2024最新大厂AI面试题

目录

第一篇 2024 年 3 月，美团视觉算法工程师之 AR/VR 面试题 3 道

1.1 编程题

①给你一个有序数组，请构建一棵 二叉树 ；

②中序输出该 二叉树 ；

③求树的深度；

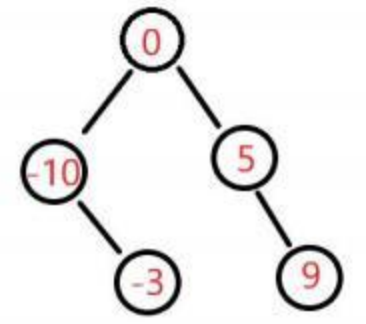
解析：

根据题目描述考察的是 leetcode 中的”将有序数组转换为二叉搜索树“题；

二叉搜索树的中序遍历是有序序列，可以确保数组是二叉搜索树的中序遍历序列，然后依次重建二叉树（如果没有别的要求，则重建的二叉树不唯一），此处给出平衡二叉树的一种解法：

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
def sortedArrayToBST(nums):
    def helper(left, right):
        if left > right:
            return None
        # 总是选择中间位置左边的数字作为根节点
        mid = (left + right) // 2
        root = TreeNode(nums[mid])
        root.left = helper(left, mid - 1)
        root.right = helper(mid + 1, right)
        return root
    return helper(0, len(nums) - 1)
```

创建的二叉树为：



中序输出二叉树：

遍历二叉树时，根据根节点的位置，可以分为前序遍历、中序遍历、后续遍历 3 种；

中序遍历即按照 左子树 = > 根节点 = > 右子树 的顺序遍历二叉树，左右子树遍历依然遵循这个顺序，直至遍历完整课树；

代码逻辑：

inorder(root) 表示当前遍历到 root 节点的值，然后递归调用 inorder(root.left) 遍历 root 节点的左子树，再递归调用 inorder(root.right) 遍历 root 节点的右子树，root 节点是空节点时递归终止。参考代码如下：

```
def inorderTraversal(root):
    def inorder(root, ret):
        if not root:  return
        inorder(root.left, ret)
        ret.append(root.val)
        inorder(root.right, ret)

    ret = []
    inorder(root, ret)
    return ret
```

计算深度：

分别计算左右子树的深度，然后选取最大的作为树的深度，使用递归遍历左右子树；参考代码如下：

```
def calDepth(root):
        if root is None:
            return 0
        else:
            left_depth = calDepth(root.left)
            right_depth = calDepth(root.right)
            return max(left_depth, right_depth) + 1
```

以上完成代码如下，可以在本地电脑运行：

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def sortedArrayToBST(nums):
    def helper(left, right):
        if left > right:
            return None
        # 总是选择中间位置左边的数字作为根节点
        mid = (left + right) // 2

        root = TreeNode(nums[mid])
        root.left = helper(left, mid - 1)
        root.right = helper(mid + 1, right)
        return root

    return helper(0, len(nums) - 1)

def inorderTraversal(root):
```

```
22.
23.     def inorder(root, ret):
24.         if not root:  return
25.         inorder(root.left, ret)
26.         ret.append(root.val)
27.         inorder(root.right, ret)
28.
29.     ret = []
30.     inorder(root, ret)
31.     return ret
32.
33. def calDepth(root):
34.         if root is None:
35.             return 0
36.         else:
37.             left_depth = calDepth(root.left)
38.             right_depth = calDepth(root.right)
39.             return max(left_depth, right_depth) + 1
40. if __name__ == "__main__":
41.     nums = [-10,-3,0,5,9]
42.     # 构建二叉树
43.     tree = sortedArrayToBST(nums)
44.     # 中序遍历
45.     ret = inorderTraversal(tree)
46.     print(ret)
47.     # 二叉树深度
48.     depth = calDepth(tree)
49.     print(depth)
```

1.2 二维平面内给 ABCD 四个点，请判断线段 AB 和线段 CD 是否有交点

解题思路：

假设两线段分别为 AB 和 CD。则 AB 直线的方程为：

$$f_{a,b}(x) = (y - y_a) \times (x_a x_b) - (x - x_a) \times (y_a - y_b) = 0$$

若线段 AB 与线段 CD 相交，则必有

（1）直线 AB 与线段 CD 相交

（2）直线 CD 与线段 AB 相交

判断线段 CD 是否与直线 AB 相交，只需判断：点 C 和点 D 在直线 AB 的不同侧,即 $f_{a,b}(c_x, c_y) \times f_{a,b}(d_x, d_y) \leq 0$, 等于 0 表示点 C 或点 D 在 AB 直线上。于是，若线段 AB 与线

段 CD 相交，则必有：

$$(1) f_{a,b}(c_x, c_y) \times f_{a,b}(d_x, d_y) \leq 0$$

$$(2) f_{c,d}(a_x, a_y) \times f_{c,d}(b_x, b_y) \leq 0$$

上述思路的参考代码如下：

```
#include <stdio.h>
//#include <cstdio.h>

struct Point{
    double x;
    double y;
};

double mult(Point a, Point b, Point c){
    return (a.x - c.x)*(b.y - c.y) - (a.y - c.y)*(b.x - c.x);
}

bool solution(Point aa, Point bb, Point cc, Point dd){
    if (mult(aa, cc, dd)*mult(bb, cc, dd) > 0){
        return false;
    }
    if (mult(cc, aa, bb)*mult(dd, aa, bb) > 0){
        return false;
    }
    return true;
}

int main(){
    Point aa = {0.0, 0.0};
    Point bb = {10.0, 10.0};
    Point cc = {0.0, 0.0};
    Point dd = {10.0, -10.0};
    bool ret;
    ret = solution(aa, bb, cc, dd);
    printf("%d\n", ret);
    return 0;
}
```

1.3 C++值交换

void swap(int *p1， int *p2)

请实现它们值的交换，主要考察你对指针的掌握。参考代码如下：

```
#include <stdio.h>
#include<algorithm>
using namespace std;
void swap(int *p1, int *p2){
        int temp;
        temp = *p1;
        *p1 = *p2;
        *p2 = temp;
}
```

```
10.
11. int main(){
12.         int a = 77, b = 777;
13.         swap(a, b);
14.         printf("%d, %d\n", a, b);
15.         return 0;
16. }
```

第二篇 2024 年 4 月 6 日，一面网易互联网计算机视觉面试题 3 道

2.1 多标签 loss

多标签分类表示一个样本的标签可能有一个，也可能有多个。

假设对图片进行分类，它的标签数目不是固定的，可能是一个标签，也可能是两个标签，所属标签数目不确定，但标签的总数是固定的，比如 10 类。此时 one-hot 编码就不再适用了，可以采用标签补齐的方法，将缺失的标签全部使用 0 标记；比如标签总数为 10，编号为 0-9，一张图片中包含 0，3，9 号标签，则这张图片的标签编码为：1 0 0 1 0 0 0 0 0 1 0.

