i % 7 ==0 or '7' in str(i):
            res +=1
    return res
nums = [7,21,34,777,666,888,63]
print(get_num_2(nums))
```

4.2 0-1 背包问题

题目描述：给定物品的重量 weights=[1, 2, 5, 6, 7]， 对应的价值 values=[1, 6, 18, 22, 28]， 背包能装的最大重量为 capicity=11。问：我们用这个背包装什么物品能获得最大价值？ 注意：每件物品只有一件。并且最终重量不能超过背包所能承载的重量。

本题解析参考：https://www.pianshen.com/article/2073277310/

分析：

首先，说明一下，本题采用动态规划， 因为问题的本身含最优子结构。 我们先给出转态转移方程：

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1,w] & \text{if } w_i > w, \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i. \end{cases}$$

c(i, w) 表示包容量为 w 时，考虑前 i 个物品所能获得的最大价值。。 i 表示第 i 个物品，w 表示包容量

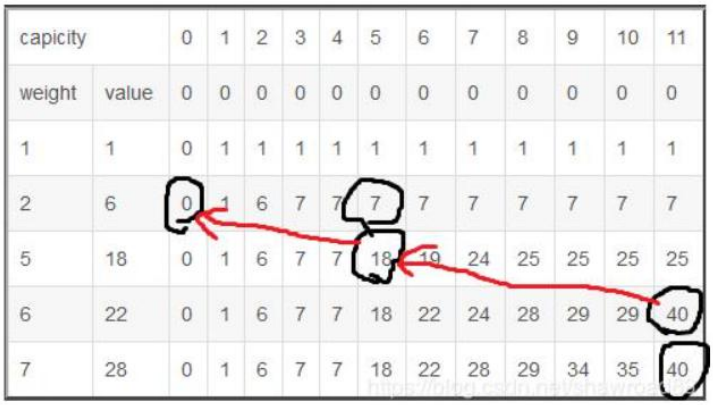
第一种情况： 当前物品的重量超过了包的承载量，显然装不上，那它当前的最大价值就是原有包中的价值（不装这个物品时的最大价值）。

第二种情况：当前物品的重量没有包承载量大。则说明当前这个物品可以装进去。 那我们就得考虑了： 装这个物品价值大还是不装这个物品价值大？ 从两种情况中选最大的。

capicity		0	1	2	3	4	5	6	7	8	9	10	11
weight	value	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	6	7	7	18	19	24	25	25	25	25
6	22	0	1	6	7	7	18	22	24	28	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	40

右下角的 40, 就是我们所能获得的最大价值

接着，我们还需要输出获得最大价值的时候，我们拿了什么物品。



从右下角的 40 开始， 背包容量为 11，由于 40 与上一行的 40 相等， 说明我们没有装第五个物品（因为当背包容量为 11，装第四个物品的时候，价值已经到达了 40）。 紧接着， 在背包容量为 11 时，对应第三个物品，最大价值是 25， 所以，我们装了第四个物品。 此时，我们的背包容量变为 W-w4 即：11-6=5。 所以，对于第三个物品，我们直接考虑在背包容量为 5 的时候， 可以看出价值是 18，考虑有没有装第三个？ 因为背包容量为 5 时，第二个物品所对应的最大价值为 7. 所以我们装了第三个物品。 接在：W 剩 - w3 = 5-5 = 0 。 我们就可以得出背包中装了第三个和第四个物品。

代码实现：

```python
#  01 背包问题
def bag_01(weights, values, capicity):
    n = len(values)
    f = [[0 for j in range(capicity+1)] for i in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, capicity+1):
            f[i][j] = f[i-1][j]
            if j >= weights[i-1] and f[i][j] < f[i-1][j-weights[i-1]] + values[i-1]:
                f[i][j] = f[i-1][j-weights[i-1]] + values[i-1]
    return f

def show(capicity, weights, f):
    n = len(weights)
    print("最大价值:", f[n][capicity])
    x = [False for i in range(n)]
    j = capicity
    for i in range(n, 0, -1):
        if f[i][j] > f[i-1][j]:
            x[i-1] = True
            j -= weights[i-1]
    print("背包中所装物品为:")
    for i in range(n):
        if x[i]:
            print("第{}个".format(i+1))

if __name__ == '__main__':
    # weights 指的是物品的重量
    # values 指的是物品的价值
    # capicity 指的是袋子能装的重量
    n = 5
    weights = [1, 2, 5, 6, 7]
    values = [1, 6, 18, 22, 28]
    capicity = 11
    m = bag_01(weights, values, capicity)
    # 打印矩阵
    for i in range(len(m)):
        print(m[i])

    # 接下来输出要装的物品
    show(capicity, weights, m)
```

第五篇 2021 年 6 月 30 日，明略科技 AI 岗位面试题 7 道!

5.1 熵，交叉熵的概念

在信息论和概率统计中，熵是表示随机变量不确定的度量，随机边量 X 的熵定义如下：

$$H(X) = -\sum_{x} p(x) \log p(x)$$

熵只依赖于 X 的分布，与 X 的取值无关。

条件熵 H(Y|X) 表示在已知随机变量 X 的条件下随机变量 Y 的不确定性，H(Y|X)定义为在给定条件 X 下，Y 的条件概率分布的熵对 X 的数学期望：

$$H(Y \mid X) = \sum_{i=1}^{n} p_i H(Y \mid X = x_i)$$

5.2 逻辑回归损失函数，并推导梯度下降公式

逻辑回归损失函数及梯度推导公式如下：

$$\begin{aligned} L(w) &= \sum_{i} \left(y_i * \operatorname{logh}\left(x_i\right) + \left(1 - y_i\right) * \log\left(1 - h\left(x_i\right)\right)\right) \\ &= \sum_{i} y_i \left(\operatorname{logh}\left(x_i\right) - \log\left(1 - h\left(x_i\right)\right)\right) + \log\left(1 - h\left(x_i\right)\right) \\ &= \sum_{i} y_i \log \frac{h\left(x_i\right)}{1 - h\left(x_i\right)} + \log\left(1 - h\left(x_i\right)\right) \\ &= \sum_{i} y_i \left(w^T x_i\right) + \log\left(1 - \frac{1}{1 + e^{-w x_i}}\right) \\ &= \sum_{i} \left(y_i * \left(w^T x_i\right) - \log\left(1 + e^{w^T x_i}\right)\right) \end{aligned}$$

求导：

$$\begin{aligned} \frac{dL}{dw} &= yx - \frac{1}{1 + e^{w^T x}} * e^{w^T x} * x \\ &= x(y - h(x)) \end{aligned}$$

5.3 归一化和标准化区别

归一化公式：

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

标准化公式：

$$x' = \frac{x - \bar{x}}{\sigma}$$

归一化和标准化的区别：归一化是将样本的特征值转换到同一量纲下把数据映射到[0,1]或者[-1, 1]区间内，仅由变量的极值决定，因区间放缩法是归一化的一种。标准化是依照特征矩阵的列处理数据，其通过求 z-score 的方法，转换为标准正态分布，和整体样本分布相关，每个样本点都能对标准化产生影响。它们的相同点在于都能取消由于量纲不同引起的误差；都是一种线性变换，都是对向量 X 按照比例压缩再进行平移。

5.4 knn 算法的思想

在训练集中数据和标签已知的情况下，输入测试数据，将测试数据的特征与训练集中对应的特征进行相互比较，找到训练集中与之最为相似的前 K 个数据，则该测试数据对应的类别就是 K 个数据中出现次数最多的那个分类，其算法的描述为：

1）计算测试数据与各个训练数据之间的距离；

2）按照距离的递增关系进行排序；

3）选取距离最小的 K 个点；

4）确定前 K 个点所在类别的出现频率；

5）返回前 K 个点中出现频率最高的类别作为测试数据的预测分类。

5.5 knn 的 k 设置的过大会有什么问题

KNN 中的 K 值选取对 K 近邻算法的结果会产生重大影响。如果选择较小的 K 值，就相当于用较小的领域中的训练实例进行预测，“学习”近似误差（近似误差：可以理解为对现有训练集的训练误差）会减小，只有与输入实例较近或相似的训练实例才会对预测结果起作用，与此同时带来的问题是“学习”的估计误差会增大，换句话说，K 值的减小就意味着整体模型变得复杂，容易发生过拟合；

如果选择较大的 K 值，就相当于用较大领域中的训练实例进行预测，其优点是可以减少学习的估计误差，但缺点是学习的近似误差会增大。这时候，与输入实例较远（不相似的）训练实例也会对预测器作用，使预测发生错误，且 K 值的增大就意味着整体的模型变得简单。

在实际应用中，K 值一般取一个比较小的数值，例如采用交叉验证法来选择最优的 K 值。经验规则：k 一般低于训练样本数的平方根。

5.6 gbdt 和 bagging 的区别，样本权重为什么会改变?

gbdt 是基于 Boosting 的算法。

Bagging 和 Boosting 的区别：

1）样本选择上：
Bagging：训练集是在原始集中有放回选取的，从原始集中选出的各轮训练集之间是独立的。

Boosting：每一轮的训练集不变，只是训练集中每个样例在分类器中的权重发生变化。而权值是根据上一轮的分类结果进行调整。

2）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，错误率越大则权重越大。

3）预测函数：
Bagging：所有预测函数的权重相等。

Boosting：每个弱分类器都有相应的权重，对于分类误差小的分类器会有更大的权重。

4）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，因为后一个模型参数需要前一轮模型的结果。

Boosting 中样本权重发生改变的原因：

通过提高那些在前一轮被弱分类器分错样例的权值，减小前一轮分对样例的权值，来使得分类器对误分的数据有较好的效果。

5.7 梯度下降的思想

梯度下降是一种非常通用的优化算法，能够为大范围的问题找到最优解。梯度下降的中心思想就是迭代地调整参数从而使损失函数最小化。

假设你迷失在山上的迷雾中，你能感觉到的只有你脚下路面的坡度。快速到达山脚的一个策略就是沿着最陡的方向下坡，这就是梯度下降的做法。即通过测量参数向量 θ 相关的损失函数的局部梯度，并不断沿着降低梯度的方向调整，直到梯度降为 0 ，达到最小值。

梯度下降公式如下：

$$\theta := \theta - \eta \nabla loss$$

对应到每个权重公式为：

$$w_i := w_i - \eta \frac{\partial loss}{\partial w_i}$$

第六篇 2021 年 6 月底 - 7 月 3 日，拼多多搜索广告算法岗位面试题 2 道！

6.1 算法题：leetcode 687. 最长同值路径

方法：递归

可以将任何路径（具有相同值的节点）看作是最多两个从其根延伸出的箭头。具体地说，路径的根将是唯一节点，因此该节点的父节点不会出现在该路径中，而箭头将是根在该路径中只有一个子节点的路径。然后，对于每个节点，我们想知道向左延伸的最长箭头和向右延伸的最长箭头是什么？我们可以用递归来解决这个问题。

令 arrow_length(node) 为从节点 node 延伸出的最长箭头的长度。如果 node.Left 存在且与节点 node 具有相同的值，则该值就会是 1 + arrow_length(node.left)。在 node.right 存在的情况下也是一样。

当我们计算箭头长度时，候选答案将是该节点在两个方向上的箭头之和。我们将这些候选答案记录下来，并返回最佳答案。

代码如下：

```
class Solution(object):
    def longestUnivaluePath(self, root):
        self.ans = 0

        def arrow_length(node):
            if not node: return 0
            left_length = arrow_length(node.left)
            right_length = arrow_length(node.right)
            left_arrow = right_arrow = 0
            if node.left and node.left.val == node.val:
                left_arrow = left_length + 1
            if node.right and node.right.val == node.val:
                right_arrow = right_length + 1
            self.ans = max(self.ans, left_arrow + right_arrow)
            return max(left_arrow, right_arrow)

        arrow_length(root)
        return self.ans
```

时间复杂度：O(N)，其中 N 是树中节点数。

空间复杂度：O(H)，其中 H 是树的高度。

6.2 算法题：leetcode 322. 零钱兑换

完全背包问题——填满容量为 amount 的背包最少需要多少硬币

dp[j]代表含义：填满容量为 j 的背包最少需要多少硬币

初始化 dp 数组：因为硬币的数量一定不会超过 amount，而 amount <= 10^4，因此初始化数组值为 10001；dp[0] = 0

转移方程：dp[j] = min(dp[j], dp[j - coin] + 1)

当前填满容量 j 最少需要的硬币 = min(之前填满容量 j 最少需要的硬币, 填满容量 j - coin 需要的硬币 + 1 个当前硬币)

返回 dp[amount]，如果 dp[amount]的值为 10001 没有变过，说明找不到硬币组合，返回-1

代码如下：

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [0] + [10001] * amount
        for coin in coins:
            for j in range(coin, amount + 1):
                dp[j] = min(dp[j], dp[j - coin] + 1)
        return dp[-1] if dp[-1] != 10001 else -1
```

时间复杂度：O(n * amount)

空间复杂度：O(amount)

第七篇 2021 年 7 月初，360 校招提前批算法岗位面试题 4 道！

7.1 非递归的二叉树中序遍历

该题为 Leetcode-94：二叉树的中序遍历

方法：迭代

需要一个栈的空间，先用指针找到每颗子数的最左下角，然后进行进出栈的操作。

代码如下：

```
1.  class Solution:
2.      def inorderTraversal(self, root: TreeNode) -> List[int]:
3.          if not root:
4.              return []
5.          stack = []
6.          res = []
7.          cur = root
8.          while stack or cur:
9.              while cur:
10.                 stack.append(cur)
11.                 cur = cur.left
12.             cur = stack.pop()
13.             res.append(cur.val)
14.             cur = cur.right
15.         return res
```

时间复杂度：O(n)，n 为树的节点个数

空间复杂度：O(h)，h 为树的高度

7.2 lightgbm 相较于 xgboost 的优势

优点：直方图算法—更高（效率）更快（速度）更低（内存占用）更泛化（分箱与之后的不精确分割也起到了一定防止过拟合的作用）；

缺点：直方图较为粗糙，会损失一定精度，但是在 gbm 的框架下，基学习器的精度损失可以通过引入更多的 tree 来弥补。

总结如下：

- 更快的训练效率
- 低内存使用
- 更高的准确率
- 支持并行化学习

- 可处理大规模数据
- 支持直接使用 category 特征

7.3 wide & deep 模型 wide 部分和 deep 部分分别侧重学习什么信息

Wide&Deep 模型的主要思路正如其名，是由单层的 Wide 部分和多层的 Deep 部分组成的混合模型。其中，Wide 部分的主要作用是让模型具有较强的“记忆能力”，“记忆能力”可以被理解为模型直接学习并利用历史数据中物品或者特征的“共现频率”的能力；Deep 部分的主要作用是让模型具有“泛化能力”，“泛化能力”可以被理解为模型传递特征的相关性，以及发掘稀疏甚至从未出现过的稀有特征与最终标签相关性的能力；正是这样的结构特点，使模型兼具了逻辑回归和深度神经网络的优点-----能够快速处理并记忆大量历史行为特征，并且具有强大的表达能力。

7.4 点击率预估任务中负样本过多怎么办

正负样本不均衡问题一直伴随着算法模型存在，样本不均衡会导致：对比例大的样本造成过拟合，也就是说预测偏向样本数较多的分类。这样就会大大降低模型的泛化能力。往往 accuracy（准确率）很高，但 auc 很低。

正负样本不均衡问题的解决办法有三类：

- 采样处理——过采样，欠采样
- 类别权重——通过正负样本的惩罚权重解决样本不均衡的问题。在算法实现过程中，对于分类中不同样本数量的类别分别赋予不同的权重
- 集成方法——使用所有分类中的小样本量，同时从分类中的大样本量中随机抽取数据来与小样本量合并构成训练集，这样反复多次会得到很多训练集，从而训练出多个模型。例如，在数据集中的正、负样本分别为 100 和 10000 条，比例为 1：100，此时可以将负样本随机切分为 100 份，每份 100 条数据，然后每次形成训练集时使用所有的正样本（100 条）和随机抽取的负样本（100 条）形成新的训练数据集。如此反复可以得到 100 个模型。然后继续集成表决

一般情况下在选择正负样本时会进行相关比例的控制，假设正样本的条数是 N，则负样本的条数会控制在 2N 或者 3N，即遵循 1:2 或者 1:3 的关系，当然具体的业务场景下要进行不同的尝试和离线评估指标的对比。

第八篇 2021 年 7 月初，TPlink 算法工程师面试题 6 道！

8.1 介绍下 K 近邻、kmeans 聚类算法

K 近邻算法也称为 knn 算法。

knn 算法的核心思想是未标记样本的类别，由距离其最近的 k 个邻居投票来决定。

具体的，假设我们有一个已标记好的数据集。此时有一个未标记的数据样本，我们的任务是预测出这个数据样本所属的类别。knn 的原理是，计算待标记样本和数据集中每个样本的距离，取距离最近的 k 个样本。待标记的样本所属类别就由这 k 个距离最近的样本投票产生。

假设 X_test 为待标记的样本，X_train 为已标记的数据集，算法原理如下：

- 遍历 X_train 中的所有样本，计算每个样本与 X_test 的距离，并把距离保存在 Distance 数组中。
- 对 Distance 数组进行排序，取距离最近的 k 个点，记为 X_knn。
- 在 X_knn 中统计每个类别的个数，即 class0 在 X_knn 中有几个样本，class1 在 X_knn 中有几个样本等。
- 待标记样本的类别，就是在 X_knn 中样本个数最多的那个类别。

算法优缺点

优点：准确性高，对异常值和噪声有较高的容忍度。

缺点：计算量较大，对内存的需求也较大。

算法参数

其算法参数是 k，参数选择需要根据数据来决定。

k 值越大，模型的偏差越大，对噪声数据越不敏感，当 k 值很大时，可能造成欠拟合；

k 值越小，模型的方差就会越大，当 k 值太小，就会造成过拟合。

kmeans 聚类算法

K-means 算法的基本思想是：以空间中 k 个点为中心进行聚类，对最靠近他们的对象归类。通过迭代的方法，逐次更新各聚类中心的值，直至得到最好的聚类结果。

假设要把样本集分为 k 个类别，算法描述如下：

（1）适当选择 k 个类的初始中心，最初一般为随机选取；

（2）在每次迭代中，对任意一个样本，分别求其到 k 个中心的欧式距离，将该样本归到距离最短的中心所在的类；

（3）利用均值方法更新该 k 个类的中心的值；

（4）对于所有的 k 个聚类中心，重复（2）（3），类的中心值的移动距离满足一定条件时，则迭代结束，完成分类。

Kmeans 聚类算法原理简单，效果也依赖于 k 值和类中初始点的选择。

8.2 介绍下随机森林和 SVM 算法

随机森林是一种基于 bagging 的分类算法，它通过自助法（bootstrap）重采样技术，从原始训练样本集 N 中有放回地重复随机抽取 n 个样本生成新的训练样本集合训练决策树，然后按以上步骤生成 m 棵决策树组成随机森林，新数据的分类结果按分类树投票多少形成的分数而定。

随机森林大致过程如下：

1）从样本集中有放回随机采样选出 n 个样本；

2）从所有特征中随机选择 k 个特征，对选出的样本利用这些特征建立决策树（一般是 CART，也可是别的或混合）；

3）重复以上两步 m 次，即生成 m 棵决策树，形成随机森林；

4）对于新数据，经过每棵树决策，最后投票确认分到哪一类。

2.随机森林特点：

随机森林有很多优点：

1） 每棵树都选择部分样本及部分特征，一定程度避免过拟合；

2） 每棵树随机选择样本并随机选择特征，使得具有很好的抗噪能力，性能稳定；

3） 能处理很高维度的数据，并且不用做特征选择；

4） 适合并行计算；

5） 实现比较简单；

缺点：

1） 参数较复杂；

2） 模型训练和预测都比较慢。

SVM 算法：

是一种二分类模型，它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机。

SVM 可分为三种：

- 线性可分 SVM

当训练数据线性可分时，通过最大化硬间隔（hard margin）可以学习得到一个线性分类器，即硬间隔 SVM。

- 线性 SVM

当训练数据不能线性可分但是近似线性可分时，通过最大化软间隔（soft margin）也可以学习到一个线性分类器，即软间隔 SVM。

- 非线性 SVM

当训练数据线性不可分时，通过使用核技巧（kernel trick）和最大化软间隔，可以学习到一个非线性 SVM。

SVM 的的学习策略就是间隔最大化，可形式化为一个求解凸二次规划的问题，也等价于正则化的合页损失函数的最小化问题。SVM 的的学习算法就是求解凸二次规划的最优化算法。

8.3 Batch-norm 作用和参数

batch norm 的作用

1.batch norm 对于输入数据做了零均值化和方差归一化过程，方便了下一层网络的训练过程，从而加速了网络的学习。不同 batch 的数据，由于加入了 batch norm，中间层的表现会更加稳定，输出值不会偏移太多。各层之间受之前层的影响降低，各层之间比较独立，有助于加速网络的学习。梯度爆炸和梯度消失现象也得到了一些缓解（我自己加上去的）。

2. batch norm 利用的是 mini-batch 上的均值和方差来做的缩放，但是不同的 mini-batch 上面的数据是有波动的，相当于给整个模型引入了一些噪音，从而相当于有了一些正则化的效果，从而提升表现。

测试时的 batch norm 在训练过程中，y，b 参数和 w 相似，直接利用梯度值乘以学习率，更新值就好了。需要注意的是，batch norm 中的 z 的均值和方差都是通过每一个 mini-batch 上的训练数据得到的。在测试过程中，不能通过单独样本的数据计算均值和方差，我们可以通过让训练过程中的每一个 mini-batch 的均值和方差数据，计算指数加权平均，从而得到完整样本的均值和方差的一个估计。在测试过程中，使用该值作为均值和方差，从而完成计算。