损失函数选择：在多标签分类任务中，一般采用 sigmoid 作为输出层的激活函数，使用 binary_crossentropy（二分类交叉熵损失函数）作为损失函数，公式如下：

$$L = -\sum_{k} y_k \log\left(\sigma(l_k)\right) + \left(1 - y_k\right) \log\left(1 - \sigma(l_k)\right)$$

其中σ(l x)表示使用 sigmoid 函数激活函数，此时相当于将一个多标签问题转化为了在每个标签上的二分类问题，计算样本各个标签的损失，然后取平均值，得到最后的损失。

2.2 多标签分类准确率

多标签分类评价指标

以下内容参考周志华老师的论文- A Review on Multi-Label Learning Algorithms

Subset Accuracy： 衡量正确率，预测的样本集和真实的样本集完全一样才算正确.

$$\frac{1}{p}\sum_{i=1}^{p} 1\left\{h\left(x^i\right) = y^i\right\}$$

其中 p 表示测试集的样本大小，1{π}表示π为真时返回 1，否则返回 0。

Hamming Loss：衡量的是错分的标签比例，正确标签没有被预测或者错误标签被预测的标签占比。

$$\frac{1}{p}\sum_{i=1}^{p} \frac{1}{q}\left|h\left(x^i\right) \Delta y^i\right|$$

其中Δ表示两个集合的对称差，返回只在其中一个集合出现的那些值。

One-error： 度量的是： "预测到的最相关的标签" 不在 “真实标签” 中的样本占比, 值越小越好.

$$\text{one} \ -\text{error}(f) = \frac{1}{p}\sum_{i=1}^{p} 1\left\{\left[\arg\max_{y_j \in Y} f\left(x^i, y_j\right)\right] \notin y^i\right\}$$

Coverage： 度量的是： “排序好的标签列表” 平均需要移动多少步，才能覆盖真实的相关标签集。

$$\text{coverage}(f) = \frac{1}{p}\sum_{i=1}^{p}\max_{y_j \in Y^i}\text{rank}_f\left(x^i, y_j\right) - 1$$

更多内容可以查看查上述论文。

2.3 数据类别不平衡

类别不平衡是指分类任务中不同类别的训练样本数量差别比较大的情况。

解决办法举例：

阈值移动(threshold-moving)

对于类别平衡的二分类问题， 假设新的测试样本预测为正例的概率为 y， 当 y>0.5 时判别为正例， 否则为反例， 即： 若 $\frac{y}{1-y} > 1$ 则预测为正例, 其中几率 $\frac{y}{1-y}$ 则反映了正例可能性与反例可能性之比值。同样的， 当训练集中正、反例的数目不同时， 令m+表示正例数目， m-表示反例数目。假设训练集是真实样本总体的无偏采样， 则有： $\frac{y}{1-y} > \frac{m^+}{m^-}$ 则预测为正例。

欠采样(undersampling)

欠采样也称**下采样(downsampling)**， 即去除一些多数类别中的样本使得两个类别数目接近， 然后再进行学习。

过采样(oversampling)

过采样也称**上采样(upsampling)**， 即增加一些少数类别中的样本使得两个类别数目接近， 然后再进行学习。

Focal Loss

是目标检测中解决正负样本严重不平衡的方法， 在标准交叉熵损失基础上修改得到的。这个函数可以通过减少易分类样本的权重， 使得模型在训练时更专注于稀疏的难分类的样本； 防止大量易分类负样本在 loss 中占主导地位。

第三篇 2024 年 3 月 29 日-4 月 1 日，伴鱼 AI 算法岗面试题 5 道

3.1 Leetcode 合并重叠区间

也就是那个最大不重叠区间的题目的变种。面试官会让你看题先整理思路，然后分析复杂度，最后面试官觉得你的思路 OK 就放你在远程连接上手撕代码

https：//leetcode-cn.com/problems/merge-intervals/

解题思路：

先对输入数组按照区间左边的值进行升序排列

初始化一个变量 res，用于存储合并直接的区间结果

遍历排序后的所有区间，针对每个区间做如下的处理：

如果当前处理的区间是第一个区间的话，那么直接将区间加入到 res

比较当前处理区间左边的值 i[0] 和 res 中最后一个区间右边的值 res[-1][1]：

如果 res[-1][1] < i[0]，说明没有重叠，那么直接将当前处理的区间加入 res

否则，说明有重叠，那么将 res 中最后一个区间的右边的值更新为：当前处理区间右边值 i[1] 和 res 中最后一个区间右边的值 res[-1][1] 的最大值.

```
class Solution:
    def merge(self, intervals:  List[List[int]]) -> List[List[int]]:
        intervals.sort()
        res = []
        for i in intervals:
            if not res or res[-1][1]<i[0]:
                res.append(i)
            else:
                res[-1][1] = max(res[-1][1],i[1])
        return res
```

3.2 过拟合和欠拟合？

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）

3.3 现在有个模型欠拟合，增加模型深度能改善吗？

可以判断神经网络的拟合能力是否足够： 使用少量数据训练模型， 让每个 epoch 或者每个 batch 都是相同的数据， 然后观察训练的 loss 和 accuracy， 如果发现 loss 下降并且 accuracy 上升了， 训练结束后可以正确预测训练数据， 那么可以确定是模型的拟合能力不足， 无法拟合全部的训练数据， 此时可以增加模型的深度， 增强网络的拟合能力。

3.4 python 多进程

线程

是操作系统中进行运算调度的最小单位。它被包含在进程之中， 是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流， 一个进程可以有多个线程， 每条线程可以同时执行不同的任务。一个线程可以看作一个 cpu 执行时所需要的一串指令。

多线程

在 Python 的标准库中提供了两个模块： _thread 和 threading， _thread 是低级模块不支持守护线程， 当主线程退出时， 所有子线程都会被强行退出。而 threading 是高级模块， 用于对_thread 进行了封装支持守护线程。在大多数情况下我们只需要使用 threading 这个高级模块即可。

进程：

进程指的是一个程序在给定数据集合上的一次执行过程， 是系统进行资源分配和运行调用的独立单位。也就是每一个应用程序都有一个自己的进程。进程在启动时都会最先产生一个线程， 这个线程被称为主线程， 然后再有主线程创建其他子线程

多进程：

多进程是 multiprocessing 模块提供远程与本地的并发， 在一个 multiprocessing 库的使用场景下，所有的子进程都是由一个父进程启动来的， 这个父进程成为 madter 进程， 它会管理一系列的对象状态，一旦这个进程退出， 子进程很可能处于一个不稳定的状态， 所以这个父进程尽量要少做事来保持其稳定性

线程与进程的区别

（1）线程必须在某个进程中执行。一个进程可包含多个线程， 并且只有一个主线程。

（2）多线程共享同个地址空间、打开的文件以及其他资源。而多进程共享物理内存、磁盘、打印机以及其他资源。

（3）线程几乎不占资源， 系统开销少， 切换速度快， 而且同个进程中的多个线程可以实现数据共享，而进程之间是不可共享的

（4）新线程的创建很简单而新进程的创建需要对父进程进行克隆

（5）一个线程可以控制和操作同一进程里的其他线程； 但是进程只能操作子进程

参考 ： https：//www.php.cn/python-tutorials-416932.html

3.5 python 深浅拷贝

不可变类型（字符串、数值型、布尔值）的深浅拷贝

不可变对象的浅拷贝， 以数值为例

```
import copy
a=5
# 浅拷贝
b=copy.copy(a)
print(id(a))
print(id(b))

# 修改 a 的值
a=10
print(id(a))
print(id(b))
# 140732266387232
# 140732266387232
# 140732266387392
# 140732266387232

# 不可变对象的深拷贝，以数值为例
import copy
a=5
# 深拷贝
b=copy.deepcopy(a)
print(id(a))
print(id(b))

# 修改 a 的值
a=10
print(id(a))
print(id(b))

# 140732266387232
# 140732266387232
# 140732266387392
# 140732266387232
```

对于不可变类型（字符串、数值型、布尔值）：浅拷贝和深拷贝一样， 对象的引用（内存地址）没有发生变化。

可变对象（列表、字典、集合）的浅拷贝

```
import copy
lsta = [1, 2, 3]
lstb = [1, 2, [3, 4, 5]]
lsta1 = copy.copy(lsta) # 非嵌套浅拷贝
lstb1 = copy.copy(lstb) # 嵌套列表浅拷贝
print(id(lsta))
print(id(lsta1))
print(id(lsta[0]), id(lsta[1]),id(lsta[2]))
print(id(lsta1[0]), id(lsta1[1]), id(lsta1[2]))
# 1373054849088
# 1373054934336
```

```
# 140732266387104 140732266387136 140732266387168
# 140732266387104 140732266387136 140732266387168
print(id(lstb))
print(id(lstb1))
print(id(lstb[0]), id(lstb[2]), id(lstb[2][0]))
print(id(lstb1[0]), id(lstb1[2]), id(lstb1[2][0]))
# 2735667449472
# 2735667458368
# 140732266387104 2735667382912 140732266387168
# 140732266387104 2735667382912 140732266387168
```

可变对象（列表、字典、集合）的深拷贝

```
import copy
lsta = [1, 2, 3]
lstb = [1, 2, [3, 4, 5]]
lsta1 = copy.deepcopy(lsta) # 非嵌套深拷贝
lstb1 = copy.deepcopy(lstb) # 嵌套列表深拷贝

print(id(lsta))
print(id(lsta1))
print(id(lsta[0]), id(lsta[1]),id(lsta[2]))
print(id(lsta1[0]), id(lsta1[1]), id(lsta1[2]))

# 2310617518336
# 2310617599488
# 140732194690720 140732194690752 140732194690784
# 140732194690720 140732194690752 140732194690784

print(id(lstb))
print(id(lstb1))
print(id(lstb[0]), id(lstb[2]), id(lstb[2][0]))
print(id(lstb1[0]), id(lstb1[2]), id(lstb1[2][0]))

# 2310617599744
# 2310617608704
# 140732194690720 2310617532800 140732194690784
# 140732194690720 2310617608640 140732194690784
```

对于可变对象（列表、字典、集合）：浅拷贝在拷贝时，只会 copy 一层，在内存中开辟一个空间，存放这个 copy 的列表。更深的层次并没有 copy，即第二层用的都是同一个内存；深拷贝时，会逐层进行拷贝，遇到可变类型，就开辟一块内存复制下来，遇到不可变类型就沿用之前的引用。因为不可变数据修改会从新开辟新的空间，所以，深拷贝数据之间的修改都不会相互影响。

总结如下：

浅拷贝花费时间少，占用内存少，只拷贝顶层数据，拷贝效率高。

对不可变对象拷贝时，浅拷贝和深拷贝的作用是一致的，不开辟新空间，相当于赋值操作。

可变对象浅拷贝时，只拷贝第一层中的引用，如果元素是可变对象，并且被修改，那么拷贝的对象也会发生变化。

可变对象深拷贝时，会逐层进行拷贝，遇到可变类型，就开辟一块内存复制下来。

第四篇 2024 年 3 月底， 美团实习算法岗面试题 4 道

4.1 resnet 和 transfomer 模型结构

ResNet 模型结构：

ResNet 是为解决深度网络中出现网络退化问题， 针对退化问题， 何凯明提出了残差学习模块；

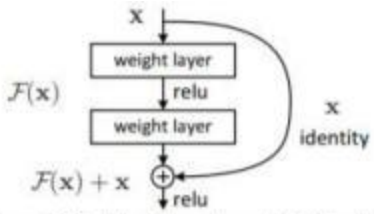
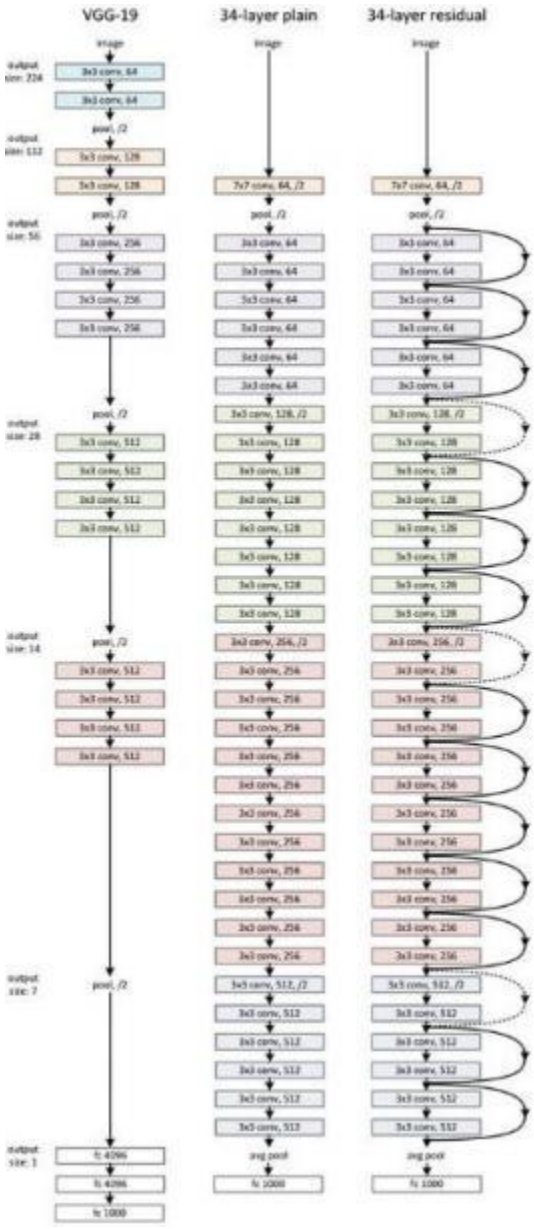