8.4 L1/L2 的区别和作用

L1/L2 的区别

- L1 是模型各个参数的绝对值之和。

  L2 是模型各个参数的平方和的开方值。

- L1 会趋向于产生少量的特征，而其他的特征都是 0。

  因为最优的参数值很大概率出现在坐标轴上，这样就会导致某一维的权重为 0 ，产生稀疏权重矩阵

  L2 会选择更多的特征，这些特征都会接近于 0。

  最优的参数值很小概率出现在坐标轴上，因此每一维的参数都不会是 0。当最小化||w||时，就会使每一项趋近于 0。

L1 的作用是为了矩阵稀疏化。假设的是模型的参数取值满足拉普拉斯分布。

L2 的作用是为了使模型更平滑，得到更好的泛化能力。假设的是参数是满足高斯分布。

8.5 模型的加速与压缩

深度学习模型压缩与加速是指利用神经网络参数和结构的冗余性精简模型，在不影响任务完成度的情况下，得到参数量更少、结构更精简的模型。被压缩后的模型对计算资源和内存的需求更小，相比原始模

型能满足更广泛的应用需求。（事实上，压缩和加速是有区别的，压缩侧重于减少网络参数量，加速侧重于降低计算复杂度、提升并行能力等，压缩未必一定能加速）

主流的压缩与加速技术有 4 种：结构优化、剪枝（Pruning）、量化（Quantization）、知识蒸馏（Knowledge Distillation）。

8.6 两个链表存在交叉结点，怎么判断交叉点

该题为 leetcode160——相交链表

方法一：暴力解法

对于 A 中的每一个结点，我们都遍历一次链表 B 查找是否存在重复结点，第一个查找到的即第一个公共结点。

代码如下：

```
1.  class Solution:
2.      def getIntersectionNode(self, headA: ListNode, headB: ListNode) -
    > ListNode:
3.          p = headA
4.          while p:
5.              q = headB
6.              while q:
7.                  if p == q:
8.                      return p
9.                  q = q.next
10.             p = p.next
11.         return p
```

时间复杂度：O(n^2)

空间复杂度：O(1)

无法通过，会超时。

方法二：

对暴力解法的一个优化方案是：先将其中一个链表存到哈希表中，此时再遍历另外一个链表查找重复结点只需 O(n) 时间。

代码如下：

```
1.  class Solution:
2.      def getIntersectionNode(self, headA: ListNode, headB: ListNode) -
    > ListNode:
3.          s = set()
4.          p,q = headA,headB
5.          while p:
6.              s.add(p)
```

```
            p = p.next
        while q:
            if q in s:
                return q
            q = q.next
        return None
```

时间复杂度：O(n)

空间复杂度：O(n)

方法三：走过彼此的路

利用两链表长度和相等的性质来使得两个遍历指针同步。

具体做法是：让两指针同时开始遍历，遍历到结尾的时候，跳到对方的头指针，如果有公共结点，则，会同时到达相遇的地方。

代码如下：

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -
> ListNode:
        p,q = headA,headB
        while p != q:
            p = p.next if p else headB
            q = q.next if q else headA
        return p
```

时间复杂度：O(n)

空间复杂度：O(1)

第九篇 2021 年 7 月 20 日 - 7 月 26 日，字节推荐算法面试题 10 道!

9.1 bert 蒸馏了解吗

知识蒸馏的本质是让超大线下 teacher model 来协助线上 student model 的 training。

bert 的知识蒸馏，大致分成两种。

第一种，从 transformer 到非 transformer 框架的知识蒸馏

这种由于中间层参数的不可比性，导致从 teacher model 可学习的知识比较受限。但比较自由，可以把知识蒸馏到一个非常小的 model，但效果肯定会差一些。

第二种，从 transformer 到 transformer 框架的知识蒸馏

由于中间层参数可利用，所以知识蒸馏的效果会好很多，甚至能够接近原始 bert 的效果。但 transformer 即使只有三层，参数量其实也不少，另外蒸馏过程的计算也无法忽视。

所以最后用那种，还是要根据线上需求来取舍。

9.2 给你一些很稀疏的特征，用 LR 还是树模型

参考：很稀疏的特征表明是高维稀疏，用树模型（GBDT）容易过拟合。建议使用加正则化的 LR。

- 假设有 1w 个样本， y 类别 0 和 1，100 维特征，其中 10 个样本都是类别 1，而特征 f1 的值为 0，1，且刚好这 10 个样本的 f1 特征值都为 1，其余 9990 样本都为 0(在高维稀疏的情况下这种情况很常见)，我们都知道这种情况在树模型的时候，很容易优化出含一个使用 f1 为分裂节点的树直接将数据划分的很好，但是当测试的时候，却会发现效果很差，因为这个特征只是刚好偶然间跟 y 拟合到了这个规律，这也是我们常说的过拟合。但是当时我还是不太懂为什么线性模型就能对这种 case 处理的好？照理说 线性模型在优化之后不也会产生这样一个式子：y = W1*f1 + Wi*fi+....，其中 W1 特别大以拟合这十个样本吗，因为反正 f1 的值只有 0 和 1，W1 过大对其他 9990 样本不会有任何影响。
- 现在的模型普遍都会带着正则项，而 lr 等线性模型的正则项是对权重的惩罚，也就是 W1 一旦过大，惩罚就会很大，进一步压缩 W1 的值，使他不至于过大，而树模型则不一样，树模型的惩罚项通常为叶子节点数和深度等，而我们都知道，对于上面这种 case，树只需要一个节点就可以完美分割 9990 和 10 个样本，惩罚项极其之小.
- 这也就是为什么在高维稀疏特征的时候，线性模型会比非线性模型好的原因了：带正则化的线性模型比较不容易对稀疏特征过拟合。

9.3 LR 的损失函数推导

逻辑回归损失函数及梯度推导公式如下：

$$
\begin{aligned}
L(w) &= \sum_i \left(y_i * \operatorname{logh}\left(x_i\right) + \left(1-y_i\right) * \log\left(1-h\left(x_i\right)\right)\right) \\
&= \sum_i y_i \left(\operatorname{logh}\left(x_i\right) - \log\left(1-h\left(x_i\right)\right)\right) + \log\left(1-h\left(x_i\right)\right) \\
&= \sum_i y_i \log \frac{h\left(x_i\right)}{1-h\left(x_i\right)} + \log\left(1-h\left(x_i\right)\right) \\
&= \sum_i y_i \left(w^T x_i\right) + \log\left(1 - \frac{1}{1+e^{-wx_i}}\right) \\
&= \sum_i \left(y_i * \left(w^T x_i\right) - \log\left(1+e^{w^T x_i}\right)\right)
\end{aligned}
$$

求导：

$$
\begin{aligned}
\frac{dL}{dw} &= yx - \frac{1}{1+e^{w^T x}} * e^{w^T x} * x \\
&= x(y - h(x))
\end{aligned}
$$

9.4 为什么分类用交叉熵不用 MSE（从梯度的角度想一下）

LR 的基本表达形式如下：

$$
h_\theta(x) = g\left(\theta^T x\right) = \frac{1}{1+e^{-\theta^T x}}
$$

使用交叉熵作为损失函数的梯度下降更新求导的结果如下：
首先得到损失函数如下：

$$
C = \frac{1}{n} \sum [y \operatorname{In} \hat{y} + (1-y) \operatorname{In}(1-\hat{y})]
$$

如果我们使用 MSE 作为损失函数的话，那损失函数以及求导的结果如下所示：

$$
C = \frac{(y-\hat{y})^2}{2}
$$

$$
\frac{\partial C}{\partial w} = (\hat{y} - y)\sigma'(z)(x)
$$

使用平方损失函数，会发现梯度更新的速度和 sigmod 函数本身的梯度是很相关的。sigmod 函数在它在定义域内的梯度都不大于 0.25。这样训练会非常的慢。使用交叉熵的话就不会出现这样的情况，它的导数就是一个差值，误差大的话更新的就快，误差小的话就更新的慢点，这正是我们想要的。

在使用 Sigmoid 函数作为正样本的概率时，同时将平方损失作为损失函数，这时所构造出来的损失函数是非凸的，不容易求解，容易得到其局部最优解。如果使用极大似然，其目标函数就是对数似然函数，该损失函数是关于未知参数的高阶连续可导的凸函数，便于求其全局最优解。（关于是否是凸函数，由凸函数的定义得，对于一元函数，其二阶导数总是非负，对于多元函数，其 Hessian 矩阵（Hessian 矩阵是由多元函数的二阶导数组成的方阵）的正定性来判断。如果 Hessian 矩阵是半正定矩阵）

9.5 BERT 和 Roberta 的区别

RoBERTa 模型在 Bert 模型基础上的调整：

- 训练时间更长，Batch_size 更大，（Bert 256，RoBERTa 8K）
- 训练数据更多（Bert 16G，RoBERTa 160G）
- 移除了 NPL（next predict loss）
- 动态调整 Masking 机制
- Token Encoding：使用基于 bytes-level 的 BPE

简单总结如下：

参数\模型	Bert	RoBerta
Batch_size	256	2K、8K
训练数据	13G（Bert Large）	16G、160G
训练步数	1M	125K、31K
是否有 NSP	是	否
Mask 方式	static mask（静态 mask）	dynamic（动态 mask）
text Encoding	基于 char-level 的 BPE	基于 bytes-level 的 BPE

9.6 分词方法 BPE 和 WordPiece 的区别

BPE 与 Wordpiece 都是首先初始化一个小词表，再根据一定准则将不同的子词合并。词表由小变大。BPE 与 Wordpiece 的最大区别在于，如何选择两个子词进行合并：BPE 选择频数最高的相邻子词合并，而 WordPiece 选择能够提升语言模型概率最大的相邻子词加入词表。

9.7 残差网络有哪些作用？除了解决梯度消失？

解决梯度消失和网络退化问题。

残差网络为什么能解决梯度消失?

残差模块能让训练变得更加简单，如果输入值和输出值的差值过小，那么可能梯度会过小，导致出现梯度小时的情况，残差网络的好处在于当残差为 0 时，改成神经元只是对前层进行一次线性堆叠，使得网络梯度不容易消失，性能不会下降。

什么是网络退化?

随着网络层数的增加，网络会发生退化现象：随着网络层数的增加训练集 loss 逐渐下降，然后趋于饱和，如果再增加网络深度的话，训练集 loss 反而会增大，注意这并不是过拟合，因为在过拟合中训练 loss 是一直减小的。

残差网络为什么能解决网络退化?

在前向传播时，输入信号可以从任意低层直接传播到高层。由于包含了一个天然的恒等映射，一定程度上可以解决网络退化问题。

9.8 交叉熵损失，二分类交叉熵损失和极大似然什么关系?

什么是交叉熵损失?

机器学习的交叉熵损失函数定义为：假设有 N 个样本，

$$J(w)=\frac{1}{N}\sum_{n=1}^{N}H(p_n,q_n)$$

其中：

$$H(p,q)=-\sum_{i=1}^{K}p(x_i)\log q(x_i)$$

从一个直观的例子感受最小化交叉熵损失与极大似然的关系。

$$J(w)=-\frac{1}{N}\sum_{n=1}^{N}[y_n\log\hat{y_n}+(1-y_n)\log(1-\hat{y_n})]$$

去掉 1/N 并不影响函数的单调性，机器学习任务的也可以是最小化下面的交叉熵损失：

$$J(w)=-\sum_{n=1}^{N}[y_n\log\hat{y_n}+(1-y_n)\log(1-\hat{y_n})]$$

等价于最大化下面这个函数：

$$J(w) = \sum_{n=1}^{N} [y_n \log \hat{y_n} + (1 - y_n) \log(1 - \hat{y_n})]$$

不难看出，这其实就是对伯努利分布求极大似然中的对数似然函数(log-likelihood)。

也就是说，在伯努利分布下，极大似然估计与最小化交叉熵损失其实是同一回事。

可参考：https://zhuanlan.zhihu.com/p/51099880

9.9 二叉树最大路径和（路径上的所有节点的价值和最大）

参考答案

方法一：递归

首先，考虑实现一个简化的函数 maxGain(node)，该函数计算二叉树中的一个节点的最大贡献值，具体而言，就是在以该节点为根节点的子树中寻找以该节点为起点的一条路径，使得该路径上的节点值之和最大。

- 具体而言，该函数的计算如下。
- 空节点的最大贡献值等于 00。

非空节点的最大贡献值等于节点值与其子节点中的最大贡献值之和（对于叶节点而言，最大贡献值等于节点值）。

例如，考虑如下二叉树。

```
  -10
  / \
 9  20
   / \
  15   7
```

叶节点 99、1515、77 的最大贡献值分别为 99、1515、77。

得到叶节点的最大贡献值之后，再计算非叶节点的最大贡献值。节点 2020 的最大贡献值等于 20+max(15,7)=35，节点 -10 的最大贡献值等于 -10+max(9,35)=25。

上述计算过程是递归的过程，因此，对根节点调用函数 maxGain，即可得到每个节点的最大贡献值。

根据函数 maxGain 得到每个节点的最大贡献值之后，如何得到二叉树的最大路径和？对于二叉树中的一个节点，该节点的最大路径和取决于该节点的值与该节点的左右子节点的最大贡献值，如果子节点的最大贡献值为正，则计入该节点的最大路径和，否则不计入该节点的最大路径和。维护一个全局变量 maxSum 存储最大路径和，在递归过程中更新 maxSum 的值，最后得到的 maxSum 的值即为二叉树中的最大路径和。

```
1.  class Solution:
2.      def __init__(self):
3.          self.maxSum = float("-inf")
```

```
4.
5.      def maxPathSum(self, root: TreeNode) -> int:
6.          def maxGain(node):
7.              if not node:
8.                  return 0
9.
10.             # 递归计算左右子节点的最大贡献值
11.             # 只有在最大贡献值大于 0 时，才会选取对应子节点
12.             leftGain = max(maxGain(node.left), 0)
13.             rightGain = max(maxGain(node.right), 0)
14.
15.             # 节点的最大路径和取决于该节点的值与该节点的左右子节点的最大贡献值
16.             priceNewpath = node.val + leftGain + rightGain
17.
18.             # 更新答案
19.             self.maxSum = max(self.maxSum, priceNewpath)
20.
21.             # 返回节点的最大贡献值
22.             return node.val + max(leftGain, rightGain)
23.
24.         maxGain(root)
25.         return self.maxSum
```

9.10 写题：找数组中第 k 大的数，复杂度

三种思路：一种是直接使用 sorted 函数进行排序，一种是使用小顶堆，一种是使用快排（双指针 + 分治）。

方法一：直接使用 sorted 函数进行排序

代码如下：

```
1.  class Solution:
2.      def findKthLargest(self, nums: List[int], k: int) -> int:
3.          return sorted(nums, reverse = True)[k-1]
```

方法二：

维护一个 size 为 k 的小顶堆，把每个数丢进去，如果堆的 size > k，就把堆顶 pop 掉（因为它是最小的），这样可以保证堆顶元素一定是第 k 大的数。

代码如下：

```
1.  class Solution:
2.      def findKthLargest(self, nums: List[int], k: int) -> int:
```

```
        heap = []
        for num in nums:
            heappush(heap,num)
            if len(heap) > k:
                heappop(heap)
        return heap[0]
```

时间复杂度：O(nlogk)

空间复杂度：O(k)

方法三：双指针 + 分治

partition 部分

定义两个指针 left 和 right，还要指定一个中心 pivot（这里直接取最左边的元素为中心，即 nums[i]）不断将两个指针向中间移动，使得大于 pivot 的元素都在 pivot 的右边，小于 pivot 的元素都在 pivot 的左边，注意最后满足时，left 是和 right 相等的，因此需要将 pivot 赋给此时的 left 或 right。

然后再将中心点的索引和 k - 1 进行比较，通过不断更新 left 和 right 找到最终的第 k 个位置。

代码如下：

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        left, right, target = 0, len(nums)-1, k-1
        while True:
            pos = self.partition(nums, left, right)
            if pos == target:
                return nums[pos]
            elif pos > target:
                right = pos -1
            else:
                left = pos + 1

    def partition(self, nums, left, right):
        pivot = nums[left]
        while left < right:
            while nums[right] <= pivot and left < right:
                right -=1
            nums[left] = nums[right]
            while nums[left] >= pivot and left < right:
                left +=1
            nums[right] = nums[left]
        nums[left] = pivot
        return left
```

时间复杂度：O(n)，原版快排是 O(nlogn)，而这里只需要在一个分支递归，因此降为 O(n)

空间复杂度：O(logn)

第十篇 2021 年 7 月 16 日，腾讯 PGB，NLP 算法面试题 6 道!

10.1 SVM 的 优化函数公式怎么写，代价函数是什么?

线性可分支持向量机的最优化问题函数公式：

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 \\ \text{s.t.} \quad & y_i(w \cdot x_i + b) - 1 \geqslant 0, \quad i = 1, 2, \cdots, N \end{aligned}$$

引入拉格朗日乘子，由拉格朗日对偶性可得代价函数如下：

$$\begin{aligned} \max_{\alpha} \quad & -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j (x_i \cdot x_j) + \sum_{i=1}^{N}\alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^{N}\alpha_i y_i = 0 \\ & \alpha_i \geqslant 0, \quad i = 1, 2, \cdots, N \end{aligned}$$

10.2 随机森林算法原理及优缺点?

随机森林是一种重要的基于 Bagging，进一步在决策树的训练过程中引入随机属性选择的集成学习方法，可以用来做分类、回归等问题。

随机森林的构建过程大致如下：

- 从原始训练集中使用 Bootstraping 方法随机有放回采样选出 m 个样本，共进行 n_tree 次采样，生成 n_tree 个训练集；
- 对于 n_tree 个训练集，我们分别训练 n_tree 个决策树模型；
- 对于单个决策树模型，假设训练样本特征的个数为 n，那么每次分裂时根据信息增益/信息增益比/基尼指数选择最好的特征进行分裂；
- 每棵树都一直这样分裂下去，直到该节点的所有训练样例都属于同一类。在决策树的分裂过程中不需要剪枝；
- 将生成的多棵决策树组成随机森林。对于分类问题，按多棵树分类器投票决定最终分类结果；对于回归问题，由多棵树预测值的均值决定最终预测结果。
-

随机森林的优点：

具有极高的准确率；

随机性的引入，使得随机森林不容易过拟合；

随机性的引入，使得随机森林有很好的抗噪声能力；

能处理很高维度的数据，并且不用做特征选择；

既能处理离散型数据，也能处理连续型数据，数据集无需规范化；

训练速度快，可以得到变量重要性排序；

容易实现并行化。

随机森林的缺点：

当随机森林中的决策树个数很多时，训练时需要的空间和时间会较大；

随机森林模型还有许多不好解释的地方，有点算个黑盒模型。

10.3 XGBOOST 相对于 GDBT 最大的优势是什么，XGBOOST 求二阶导数的好处都有什么?

优势：加入了正则化和二阶导数。

XGBOOST 求二阶导数的好处：

xgboost 是在 MSE 基础上推导出来的，在 MSE 的情况下，xgboost 的目标函数展开就是一阶项+二阶项的形式，而其他类似 log loss 这样的目标函数不能表示成这种形式。为了后续推导的统一，所以将目标函数进行二阶泰勒展开，就可以直接自定义损失函数了，只要损失函数是二阶可导的即可，增强了模型的扩展性。

二阶信息能够让梯度收敛的更快。一阶信息描述梯度变化方向，二阶信息可以描述梯度变化方向是如何变化的。

10.4 BERT 是怎么分词的?

BERT 主要是基于 Wordpiece 进行分词的，其思想是选择能够提升语言模型概率最大的相邻子词加入词表。

BERT 源码中 tokenization.py 就是预处理进行分词的程序，主要有两个分词器：

BasicTokenizer 和 WordpieceTokenizer，另外一个 FullTokenizer 是这两个的结合：先进行 BasicTokenizer 得到一个分得比较粗的 token 列表，然后再对每个 token 进行一次 WordpieceTokenizer，得到最终的分词结果。

其中 BasicTokenizer 完成： 转 unicode)-> 去除空字符、替换字符、控制字符和空白字符等奇怪字符 -> 中文分词 -> 空格分词 -> 小写、去掉变音符号、标点分词；WordpieceTokenizer 大致分词思路是按照从左到右的顺序，将一个词拆分成多个子词，每个子词尽可能长。按照源码中的说法，该方法称之为 greedy longest-match-first algorithm，贪婪最长优先匹配算法。

对于中文来说，一句话概括：BERT 采取的是「分字」，即每一个汉字都切开。

10.5 Word2Vec 的训练方式有哪两种？（指的是分层 softmax 和负采样），CBOW 和 SkipGram 各自的优缺点都是什么？（一个计算量小，一个慢但是细致）

Word2Vec 的训练方式有分层 softmax 和负采样。

Skip-Gram 和 CBOW 相比，多了一层循环（2c 次），所以的确是计算量大约为 CBOW 的 2c 倍。

鉴于 skip-gram 学习的词向量更细致，但语料库中有大量低频词时，使用 skip-gram 学习比较合适。

Skip-Gram 模型相当于是“家教学习模式”，即一个学生请了多个家教对其进行一对一辅导。表现为每个输入的中心词（学生）从多个上下文词标签（家教老师）那里获得知识。

CBOW 模型则相当于“大班教学模式”，即多个学生在一个大班老师那里上课学习，表现为每个输入的多个上下文词（多个学生）从中心词标签（大班老师）那里获得知识。

10.6 评估模型的指标都有什么，AUC 和 ROC 具体是什么，代表了什么属性?

评估模型的指标：准确率（accuracy），召回率（Recall），AUC 等。

ROC 曲线是基于样本的真实类别和预测概率来画的，具体来说，ROC 曲线的 x 轴是伪阳性率（false positive rate），y 轴是真阳性率（true positive rate)。

其中：真阳性率=（真阳性的数量）/（真阳性的数量+伪阴性的数量）

伪阳性率=（伪阳性的数量）/（伪阳性的数量+真阴性的数量）

AUC 是 ROC 曲线下面的面积，AUC 可以解读为从所有正例中随机选取一个样本 A，再从所有负例中随机选取一个样本 B，分类器将 A 判为正例的概率比将 B 判为正例的概率大的可能性。AUC 反映的是分类器对样本的排序能力。AUC 越大，自然排序能力越好，即分类器将越多的正例排在负例之前。

更多请看七月在线题库里的这题：https://www.julyedu.com/question/big/kp_id/23/ques_id/1052

第十一篇 2021 年 7 月 17 日，蔚来 NLP 算法工程师面试题 4 道!