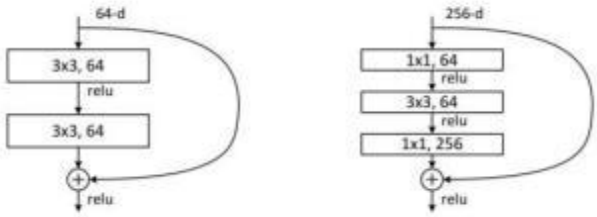
Figure 2. Residual learning: a building block.

对于一个**堆积层结构**， 当输入为 x 时， 其学习到的特征记为 H() ， 现在我们希望其可以学习到残差F()=H(q)-x ， 这样其实原始的学习特征是F()+z 。这样设计网络的原因是因为残差学习相比原始特征直接学习更容易。当残差为 0 时， 此时堆积层仅仅做了恒等映射， 网络性能不会下降， 实际训练时， 残差往往不会等于 0， 可以让堆积层在输入特征基础上学习到新的特征， 从而网络特征提取能力更强。残差学习的结构如上图所示， 这种结构称为跳转链（ skip connection ）或者短路连接（ shortcut connection ）。



ResNet 参考了 VGG19 的网络结构(上图左)，并在其基础上增加了一些修改，然后通过残差机制增加了残差单元，如上图右所示。变化主要体现在 ResNet 直接使用 stride=2 的卷积做下采样，并且用 global average pool 层替换了全连接层。

ResNet 的一个重要设计原则是：当 Feature Map 大小降低一半时，Feature Map 的数量增加一倍，以网络层的复杂度。从上图中可以看到，ResNet 相比普通网络每两层间增加了残差结构，其中虚线表示 Feature Map 数量发生了改变。图右展示的是 34 层的 ResNet 网络，还可以构建更深的网络如表 1 所示。从表中可以看到，对于 18-layer 和 34-layer 的 ResNet，其进行的两层 layer 间的残差学习（如下图左），当网络更深时，其进行的是三层 layer 间的残差学习，三层卷积的卷积核分别是 1x1，3x3 和 1x1（如下图右）。



ResNet 网络中的残差单元有两种， 如上图所示。左图对应的是浅层网络， 右图对应的是深层网络。对于跳转连接， 当输入和输出维度一致时， 可以直接将输入加到输出上。但是当维度不一致时， 有**两种策略**：

（ 1 ）采用 zero-padding 增加维度， 此时一般要先做一个 downsamp， 可以采用 strde=2 的 pooling， 这样不会增加参数；

（ 2 ）采用新的映射（ projection shortcut ），一般采用 1x1 的卷积， 这样会增加参数， 也会增加计算量。跳转连接除了直接使用恒等映射， 当然都可以采用 projection shortcut。

Transformer 结构：

Transformer 中抛弃了传统的 CNN 和 RNN 结构， 整个网络结构完全是由 Attention 机制模块组成。准确地讲， Transformer 由且仅由 self-Attenion 和 Feed Forward Neural Network 组成。一个基于 Transformer 的可训练的神经网络可以通过堆叠 Transformer 的形式进行搭建， 作者的实验中， 编码器和解码器各搭建了 6 层， 总共 12 层的 Encoder-Decoder， 并在机器翻译中取得了 BLEU 值新高。（**BLEU：机器翻译评测指标**）

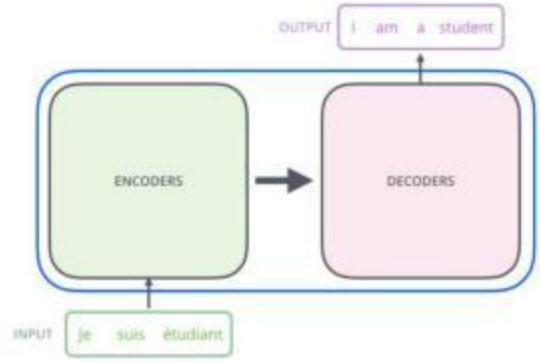
作者采用 Attention 机制的原因：考虑到 RNN（ LSTM、GRU 等）的计算限制（训练时只能顺序执行），即 RNN 相关算法只能从左向右依次计算或者从右向左依次计算， 这种机制带来了两个问题：

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

顺序计算的过程中信息会丢失， 尽管 LSTM 等门控机制结构在一定程度上缓解了长期依赖的问题，但是对于特别长期的依赖现象， LSTM 依旧不能处理。

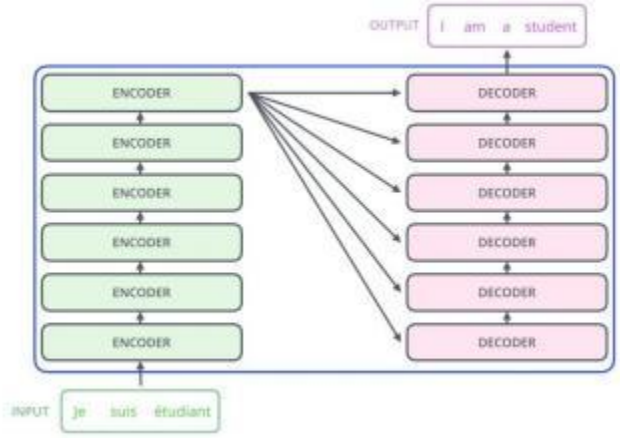
Transformer 的提出解决了上面两个问题， 首先它使用了 Attention 机制， 将序列中任意两个位置之间的距离缩小为一个常量； 其次它不是类似 RNN 的顺序结构， 具有更好的并行性, 符合现有的 GPU 框架;可以并行处理数据, 加快计算速度。

Transformer 的本质上是一个 Encoder-Decoder 的结构, 可以抽象的理解为下图结构：

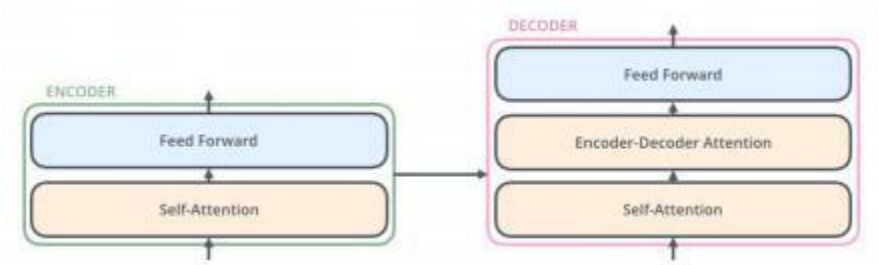


论文中设置： 编码器由 6 个编码 block 组成， 同样解码器是 6 个解码 block 组成； 编码器的输出会

作为解码器的输入，如下图所示



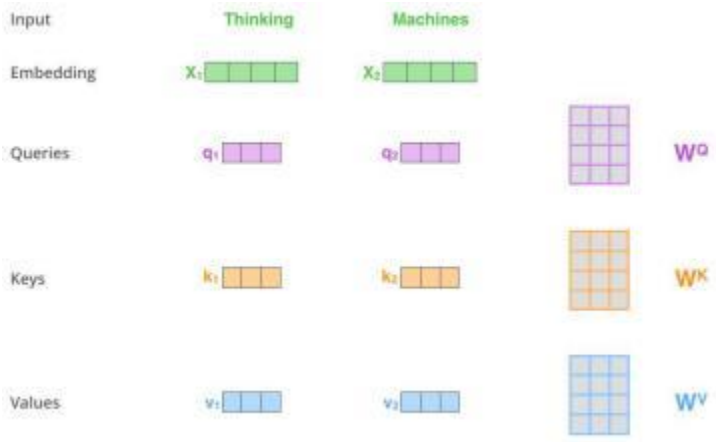
Encoder 编码器(如下图左)由 Self-Attention 结构和 FFN 前向传播网络组成，Decoder 的结构如下图右所示，和 Encoder 的不同之处在于 Decoder 多了一个 Encoder-Decoder Attention，两个 Attention 分别用于计算输入和输出向量的权值：



Self-Attention：当前翻译和已经翻译的前文之间的关系；

Encoder-Decnoder Attention：当前翻译和编码的特征向量之间的关系。

Attention 计算步骤：



如上图， 将输入单词转化成嵌入向量；

根据嵌入向量得到 q、k、v 三个向量；

为每个向量计算一个 score： socre=q*k；

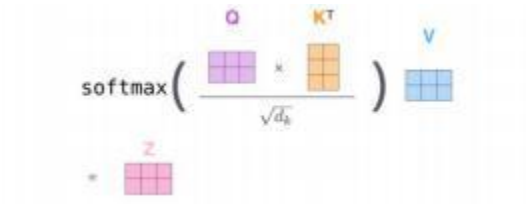
为了梯度的稳定， Transformer 使用了 score 归一化， 即除以 $\sqrt{d_k}$ ；

对 score 施以 softmax 激活函数；

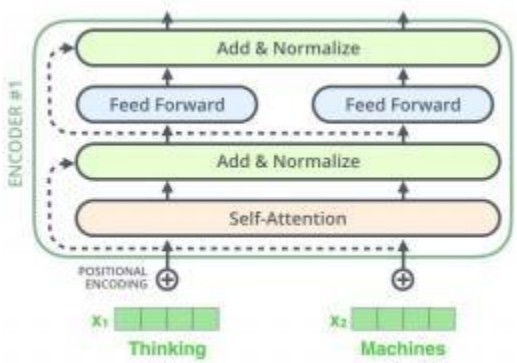
softmax 点乘 Value 值 v， 得到加权的每个输入向量的评分 v；

相加之后得到最终的输出结果 z ： $z=\sum v$。

用矩阵乘法可以表示为：



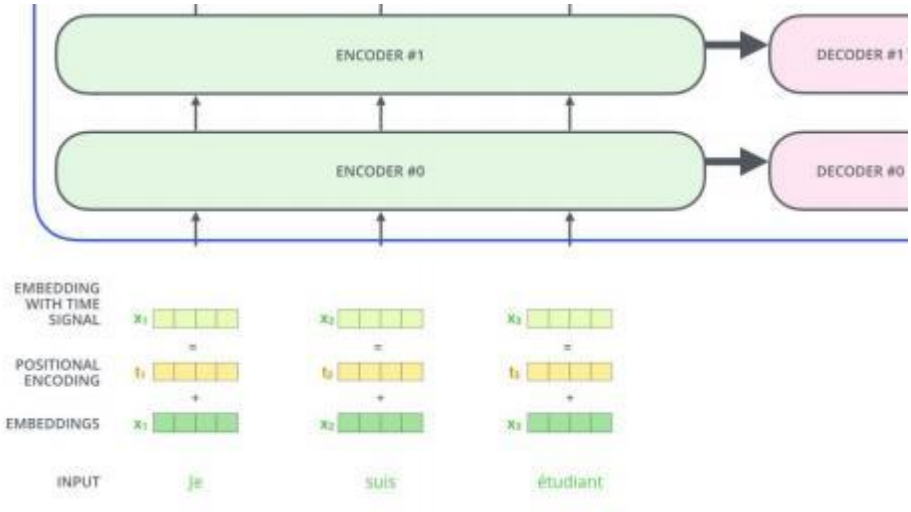
在 Self-Attention 中也使用了残差网络结构-图中虚线部分：



位置编码：

此时的 Transformer 模型并没有捕捉顺序序列的能力， 即目前的 Transformer 只是一个功能更强大的词袋模型而已。为了解决这个问题， 论文中在编码词向量时引入了位置编码 （ Position Embedding ）的特征。网络输入时， 在词向量中加入单词的位置编码信息， 这样 Transformer 就能区分不同位置的单词。

这里的位置编码规则时作者自己设计的， 通常位置编码是一个长度为 d_model 的特征向量， 这样便于和词向量进行单位加的操作 ， 如下所示：



论文中给出的公式如下所示：

$$PE(pos,2i)=sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

$$PE(pos,2i+1)=cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$

其中 pos 表示单词位置， i 表示单词维度

作者这么设计的原因是考虑到在 NLP 任务中， 除了单词的绝对位置， 单词的相对位置也非常重要。根据公式 $sin(a+F)=sinacosf+cosasinfl$ $cos(a+B)=cosacosB-sinasinf$ ， 这表明位置 $k+p$ 的位置编码向量可以表示为位置 k 和 p 的编向量的线性组合， 这为模型捕捉单词之间的相对位置关系提供了非常大的便利。

更多请参见七月在线题库里的这题：《请详细说说 Transformer（超详细图解，一图胜千言）》 。

4.2 手撕快排

对数据 arr =[3， 2， 1， 5， 0， 4]使用快排进行排序

排序逻辑： 从 arr 中随机选取一个数定义为 pvoite(假定选择左端点得值 arr[0])