11.1 你常用的优化算法？有什么特点？为什么?

梯度下降：在一个方向上更新和调整模型的参数，来最小化损失函数。

随机梯度下降（Stochastic gradient descent，SGD）对每个训练样本进行参数更新，每次执行都进行一次更新，且执行速度更快。

为了避免 SGD 和标准梯度下降中存在的问题，一个改进方法为小批量梯度下降（Mini Batch Gradient Descent），因为对每个批次中的 n 个训练样本，这种方法只执行一次更新。

使用小批量梯度下降的优点是：
1) 可以减少参数更新的波动，最终得到效果更好和更稳定的收敛。
2) 还可以使用最新的深层学习库中通用的矩阵优化方法，使计算小批量数据的梯度更加高效。
3) 通常来说，小批量样本的大小范围是从 50 到 256，可以根据实际问题而有所不同。
4) 在训练神经网络时，通常都会选择小批量梯度下降算法。

SGD 方法中的高方差振荡使得网络很难稳定收敛，所以有研究者提出了一种称为动量（Momentum）的技术，通过优化相关方向的训练和弱化无关方向的振荡，来加速 SGD 训练。

Nesterov 梯度加速法，通过使网络更新与误差函数的斜率相适应，并依次加速 SGD，也可根据每个参数的重要性来调整和更新对应参数，以执行更大或更小的更新幅度。

AdaDelta 方法是 AdaGrad 的延伸方法，它倾向于解决其学习率衰减的问题。Adadelta 不是累积所有之前的平方梯度，而是将累积之前梯度的窗口限制到某个固定大小 w。

Adam 算法即自适应时刻估计方法（Adaptive Moment Estimation），能计算每个参数的自适应学习率。这个方法不仅存储了 AdaDelta 先前平方梯度的指数衰减平均值，而且保持了先前梯度 M(t)的指数衰减平均值，这一点与动量类似。

Adagrad 方法是通过参数来调整合适的学习率η，对稀疏参数进行大幅更新和对频繁参数进行小幅更新。因此，Adagrad 方法非常适合处理稀疏数据。

11.2 Kmeans 和 EM 算法？Kmeans 和 EM 算法很相似，类比一下?

K-means（K 均值聚类）是一种基于中心的聚类算法，通过迭代，将样本分到 K 个类中，使每个样本与其所属类中心的距离之和最小。

EM 的算法核心就是 E 步和 M 步（期望步和最大化步）

期望步（E-步）

给定当前的簇中心，每个对象都被指派到簇中心离该对象最近的簇。这里，期望每个对象都属于最近的簇。

最大化步（M-步）

给定簇指派，对于每·个簇，算法调整其中心，使得指派到该簇的对象到该新中心到的距离之和最小化。也就是说，将指派到一个簇的对象的相似度最大化。

11.3 圆上任选一条弦，其长度大于圆内接正三角形边长的概率为?

解法一：由于对称性，可预先指定弦的方向。作垂直于此方向的直径，只有交直径于 1/4 点与 3/4 点间的弦，其长才大于内接正三角形边长。所有交点是等可能的,则所求概率为 1/2 。此时假定弦的中心在直径上均匀分布。

解法二：由于对称性，可预先固定弦的一端。仅当弦与过此端点的切线的交角在 60° ~ 120° 之间，其长才合乎要求。所有方向是等可能的，则所求概率为 1/3 。此时假定端点在圆周上均匀分布。

解法三： 弦被其中点位置唯一确定。只有当弦的中点落在半径缩小了一半的同心圆内，其长才合乎要求。中点位置都是等可能的,则所求概率为 1/4。此时假定弦长被其中心唯一确定。

可见，上述三个答案是针对三个不同样本空间引起的，它们都是正确的，贝特朗悖论引起人们注意，在定义概率时要事先明确指出样本空间是什么。

贝特朗悖论在普通高中中模拟概率时会出现。一般第一种答案（即”1/3 “）使用较为广泛。

可参考：https://kechuang.org/t/51595

https://zhuanlan.zhihu.com/p/158679111

11.4 判断一个链表是否是回文串

思路：将值添加到数组中，翻转后进行比较。

将链表值添加到数组列表中，然后直接比较翻转前后是否相等。

```
1.  class Solution:
2.      def isPalindrome(self, head: ListNode) -> bool:
3.          vals = []
```

```
        current_node = head
        while current_node is not None:
            vals.append(current_node.val)
            current_node = current_node.next
        return vals == vals[::-1]
```

时间复杂度：O(n)

空间复杂度：O(n)

第十二篇 2021 年 7 月 18 日，虾皮北京提前批法算法工程师面试题 5 道！

12.1 删除链表倒数第 K 个节点

该题为 leetcode 第 19 题。

在对链表进行操作时，一个常用的技巧就是添加一个哑结点（dummy node）,它的 next 忠贞纸箱链表的头结点，这样就不需要对头结点进行特殊判断了。

思路一：计算链表长度

先对链表进行一次遍历，得到链表的长度 L，随后再从头节点开始对链表进行一次遍历，当遍历到第 L-n+1 个节点时，就是需要删除的节点。当我们在头结点前面加上 dummy 节点后，删除的节点就变为了 L-n+1 的下一个节点，通过修改指针来完成删除操作。

代码如下：

```
1.  class Solution:
2.      def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
3.          def getLength(head: ListNode) -> int:
4.              length = 0
5.              while head:
6.                  length += 1
7.                  head = head.next
8.              return length
9.
10.         dummy = ListNode(0, head)
11.         length = getLength(head)
12.         cur = dummy
13.         for i in range(1, length - n + 1):
14.             cur = cur.next
15.         cur.next = cur.next.next
16.         return dummy.next
```

时间复杂度：O(L)，其中 L 是链表的长度。

空间复杂度：O(1)。

方法二：栈

在对链表进行遍历时将所有的节点依次放入栈，根据栈先进后出的原则，我们弹出的第 n 个节点就是需要删除的节点，且此时栈顶的节点就是待删除节点的前驱节点。

代码如下：

```
1.  class Solution:
2.      def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
```

```
        dummy = ListNode(0, head)
        stack = list()
        cur = dummy
        while cur:
            stack.append(cur)
            cur = cur.next

        for i in range(n):
            stack.pop()

        prev = stack[-1]
        prev.next = prev.next.next
        return dummy.next
```

时间复杂度：O(L)，其中 L 是链表的长度。

空间复杂度：O(L)，其中 L 是链表的长度。主要为栈的开销。

方法三：双指针

该方法的优点在于不需要预处理链表的长度

使用两个指针 first 和 second 同时对链表进行遍历，并且 first 比 second 超前 n 个节点。当 first 遍历到链表的末尾时 second 就恰好处于倒数第 n 个节点。

代码如下：

```
class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0, head)
        first = head
        second = dummy
        for i in range(n):
            first = first.next

        while first:
            first = first.next
            second = second.next

        second.next = second.next.next
        return dummy.next
```

时间复杂度：O(L)，其中 LL 是链表的长度。

空间复杂度：O(1)。

12.2 将数组划分为给定和为 k 的 2 部分。

该题为 leetcode416 题，结题思路如下：

01 背包问题——能不能装满容量为 target 的背包

本题要求把数组分成两个等和的子集，相当于找到一个子集，其和为 sum/2，这个 sum/2 就是 target

具体步骤如下：

1、特例：如果 sum 为奇数，那一定找不到符合要求的子集，返回 False。

2、dp[j]含义：有没有和为 j 的子集，有为 True，没有为 False。

3、初始化 dp 数组：长度为 target + 1，用于存储子集的和从 0 到 target 是否可能取到的情况。

比如和为 0 一定可以取到（也就是子集为空），那么 dp[0] = True。

4、接下来开始遍历 nums 数组，对遍历到的数 nums[i]有两种操作，一个是选择这个数，一个是不选择这个数。

- 不选择这个数：dp 不变
- 选择这个数：dp 中已为 True 的情况再加上 nums[i]也为 True。比如 dp[0]已经为 True，那么 dp[0 + nums[i]]也是 True

5、在做出选择之前，我们先逆序遍历子集的和从 nums[i]到 target 的所有情况，判断当前数加入后，dp 数组中哪些和的情况可以从 False 变成 True。

（为什么要逆序，是因为 dp 后面的和的情况是从前面的情况转移过来的，如果前面的情况因为当前 nums[i]的加入变为了 True，比如 dp[0 + nums[i]]变成了 True，那么因为一个数只能用一次，dp[0 + nums[i] + nums[i]]不可以从 dp[0 + nums[i]]转移过来。如果非要正序遍历，必须要多一个数组用于存储之前的情况。而逆序遍历可以省掉这个数组）

dp[j] = dp[j] or dp[j - nums[i]]

这行代码的意思是说，如果不选择当前数，那么和为 j 的情况保持不变，dp[j]仍然是 dp[j]，原来是 True 就还是 True，原来是 False 也还是 False；

如果选择当前数，那么如果 j - nums[i]这种情况是 True 的话和为 j 的情况也会是 True。比如和为 0 一定为 True，只要 j - nums[i] == 0，那么 dp[j]就变成了 True。

dp[j]和 dp[j-nums[i]]只要有一个为 True，dp[j]就变成 True，因此用 or 连接两者。

最后就看 dp[-1]是不是 True，也就是 dp[target]是不是 True

代码如下：

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        sumAll = sum(nums)
        if sumAll % 2:
            return False
        target = sumAll // 2

        dp = [False] * (target + 1)
        dp[0] = True

        for i in range(len(nums)):
            for j in range(target, nums[i] - 1, -1):
                dp[j] = dp[j] or dp[j - nums[i]]
        return dp[-1]
```

时间复杂度：O(n * target)

空间复杂度：O(target)

参考：https://leetcode-cn.com/problems/partition-equal-subset-sum/solution/416-fen-ge-deng-he-zi-ji-dong-tai-gui-hu-csk5/

12.3 二叉树的后序遍历（非递归）

方法一：递归

树本身就有递归的特性，因此递归方法最简单。

代码如下：

```python
class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        return self.postorderTraversal(root.left) + self.postorderTraversal(root.right) + [root.val]
```

时间复杂度：O(n)，n 为树的节点个数

空间复杂度：O(h)，h 为树的高度

方法二：迭代

注意：该代码是基于前序遍历改来的，由于前序遍历是中左右，后序遍历是左右中，所以只需要写出中右左，再进行反转即可得到左右中。

代码如下：