小放左 大放右： 将 arr 中小于 pivote 的数放在 pivote 的左边， 大于 pivote 的数放在 pivote 的右边

对于左边的数组重新选取 pivote 并小放左、大放右(递归)

对于右边的数组重新选取 pivote 并小放左、大放右(递归)

排序样例-双指针法

指针初始化： left 指针为当前 arr 开始位置， right 指针为当前 arr 的末尾位置；

从 **right 指针**位置开始判断：

如果 arr[right] >= pivote && left < right, 则 right--

如果 arr[right] < pivote && left < right, 则 arr[left] = arr[right], left++

然后从 **left 指针**处开始判断：

如果 arr[left] <= pivote && left < right, 则 left++

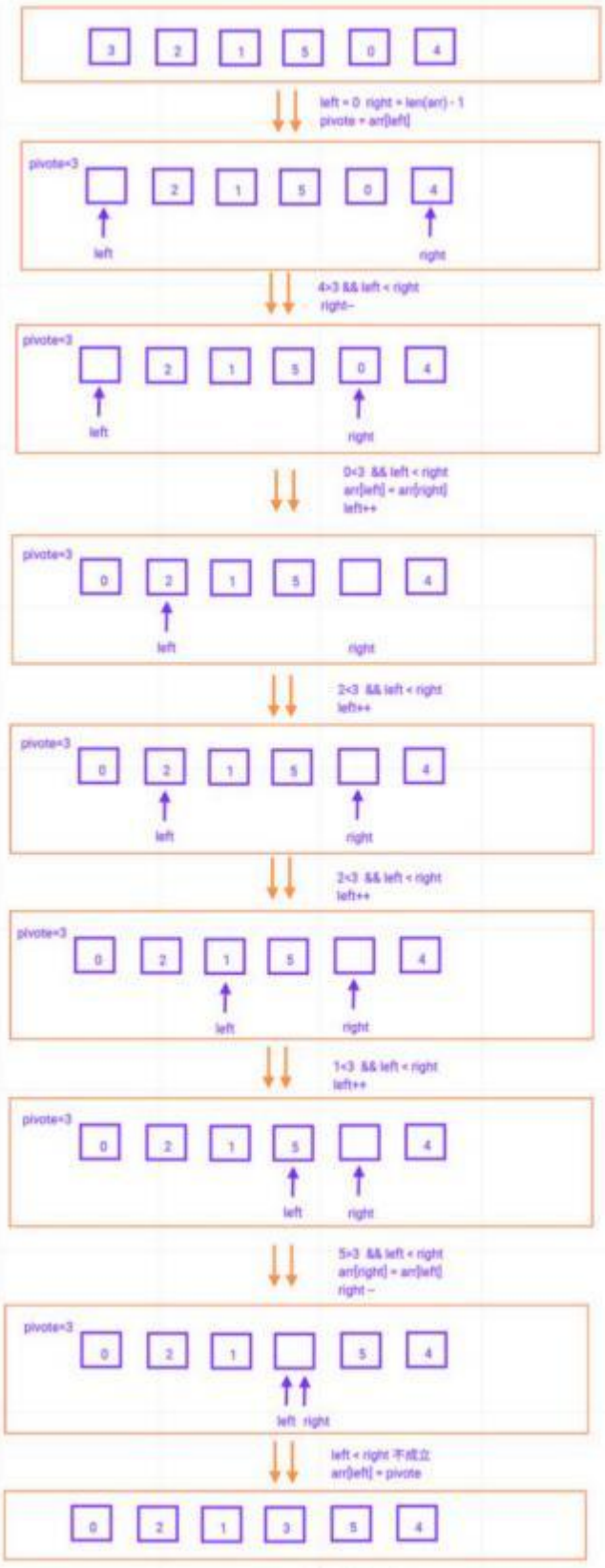
如果 arr[left] > pivote && left < right, 则 arr[right] = arr[left], right++

然后再从 right 指针处开始判断， 左右指针交替进行；

终止条件： 不满足 left < right 时 结束， arr[left] = pivote,（ 此时： pivote 左边都是小于 pivote 的数， 右边都是大于 pivote 的

第一次快排结束， 然后对左右两边递归进行上述过程完成整体排序。

上述过程图示如下：


3 2 1 5 0 4
left = 0 right = len(arr) - 1
pivote = arr[left]
pivote=3
2 1 5 0 4
left
right
4>3 && left < right
right--
pivote=3
2 1 5 0 4
left
right
0<3 && left < right
arr[left] = arr[right]
left++
pivote=3
0 2 1 5 4
left
right
2<3 && left < right
left++
pivote=3
0 2 1 5 4
left
right
2<3 && left < right
left++
pivote=3
0 2 1 5 4
left
right
1<3 && left < right
left++
pivote=3
0 2 1 5 4
left
right
5>3 && left < right
arr[right] = arr[left]
right --
pivote=3
0 2 1 5 4
left right
left < right 不成立
arr[left] = pivote
0 2 1 3 5 4

整体排序过程如下：

代码样例：

```
# 指针 left：左=>右找大于 pivote 的值
# 指针 right：右=>左找小于 pivote 的值
def getPartitionIndex(arr, left, right):
    pivote = arr[left]
    while left < right:
        while arr[right] >= pivote and right > left:
            right -= 1
        arr[left] = arr[right]
        while arr[left] <= pivote and right > left:
            left += 1
        arr[right] = arr[left]
    arr[left] = pivote
    return left
def quickSort(arr, left=None, right=None):
    # print(arr)
    left = 0 if not isinstance(left, (int, float)) else left
    right = len(arr)-1 if not isinstance(right, (int, float)) else right

    if left < right:
        partitionIndex = getPartitionIndex(arr, left, right)
        quickSort(arr, left, partitionIndex)
        quickSort(arr, partitionIndex+1, right)
    return arr

if __name__ == '__main__':
    arr = [3, 2, 1, 5, 0, 4]
print(quickSort(arr))
```