```python
class Solution:
```

```
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        res = []
        stack = []
        cur = root
        while stack or cur:
            while cur:
                stack.append(cur)
                res.append(cur.val)
                cur = cur.right
            cur = stack.pop()
            cur = cur.left
        return res[::-1]
```

时间复杂度：O(n)，n 为树的节点个数

空间复杂度：O(h)，h 为树的高度

12.4 怎么求特征重要性（GBDT RF 等）

RF 有两种方法：

1.通过计算 Gini 系数的减少量 VIm=GI−(GIL+GIR) 判断特征重要性，越大越重要。

2.对于一颗树，先使用 OOB 样本计算测试误差 a，再随机打乱 OOB 样本中第 i 个特征（上下打乱特征矩阵第 i 列的顺序）后计算测试误差 b，a 与 b 差距越大特征 i 越重要。

GBDT 计算方法：所有回归树中通过特征 i 分裂后平方损失的减少值的和/回归树数量 得到特征重要性。

在 sklearn 中，GBDT 和 RF 的特征重要性计算方法是相同的，都是基于单棵树计算每个特征的重要性，探究每个特征在每棵树上做了多少的贡献，再取个平均值。

Xgb 主要有三种计算方法：

a. importance_type=weight（默认值），特征重要性使用特征在所有树中作为划分属性的次数。

b. importance_type=gain，特征重要性使用特征在作为划分属性时 loss 平均的降低量。

c. importance_type=cover，特征重要性使用特征在作为划分属性时对样本的覆盖度。

12.5 梯度爆炸和梯度消失原因，解决方案

梯度消失：（1）隐藏层的层数过多；（2）采用了不合适的激活函数(更容易产生梯度消失，但是也有可能产生梯度爆炸)

梯度爆炸：（1）隐藏层的层数过多；（2）权重的初始化值过大。

梯度消失和梯度爆炸问题都是因为网络太深，网络权值更新不稳定造成的，本质上是因为梯度反向传播中的连乘效应。对于更普遍的梯度消失问题，可以考虑一下三种方案解决：

（1）用 ReLU、Leaky-ReLU、P-ReLU、R-ReLU、Maxout 等替代 sigmoid 函数。(几种激活函数的比较见我的博客)

（2）用 Batch Normalization。(对于 Batch Normalization 的理解可以见我的博客)

（3）LSTM 的结构设计也可以改善 RNN 中的梯度消失问题。

第十三篇 2021 年 7 月 19 日，百度算法面试题 5 道！

13.1 过拟合 怎么解决

过拟合：是指训练误差和测试误差之间的差距太大。换句换说，就是模型复杂度高于实际问题，模型在训练集上表现很好，但在测试集上却表现很差。

欠拟合：模型不能在训练集上获得足够低的误差。换句换说，就是模型复杂度低，模型在训练集上就表现很差，没法学习到数据背后的规律。

如何解决欠拟合?

欠拟合基本上都会发生在训练刚开始的时候，经过不断训练之后欠拟合应该不怎么考虑了。但是如果真的还是存在的话，可以通过增加网络复杂度或者在模型中增加特征，这些都是很好解决欠拟合的方法。

如何防止过拟合?

获取和使用更多的数据（数据集增强）、降低模型复杂度、L1\L2\Dropout 正则化、Early stopping（提前终止）

13.2 朴素贝叶斯 为啥朴素

之所以被称为“朴素”， 是因为它假定所有的特征在数据集中的作用是同样重要和独立的，正如我们所知，这个假设在现实世界中是很不真实的，因此，说是很“朴素的”。

13.3 python 装饰器

装饰器本质上是一个 Python 函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。

如：使用 python 装饰器获取某个函数的运行时间

```
1.  import datetime
2.  def count_time(func):
3.      def int_time(*args, **kwargs):
4.          start_time = datetime.datetime.now()  # 程序开始时间
5.          func()
6.          over_time = datetime.datetime.now()   # 程序结束时间
7.          total_time = (over_time-start_time).total_seconds()
8.          print('程序共计%s 秒' % total_time)
9.      return int_time
10.
11. @count_time
```

```
def main():
    print('>>>>开始计算函数运行时间')
    for i in range(1, 1000):           # 可以是任意函数 ， 这里故意模拟函数的运行时间
        for j in range(i):
            print(j)

if __name__ == '__main__':
    main()
```

13.4 python 生成器

介绍 python 生成器需要先介绍可迭代对象和迭代器。

可迭代对象（Iterable Object），简单的来理解就是可以使用 for 来循环遍历的对象。比如常见的 list、set 和 dict。

可迭代对象具有__iter__ 方法，用于返回一个迭代器，或者定义了 getitem 方法，可以按 index 索引的对象（并且能够在没有值时抛出一个 IndexError 异常），因此，可迭代对象就是能够通过它得到一个迭代器的对象。所以，可迭代对象都可以通过调用内建的 iter() 方法返回一个迭代器。

生成器其实是一种特殊的迭代器，不过这种迭代器更加优雅。它不需要再像上面的类一样写__iter__()和__next__()方法了，只需要一个 yiled 关键字。

13.5 L1 正则化与 L2 正则化的区别

参考答案：

L1 范数（L1 norm）是指向量中各个元素绝对值之和，也有个美称叫“稀疏规则算子”（Lasso regularization）。

比如 向量 A=[1，-1，3]， 那么 A 的 L1 范数为 |1|+|-1|+|3|.

简单总结一下就是：

L1 范数: 为 x 向量各个元素绝对值之和。

L2 范数: 为 x 向量各个元素平方和的 1/2 次方，L2 范数又称 Euclidean 范数或者 Frobenius 范数

Lp 范数: 为 x 向量各个元素绝对值 p 次方和的 1/p 次方.

在支持向量机学习过程中，L1 范数实际是一种对于成本函数求解最优的过程，因此，L1 范数正则化通过向成本函数中添加 L1 范数，使得学习得到的结果满足稀疏化，从而方便人类提取特征，即 L1 范数可以使权值稀疏，方便特征提取。

L2 范数可以防止过拟合，提升模型的泛化能力。

L1 和 L2 的差别，为什么一个让绝对值最小，一个让平方最小，会有那么大的差别呢？看导数一个是 1 一个是 w 便知, 在靠进零附近, L1 以匀速下降到零, 而 L2 则完全停下来了. 这说明 L1 是将不重要的特征(或者说, 重要性不在一个数量级上)尽快剔除, L2 则是把特征贡献尽量压缩最小但不至于为零. 两者一起作用, 就是把重要性在一个数量级(重要性最高的)的那些特征一起平等共事(简言之, 不养闲人也不要超人)。

第十四篇 2021 年 9 月初，b 站-主站技术中心-算法开发面试题 5 道

14.1 介绍 word2vec，负采样的细节

word2vec 是 google 于 2013 年开源推出的一个词向量表示的工具包，其具体是通过学习文本来用词向量的方式表征词的语义信息，即通过一个低维嵌入空间使得语义上相似的单词在该空间内的距离很近。有两种模型：CBOW 和 Skip-Gram，其中 CBOW 模型的输入是某一个特征词的上下文固定窗口的词对应的词向量，而输出就是该特定词的词向量；Skip-Gram 模型的输入是特定的一个词的词向量，输出就是特定词对应的上下文固定窗口的词向量。

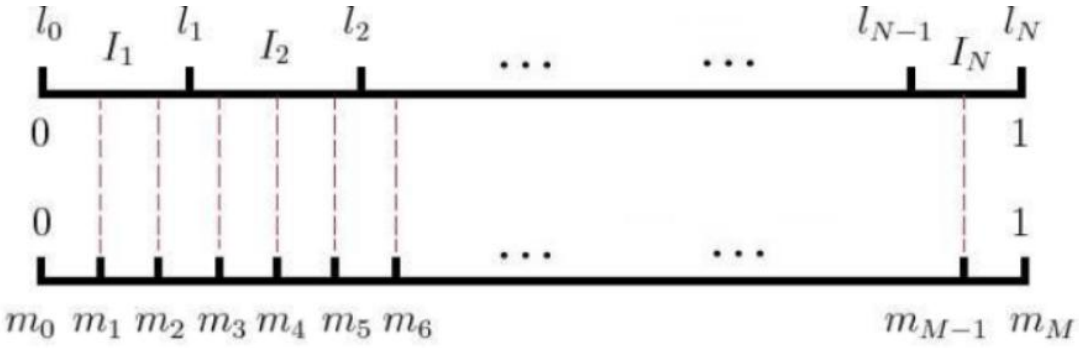
现在我们看下 Word2vec 如何通过 Negative Sampling 负采样方法得到 neg 个负例；如果词汇表的大小为 V,那么我们就将一段长度为 1 的线段分成 V 份，每份对应词汇表中的一个词。当然每个词对应的线段长度是不一样的，高频词对应的线段长，低频词对应的线段短。每个词 w 的线段长度由下式决定：

$$\mathrm{len}(w) = \frac{\mathrm{count}(w)}{\sum_{u \in vocab} \mathrm{count}(u)}$$

在 word2vec 中，分子和分母都取了 3/4 次幂如下：

$$\mathrm{len}(w) = \frac{\mathrm{count}(w)^{3/4}}{\sum_{u \in vocab} \mathrm{count}(u)^{3/4}}$$

在采样前，我们将这段长度为 1 的线段划分成 M 等份，这里 M>>V，这样可以保证每个词对应的线段都会划分成对应的小块。而 M 份中的每一份都会落在某一个词对应的线段上。在采样的时候，我们只需要从 M 个位置中采样出 neg 个位置就行，此时采样到的每一个位置对应到的线段所属的词就是我们的负例词。



在 word2vec 中，M 取值默认为 10^8。

14.2 fasttext 的改进，特征 hash 的作用

fastText 模型架构和 word2vec 中的 CBOW 很相似， 不同之处是 fastText 预测标签而 CBOW 预测的是中间词，即模型架构类似但是模型的任务不同。

对于 word2vec 来说，词向量的加和求平均会丢失词顺序的信息，为了弥补这一点 Fasttext 增加了 N-gram 特征，将 n-gram 当成一个词，也用 embedding 来表示，所以 look-up 矩阵为 [vocab+n_gram_num, dim]。由于 n-gram 的数量远大于 word 的数量，完全存储不现实，所以 fasttext 采用 hash 桶的方式，将所有 n-gram 都哈希到 buckets 桶中，哈希到同一个桶中的 n-gram 共享 embedding vector，潜在的问题是存在哈希冲突，当然如果桶 buckets 取足够大时，可以避免这个问题。

14.3 用 rand7 实现 rand10

该题为 leetocde470. 用 Rand7() 实现 Rand10()，下面给出解法。

首先需要找到两个规律，规律一：

已知 rand_N() 可以等概率的生成[1, N]范围的随机数

那么：

(rand_X() - 1) × Y + rand_Y() ==> 可以等概率的生成[1, X * Y]范围的随机数

即实现了 rand_XY()

规律二：

要实现 rand10()，就需要先实现 rand_N()，并且保证 N 大于 10 且是 10 的倍数。这样再通过 rand_N() % 10 + 1 就可以得到[1,10]范围的随机数了。

最后，由于生成的 49 不是 10 的倍数，故需要采用‘拒绝采样’，也就是说如果某个采样结果不在范围内就要丢弃它，如代码如下：

```
1.  class Solution:
2.      def rand10(self) -> int:
3.          while True:
4.              idx = (rand7() - 1) * 7 + rand7()
5.              if idx <= 40:
6.                  return 1 + idx % 10
```

时间复杂度：O(1)

空间复杂度：O(1)

参考：https://leetcode-cn.com/problems/implement-rand10-using-rand7/solution/cong-zui-ji-chu-de-jiang-qi-ru-he-zuo-dao-jun-yun-/

14.4 介绍一下 Bert 以及三个下游任务

Bert 模型是一种自编码语言模型，其主要结构是 transformer 的 encoder 层，其主要包含两个训练阶段，预训练与 fine-tuning，其中预训练阶段的任务是 Masked Language Model(完形填空) 和 Next Sentence Prediction。

下游任务：句子对分类任务，单句子分类任务，问答任务，单句子标注任务。

14.5 除了 Bert，其他预训练模型的拓展

RoBERTa 模型在 Bert 模型基础上的调整：

- 训练时间更长，Batch_size 更大，（Bert 256，RoBERTa 8K）
- 训练数据更多（Bert 16G，RoBERTa 160G）
- 移除了 NPL（next predict loss）
- 动态调整 Masking 机制
- Token Encoding：使用基于 bytes-level 的 BPE

第十五篇 2021 年 9 月初，字节秋招算法面试题 5 道

15.1 搜索旋转排序数组带重复值

该题为 leetcode 第 81 题，搜索先转排序数组 II

对于数组中有重复元素的情况，二分查找时可能会有 a[l]=a[mid]=a[r]，此时无法判断区间 [l,mid] 和区间 [mid+1,r] 哪个是有序的。

例如 nums=[3,1,2,3,3,3,3]，target=2，首次二分时无法判断区间 [0,3][0,3] 和区间 [4,6][4,6] 哪个是有序的。

对于这种情况，只能将当前二分区间的左边界加一，右边界减一，然后在新区间上继续二分查找。

代码如下：

```
1.  class Solution:
2.      def search(self, nums: List[int], target: int) -> bool:
3.          if not nums:
4.              return False
5.
6.          n = len(nums)
7.          if n == 1:
8.              return nums[0] == target
9.
10.         l, r = 0, n - 1
11.         while l <= r:
12.             mid = (l + r) // 2
13.             if nums[mid] == target:
14.                 return True
15.             if nums[l] == nums[mid] and nums[mid] == nums[r]:
16.                 l += 1
17.                 r -= 1
18.             elif nums[l] <= nums[mid]:
19.                 if nums[l] <= target and target < nums[mid]:
20.                     r = mid - 1
21.                 else:
22.                     l = mid + 1
23.             else:
24.                 if nums[mid] < target and target <= nums[n - 1]:
25.                     l = mid + 1
26.                 else:
27.                     r = mid - 1
28.
29.          return False
```

15.2 二叉树的之字形遍历

方法一：层序遍历 + 双端队列

- 利用双端队列的两端皆可添加元素的特性，设打印列表（双端队列） tmp ，并规定：

  奇数层 则添加至 tmp 尾部，

  偶数层 则添加至 tmp 头部 。

算法流程：

- 特例处理： 当树的根节点为空，则直接返回空列表 [] ；
- 初始化： 打印结果空列表 res ，包含根节点的双端队列 deque ；
- BFS 循环： 当 deque 为空时跳出；

  新建列表 tmp ，用于临时存储当前层打印结果；

  当前层打印循环： 循环次数为当前层节点数（即 deque 长度）；

  出队： 队首元素出队，记为 node；

  打印： 若为奇数层，将 node.val 添加至 tmp 尾部；否则，添加至 tmp 头部；

  添加子节点： 若 node 的左（右）子节点不为空，则加入 deque ；

  将当前层结果 tmp 转化为 list 并添加入 res ；
- 返回值： 返回打印结果列表 res 即可；