另外，七月在线题库里也有一篇关于快速排序的解析，可以看下请手写下快速排序。

4.3 CNN 和 RNN 的优缺点

CNN 优缺点：

优点：

(1) 参数共享、减少了网络自由参数的个数，对高维数据处理无压力；

(2) 无需手动选取特征，训练好权重，即得特征；

(3) 网络结构在有监督的方式下可以学习到良好的性能： 对平移、比例缩放、倾斜或其他形式的变形具有高度不变性。

缺点：需要调参，需要大量样本；

RNN 优缺点：

优点

(1) 可以拟合序列数据；

(2) LSTM 等 RNN 网络通过遗忘门和输出门忘记部分信息来解决梯度消失的问题；

缺点：

(1) LSTM 虽然部分解决了 RNN 梯度消失问题，但是特征信息在长时传播中损失很厉害；

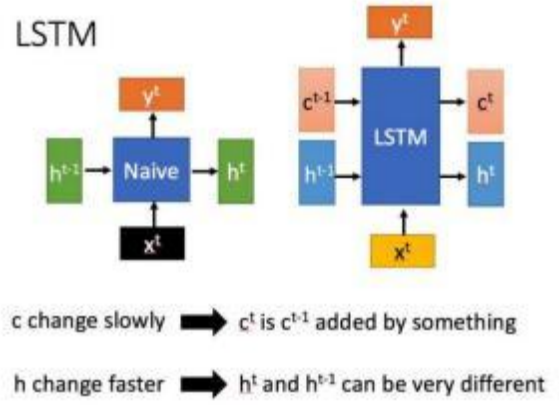
(2) RNN 只能顺序执行，无法很好的并行计算（在工业上影响较大）；

4.4 LSTM 介绍 内部结构

介绍

长短期记忆（Long short-term memory, LSTM）是一种特殊的 RNN，主要是为了解决长序列训练过程中的梯度消失和梯度爆炸问题。简单来说，就是相比普通的 RNN，LSTM 能够在更长的序列中有更好的表现。

LSTM 结构（图右）和普通 RNN 的主要输入输出区别如下所示。

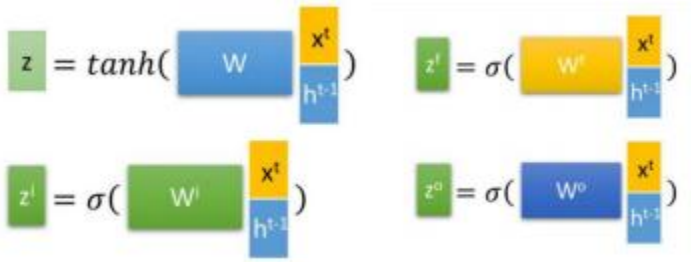


相比 RNN 只有一个传递状态 h^t，LSTM 有两个传输状态，一个 c^t（cell state）和一个 h^t（hidden state）。（Tips：RNN 中的 h^t 对应于 LSTM 中的 c^t）

其中对于传递下去的 c^t，数值改变很慢，通常输出的 c^t 是上一个状态传过来的 c^(t-1)加上一些数值。而 h^t 则在不同节点下往往会有很大的区别。

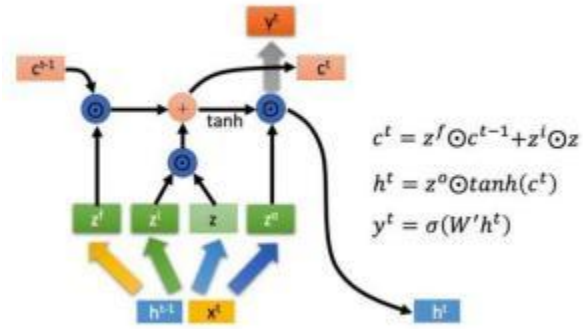
内部结构

首先使用 LSTM 的当前输入 x^t 和上一个状态传递下来的 h^(t-1)拼接训练得到四个状态。



其中 z ^f、z ^i、z ^o 是由拼接向量乘以相应权重矩阵之后， 再通过一个 Sigmoid 激活函数转换成 0 到 1 之间的数值， 来作为一种门控状态。而 z 则是将计算结果通过一个 tanh 激活函数将转换成-1 到 1 之间的值.

门控单元的作用



LSTM 内部主要有三个门控单元：

1. 忘记门： 这个阶段主要是对上一个节点传进来的输入进行选择性忘记。简单来说就是会 “忘记不重要的， 记住重要的” 。具体来说是通过计算得到的 z ^f（f 表示 forget ）来作为忘记门控， 来控制上一个状态的 c^(t-1)哪些需要留哪些需要忘。

2. 选择记忆门。这个阶段将这个阶段的输入有选择性地进行 “记忆” 。主要是会对输入 x 进行**选择记忆**。哪些重要则着重记录下来， 哪些不重要， 则少记一些。当前的输入内容由前面计算得到的 z 表示。而选择的门控信号则是由 z ^i （ i 代表 information ）来进行控制。

将上面两步得到的结果相加， 即可得到传输给下一个状态的 c^T。也就是上图中的第一个公式。

3. 输出门。这个阶段将决定哪些将会被当成当前状态的输出。主要是通过 z ^o 来进行控制的。并且还对上一阶段得到的 c^o 进行了放缩（通过一个 tanh 激活函数进行变化）。

与普通 RNN 类似， 输出 y^t 往往最终也是通过 h^t 变化得到。

以上就是 LSTM 的内部结构， 通过门控状态来控制传输状态， 可以记住需要长时间记忆的内容， 忘记不重要的信息。

此外， 这篇文章也可以看下： [如何从 RNN 起步，一步一步通俗理解 LSTM（全网最通俗的 LSTM 详解）]。

第五篇 2024 年 3 月底 4 月初美团优选 NLP 算法岗面试题 10 道

5.1 求点到圆心距离的期望

一个单位圆内随机取一点， 求到圆心距离的期望。 用代码模拟一下结果， 然后再用数学推导结果。

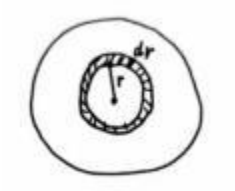
(代码模拟直接随机取坐标(x, y) (x,y in [-1, 1]， 圆心为(0,0)， 模拟 1000 词， 求均值结果)

(数学推导， 求分布函数 -> 密度函数 -> 积分) 答案是 2/3

求期望， 就是找概率， 该点到圆心的距离为 r 的概率为：

$$P(X=r)=\frac{2\pi r dr}{\pi 1^2}$$

式中， 分子为该点落到圆心为 r 的小圆环的面积， 如下图阴影部分， 分母是整个单位圆的面积。



下面求点到圆心的距离的期望。

$$E(X)=\int_0^1 P(X=r)r=\frac{2}{3}$$

代码如下：

```python
import numpy as np
import random
import math

def random_point(num_of_points, radius):
    points = []
    for i in range(0, num_of_points):
        theta = random.random() * 2 * np.pi
        r = random.random() * radius
        x = math.cos(theta) * (r ** 0.5)
        y = math.sin(theta) * (r ** 0.5)
        points.append([x, y])
    return points

def main():
    num_of_points = 10000
    points = random_point(num_of_points, 1)
```

```
    res = 0
    for i in points:
        res += (i[0] ** 2 + i[1] ** 2) ** 0.5
    print(res / num_of_points)

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

5.2 字符串内含有小中大括号和其他字符，判断括号是否匹配

参考答案：

此题出自 leetcode 的 20 题-判断有效括号，只需要将字符串中的“其他字符”去除，那么就和 leetcode 上的一样了：

使用栈解题思路：

定义一个栈 stack = []，用于存储括号的左半部分（" (, { , [" ）；定义一个字典 键为括号的右半部分，值为括号的左半部分；

遍历整个字符串，遇到左括号就保存在 stack 中；

遇到右括号就确认与 stack 最顶端的元素是否匹配，若不匹配或者 stack 为空，则返回 False；

遍历整个字符串后检查 stack，正常情况此时栈为空，若不为空 则返回 False；

Python 参考代码如下：