```
1.  class Solution:
2.      def levelOrder(self, root: TreeNode) -> List[List[int]]:
3.          if not root: return []
4.          res, deque = [], collections.deque([root])
5.          while deque:
6.              tmp = collections.deque()
7.              for _ in range(len(deque)):
8.                  node = deque.popleft()
9.                  if len(res) % 2: tmp.appendleft(node.val) # 偶数层 -> 队列头
    部
10.                 else: tmp.append(node.val) # 奇数层 -> 队列尾部
11.                 if node.left: deque.append(node.left)
12.                 if node.right: deque.append(node.right)
13.             res.append(list(tmp))
14.         return res
```

15.3 Transformer Encoder 和 decoder 的区别

Decoder 和 Encoder 的结构差不多，但是多了一个 attention 的 sub-layer，这里先明确一下 decoder 的输入输出和解码过程：

输出：对应 i 位置的输出词的概率分布输入：encoder 的输出 & 对应 i-1 位置 decoder 的输出。所以中间的 attention 不是 self-attention，它的 K，V 来自 encoder，Q 来自上一位置 decoder 的输出解码：这里要注意一下，训练和预测是不一样的。在训练时，解码是一次全部 decode 出来，用上一步的 ground truth 来预测（mask 矩阵也会改动，让解码时看不到未来的 token）；而预测时，因为没有 ground truth 了，需要一个个预测。

15.4 transformer 对比 lstm 的优势

Transformer 和 LSTM 的最大区别，就是 LSTM 的训练是迭代的、串行的，必须要等当前字处理完，才可以处理下一个字。而 Transformer 的训练时并行的，即所有字是同时训练的，这样就大大增加了计算效率。

15.5 Self-Attention 公式为什么要除以 d_k 的开方

避免出现梯度消失点积注意力被缩小了深度的平方根倍。这样做是因为对于较大的深度值，点积的大小会增大，从而推动 softmax 函数往仅有很小的梯度的方向靠拢，导致了一种很硬的（hard）softmax。例如，假设 Q 和 K 的均值为 0，方差为 1。它们的矩阵乘积将有均值为 0，方差为 d_k 。因此，d_k 的平方根被用于缩放（而非其他数值），因为，Q 和 K 的矩阵乘积的均值本应该为 0，方差本应该为 1，这样会获得一个更平缓的 softmax。

第十六篇 2021 年 9 月初，中兴 AI 算法岗面试题 5 道

16.1 给定图像大小 w，卷积核 k，步长 s，padding，求计算量

字符含义：i：输入的宽度，k:卷积核的宽度，p：单边填充宽度，o：输出宽度，s：步长

卷积的数据关系：o=(i+2p-k)/s+1

16.2 问项目中卷积核大小，是不是越大越好，1*1 的卷积核的作用

考虑到计算量，不是越大越好。

1*1 卷积可以修改通道数

16.3 讲讲你所知道的超参数

批量大小、损失权重比、卷积核个数、学习率、 子模块的个数

16.4 你是怎么进行数据增强的?

使用 tf 自带的 对比度、亮度、饱和度增强方法

和 python imgaug 库的增强方法

16.5 在文本识别中使用大卷积核的好处

增大感受野，可以一次性读取较多的字符，然后进行识别

第十七篇 2021 年 9 月初，科大讯飞 AI 算法岗面试题 3 道

17.1 你对 fast rcnn 了解多少

两阶段目标检测算法

Fast RCNN，是 RCNN 算法的升级版，之所以提出 Fast R-CNN，主要是因为 R-CNN 存在以下几个问题：1、训练分多步。通过上一篇博文我们知道 R-CNN 的训练先要 fine tuning 一个预训练的网络，然后针对每个类别都训练一个 SVM 分类器，最后还要用 regressors 对 bounding-box 进行回归，另外 region proposal 也要单独用 selective search 的方式获得，步骤比较繁琐。2、时间和内存消耗比较大。在训练 SVM 和回归的时候需要用网络训练的特征作为输入，特征保存在磁盘上再读入的时间消耗还是比较大的。3、测试的时候也比较慢，每张图片的每个 region proposal 都要做卷积，重复操作太多。

虽然在 Fast RCNN 之前有提出过 SPPnet 算法来解决 RCNN 中重复卷积的问题，但是 SPPnet 依然存在和 RCNN 一样的一些缺点比如：训练步骤过多，需要训练 SVM 分类器，需要额外的回归器，特征也是保存在磁盘上。因此 Fast RCNN 相当于全面改进了原有的这两个算法，不仅训练步骤减少了，也不需要额外将特征保存在磁盘上。

17.2 ADM 和 SGD

SGD 的一阶动量：

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

加上 AdaDelta 的二阶动量：

$$V_t = \beta_2 * V_{t-1} + (1 - \beta_2) g_t^2$$

17.3 池化的作用

（1）保留主要特征的同时减少参数和计算量，防止过拟合。

（2）invariance(不变性)，这种不变性包括 translation(平移)，rotation(旋转)，scale(尺度)。

第十八篇 2021 年 9 月 10 日， 京东 CV 算法秋招面试题 3 道

18.1 样本不平衡解决

少的一方上采样 或 多的一方下采样

18.2 过拟合的解决办法

增多数据、减小模型复杂度、提高数据增强复杂度

18.3 BN 的作用， 如何做

数据分布更一致，收敛速率增加

可以达到更好的精度

输入： 批处理（mini-batch）输入 $\boldsymbol{x}$：$\mathcal{B}=\{x_{1,\ldots,m}\}$

输出： 规范化后的网络响应 $\{y_i=\mathrm{BN}_{\gamma,\beta}(x_i)\}$

1: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // 计算批处理数据均值

2: $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_{\mathcal{B}})^2$ // 计算批处理数据方差

3: $\hat{x}_i \leftarrow \frac{x_i-\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}}$ // 规范化

4: $y_i \leftarrow \gamma\hat{x}_i+\beta=\mathrm{BN}_{\gamma,\beta}(x_i)$ // 尺度变换和偏移

5: **return** 学习的参数 γ 和 β.

第十九篇 2021 年 9 月， 科大讯飞 CV 岗位面试题 6 道

19.1 常见的 attention 机制，说明 channel attention 和 self attention 的原理

self-attention、channel attention、spatial attention、multi-head attention、transformer

自注意力机制是注意力机制的变体，其减少了对外部信息的依赖，更擅长捕捉数据或特征的内部相关性。

ce loss 的公式，说完了问 BCE loss,就纯背公式

We again use the cross-entropy error function, but it takes a slightly different form. The softmax activation of the ith output unit is

$$y_i = \frac{e^{s_i}}{\sum_c^{nclass} e^{s_c}} \tag{17}$$

and the cross entropy error function for multi-class output is

$$E = -\sum_i^{nclass} t_i \log(y_i) \tag{18}$$

Thus, computing the gradient yields

$$\frac{\partial E}{\partial y_i} = -\frac{t_i}{y_i} \tag{19}$$

$$\frac{\partial y_i}{\partial s_k} = \begin{cases} \frac{e^{s_i}}{\sum_c^{nclass} e^{s_c}} - (\frac{e^{s_i}}{\sum_c^{nclass} e^{s_c}})^2 & i = k \\ -\frac{e^{s_i} e^{s_k}}{(\sum_c^{nclass} e^{s_c})^2} & i \neq k \end{cases} \tag{20}$$

$$= \begin{cases} y_i(1 - y_i) & i = k \\ -y_i y_k & i \neq k \end{cases} \tag{21}$$

$$\frac{\partial E}{\partial s_i} = \sum_k^{nclass} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial s_i} \tag{22}$$

$$= \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial s_i} - \sum_{k \neq i} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial s_i} \tag{23}$$

$$= -t_i(1 - y_i) + \sum_{k \neq i} t_k y_i \tag{24}$$

$$= -t_i + y_i \sum_k t_k \tag{25}$$

$$= y_i - t_i \tag{26}$$

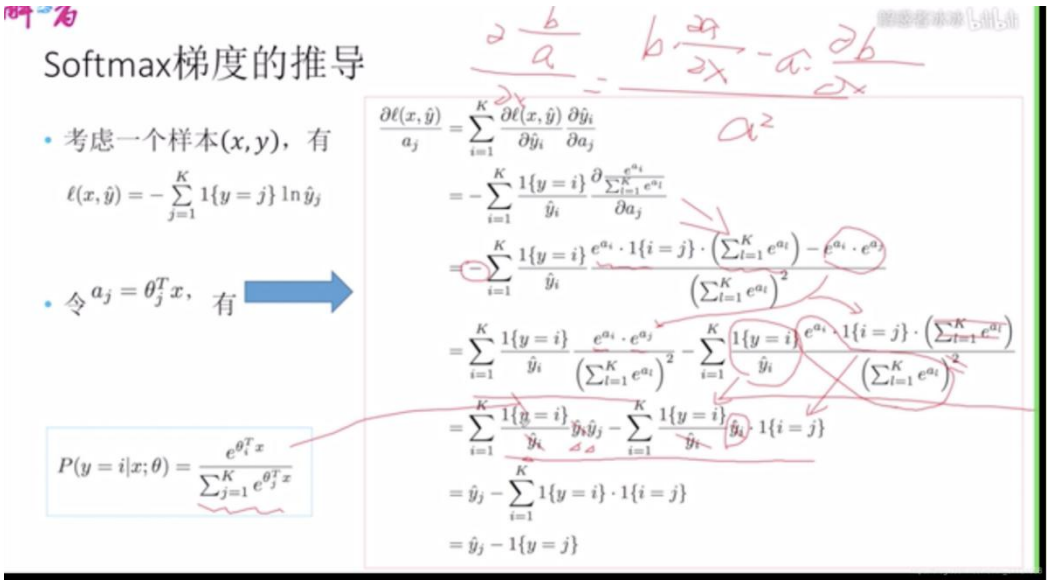
(27)

sigmoid 和 softmax，BCE 与 CE loss function_阿猫的自拍的博客-CSDN 博客_ce loss

19.2 triplet loss 的训练要注意什么

构造类内差异大、类间差异小的数据集

19.3 softmax 求导



19.4 KL 散度

KL 散度可以用来衡量两个概率分布之间的相似性，两个概率分布越相近，KL 散度越小

19.5 检测模型里为啥用 smoothL1 去回归 bbox

从上面的导数可以看出，L2 Loss 的梯度包含 (f(x) - Y)，当预测值 f(x) 与目标值 Y 相差很大时，容易产生梯度爆炸，而 L1 Loss 的梯度为常数，通过使用 Smooth L1 Loss，在预测值与目标值相差较大时，由 L2 Loss 转为 L1 Loss 可以防止梯度爆炸。

19.6 前沿的检测范式 DETR，transformer 等等

与传统的计算机视觉技术不同，DETR 将目标检测作为一个直接的集合预测问题来处理。它由一个基于集合的全局损失和一个 Transformer encoder-decoder 结构组成，该全局损失通过二分匹配强制进行唯一预测。给定固定的学习对象查询集，则 DETR 会考虑对象与全局图像上下文之间的关系，以直接并行并行输出最终的预测集。由于这种并行性，DETR 非常快速和高效。

第二十篇 2021 年 9 月 27 日 - 9 月 30 日， 字节跳动算法岗面试题 5 道

20.1 SVM 相关，怎么理解 SVM，对偶问题怎么来的，核函数是怎么回事。

SVM 是一种二分类模型，它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机。

SVM 为什么要将原始问题转换为对偶问题来求解，原因如下：

- 对偶问题将原始问题中的约束转为了对偶问题中的等式约束；
- 方便核函数的引入；

改变了问题的复杂度。由求特征向量 w 转化为求比例系数 a，在原始问题下，求解的复杂度与样本的维度有关，即 w 的维度。在对偶问题下，只与样本数量有关。

核函数的使用实际上是增加维度，把原本在低维度里的样本，映射到更高的维度里，将本来不可以线性分类的点，变成可以线性分类的。

20.2 集成学习的方式，随机森林讲一下，boost 讲一下，XGBOOST 是怎么回事讲一下。

集成学习的方式主要有 bagging，boosting，stacking 等，随机森林主要是采用了 bagging 的思想，通过自助法（bootstrap）重采样技术，从原始训练样本集 N 中有放回地重复随机抽取 n 个样本生成新的训练样本集合训练决策树，然后按以上步骤生成 m 棵决策树组成随机森林，新数据的分类结果按分类树投票多少形成的分数而定。

boosting 是分步学习每个弱分类器，最终的强分类器由分步产生的分类器组合而成，根据每步学习到的分类器去改变各个样本的权重（被错分的样本权重加大，反之减小）

它是一种基于 boosting 增强策略的加法模型，训练的时候采用前向分布算法进行贪婪的学习，每次迭代都学习一棵 CART 树来拟合之前 t-1 棵树的预测结果与训练样本真实值的残差。

XGBoost 对 GBDT 进行了一系列优化，比如损失函数进行了二阶泰勒展开、目标函数加入正则项、支持并行和默认缺失值处理等，在可扩展性和训练速度上有了巨大的提升，但其核心思想没有大的变化。

20.3 决策树是什么东西，选择叶子节点的评价指标都有什么。对于连续值，怎么选择分割点。

决策树有三种：分别为 ID3，C4.5，Cart 树

ID3 损失函数：

$$
\begin{aligned}
H(D \mid A) &= \sum_{i=1}^{n} \frac{|D_i|}{|D|} H(D_i) \\
&= -\sum_{i=1}^{n} \frac{|D_i|}{|D|} \left(\sum_{k=1}^{K} \frac{|D_{ik}|}{|D_i|} \log_2 \frac{|D_{ik}|}{|D_i|} \right)
\end{aligned}
$$

C4.5 损失函数：

$$
\begin{aligned}
\operatorname{Gain}_{\text{ratio}}(D, A) &= \frac{\operatorname{Gain}(D, A)}{H_A(D)} \\
H_A(D) &= -\sum_{i=1}^{n} \frac{|D_i|}{|D|} \log_2 \frac{|D_i|}{|D|}
\end{aligned}
$$

Cart 树损失函数：

$$
\begin{aligned}
\operatorname{Gini}(D) &= \sum_{k=1}^{K} \frac{|C_k|}{|D|} \left(1 - \frac{|C_k|}{|D|} \right) \\
&= 1 - \sum_{k=1}^{K} \left(\frac{|C_k|}{|D|} \right)^2 \operatorname{Gini}(D \mid A) \\
&= \sum_{i=1}^{n} \frac{|D_i|}{|D|} \operatorname{Gini}(D_i)
\end{aligned}
$$

对于连续值往往通过随机取值、给定采样间隔或者根据样本的值这三种方法的其中一个选择分割点，据称，使用随机取值的办法最终得到的决策树（随机森林）最优。

20.4 模型评价指标都有什么，AUC 是什么，代表什么东西。

准确率：分类正确的样本占总样本的比例

准确率的缺陷：当正负样本不平衡比例时，当不同类别的样本比例非常不均衡时，占比大的类别往往成为影响准确率的最主要因素。

精确率：分类正确的正样本个数占分类器预测为正样本的样本个数的比例；

召回率：分类正确的正样本个数占实际的正样本个数的比例。

F1 score：是精确率和召回率的调和平均数，综合反应模型分类的性能。

Precision 值和 Recall 值是既矛盾又统一的两个指标，为了提高 Precision 值，分类器需要尽量在“更有把握”时才把样本预测为正样本，但此时往往会因为过于保 守而漏掉很多“没有把握”的正样本，导致 Recall 值降低。

AUC 是 ROC 曲线下面的面积，AUC 可以解读为从所有正例中随机选取一个样本 A，再从所有负例中随机选取一个样本 B，分类器将 A 判为正例的概率比将 B 判为正例的概率大的可能性。AUC 反映的是分类

器对样本的排序能力。AUC 越大，自然排序能力越好，即分类器将越多的正例排在负例之前。

20.5 关于样本不平衡都有什么方法处理

常用于解决数据不平衡的方法：

- 欠采样：从样本较多的类中再抽取，仅保留这些样本点的一部分；
- 过采样：复制少数类中的一些点，以增加其基数；
- 生成合成数据：从少数类创建新的合成点，以增加其基数。
- 添加额外特征：除了重采样外，我们还可以在数据集中添加一个或多个其他特征，使数据集更加丰富，这样我们可能获得更好的准确率结果。

第二十一篇 2021 年 9 月 27 日 - 9 月 30 日， 快手社科广告算法岗面试题 5 道

21.1 l1,l2 公式，区别

L1/L2 的区别

- L1 是模型各个参数的绝对值之和。

  L2 是模型各个参数的平方和的开方值。

- L1 会趋向于产生少量的特征，而其他的特征都是 0。

  因为最优的参数值很大概率出现在坐标轴上，这样就会导致某一维的权重为 0 ，产生稀疏权重矩阵

  L2 会选择更多的特征，这些特征都会接近于 0。

  最优的参数值很小概率出现在坐标轴上，因此每一维的参数都不会是 0。当最小化||w||时，就会使每一项趋近于 0。

L1 的作用是为了矩阵稀疏化。假设的是模型的参数取值满足拉普拉斯分布。

L2 的作用是为了使模型更平滑，得到更好的泛化能力。假设的是参数是满足高斯分布。

21.2 二分查找

leetcode704，搜索区间两端闭，while 条件带等号，mid 要加减 1。

代码：

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = left + (right - left)//2
            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid + 1
            elif nums[mid] > target:
                right = mid - 1
        return -1
```

时间复杂度：O(logN)。

空间复杂度：O(1)。

21.3 翻转数组二分查找

该题为 leetcode153 题：数组不包含重复元素，并且只要当前的区间长度不为 1，pivot 就不会与 high 重合；而如果当前的区间长度为 1，这说明我们已经可以结束二分查找了。因此不会存在 nums [pivot] = nums [high] 的情况。

当二分查找结束时，我们就得到了最小值所在的位置。

代码：

```
1.  class Solution:
2.      def findMin(self, nums: List[int]) -> int:
3.          low, high = 0, len(nums) - 1
4.          while low < high:
5.              pivot = low + (high - low) // 2
6.              if nums[pivot] < nums[high]:
7.                  high = pivot
8.              else:
9.                  low = pivot + 1
10.         return nums[low]
```

21.4 决策树都用什么指标，信息增益是什么

信息增益，信息增益率，基尼指数

信息增益是以某特征划分数据集前后的熵的差值，熵可以表示样本集合的不确定性，熵越大，样本的不确定性就越大。

21.5 auc 含义公式

AUC 是 ROC 曲线下面的面积，AUC 可以解读为从所有正例中随机选取一个样本 A，再从所有负例中随机选取一个样本 B，分类器将 A 判为正例的概率比将 B 判为正例的概率大的可能性。AUC 反映的是分类器对样本的排序能力。AUC 越大，自然排序能力越好，即分类器将越多的正例排在负例之前。

公式如下：

$$AUC = \frac{\sum_{ie\ positiveClass} rank_i - \frac{M(1+M)}{2}}{M \times N}$$