```
class Solution:
    def isValid(self, s:  str) -> bool:
        # 定义栈
        stack = []
        # 括号字符列表
        chars = ["(", ")", "{", "}", "[", "]"]
        # 括号匹配对
        pair = {
            ")" :  "(",
            "}" :  "{",
            "]" :  "["
        }
        # 遍历整个字符串
        for char in s:
            # 去除"其他字符"
            if char not in chars:  continue
            # 右括号：栈为空或 栈顶元素不匹配则返回 False，  否则弹出栈顶元素
            if char in pair:
                if not stack or stack[-1] != pair[char]:  return False
                stack.pop()
            else:
                # 将左括号压入栈
                stack.append(char)
        # 左右括号如果全部是匹配的 那么此时栈应该是空的
        # 栈为空 返回 True 不为空返回 False:
        return not stack
```

5.3 介绍一下 Transformer 及可以调整的参数

Transformer 结构， 可参考前面已有答案。

可调整的参数： 训练数据集中最长句子的长度， beam_size（候选集的个数） 和 max_out_len（ 最大输出的句子长度）。

关于 Transformer ， 可以看下此文： 《请详细说说 Transformer（超详细图解，一图胜千言）。

5.4 具体讲一下 Self Attention

参考答案：

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。

自注意力机制在文本中的应用， 主要是通过计算单词间的互相影响， 来解决长距离依赖问题。

自注意力机制的计算过程：

1.将输入单词转化成嵌入向量；

2.根据嵌入向量得到 q， k， v 三个向量；

3.为每个向量计算一个 score： score =q . k；

4.为了梯度的稳定， Transformer 使用了 score 归一化， 即除以 $\sqrt{d_k}$ ；

5.对 score 施以 softmax 激活函数；

6.softmax 点乘 Value 值 v， 得到加权的每个输入向量的评分 v；

7.相加之后得到最终的输出结果 z： z= Σv。

5.5 讲一下 Attention

参考答案：

Attention 是从大量信息中有筛选出少量重要信息， 并聚焦到这些重要信息上， 忽略大多不重要的信息。权重越大越聚焦于其对应的 Value 值上， 即权重代表了信息的重要性， 而 Value 是其对应的信息。

计算过程可以概括为三个部分：

第一个部分： **根据 Query 和某个 Keyi， 计算两者的相似性或者相关性**， 最常见的方法包括： 求两者的向量点积、求两者的向量 Cosine 相似性或者通过再引入额外的神经网络来求值；

第二个部分： **采用 SoftMax 的计算方式对第一阶段的得分进行数值转换**， 一方面可以进行归一化， 将原始计算分值整理成所有元素权重之和为 1 的概率分布； 另一方面也可以通过 SoftMax 的内在机制更加突出重要元素的权重。

第三个部分： 计算结果 ai 即为 Valuei 对应的权重系数， 然后进行加权求和即可得到 Attention 数值。

5.6 Self attention、Attention 及 双向 LSTM 的区别

参考答案：

传统的 Attention 机制在一般任务的 Encoder-Decoder model 中， 输入 Source 和输出 Target 内容是不一样的， 比如对于英-中机器翻译来说， Source 是英文句子， Target 是对应的翻译出的中文句子， Attention 机制发生在 Target 的元素 Query 和 Source 中的所有元素之间。简单的讲就是 Attention 机制中的权重的计算需要 Target 来参与的， 即在 Encoder-Decoder model 中 Attention 权值的计算不仅需要 Encoder 中的隐状态而且还需要 Decoder 中的隐状态。

而 Self Attention 顾名思义， 指的不是 Target 和 Source 之间的 Attention 机制， 而是 Source 内部元素之间或者 Target 内部元素之间发生的 Attention 机制， 也可以理解为 Target =Source 这种特殊情况下的注意力计算机制。例如在 Transformer 中在计算权重参数时将文字向量转成对应的 KQV， 只需要在 Source 处进行对应的矩阵操作， 用不到 Target 中的信息。

Self Attention 后会更容易捕获句子中长距离的相互依赖的特征， 因为如果是 RNN 或者 LSTM， 需要依次序序列计算， 对于远距离的相互依赖的特征， 要经过若干时间步步骤的信息累积才能将两者联系起来， 而距离越远， 有效捕获的可能性越小。

5.7 常见激活函数及其特点

常见的激活函数有哪些？各自有什么特点。分布应用于场景。leaky relu 公式。

参考答案：

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。

5.8 layer norm 和 batch norm 的区别

layer norm 和 batch norm 的区别。和各自的应用场景。已经各自在 norm 之后还有没有其他操作。训练时候和测试时候的区别。

参考答案：

Batch Normalization 的处理对象是对一批样本，Layer Normalization 的处理对象是单个样本。Batch Normalization 是对这批样本的同一维度特征做归一化，Layer Normalization 是对这单个样本的所有维度特征做归一化。

BatchNorm 的缺点：

1.需要较大的 batch 以体现整体数据分布

2.训练阶段需要保存每个 batch 的均值和方差，以求出整体均值和方差在 infrence 阶段使用

3.不适用于可变长序列的训练，如 RNN

Layer Normalization

Layer Normalization 是一个独立于 batch size 的算法，所以无论一个 batch 样本数多少都不会影响参与 LN 计算的数据量，从而解决 BN 的两个问题。**LN 的做法是根据样本的特征数做归一化。**

LN 不依赖于 batch 的大小和输入 sequence 的深度，因此可以用于 batch-size 为 1 和 RNN 中对边长的输入 sequence 的 normalize 操作。但在大批量的样本训练时，效果没 BN 好。

实践证明，LN 用于 RNN 进行 Normalization 时，取得了比 BN 更好的效果。但用于 CNN 时，效果并不如 BN 明显。

5.9 编程题：字符串最长公共子序列

参考答案：

此题来源于 leetcode 1143. 最长公共子序列 ，解析如下：

最长公共子序列问题是典型的二维动态规划问题。

假设字符串 text1 和 text2 的长度分别为 m 和 n，创建 m+1 行 n+1 列的二维数组 dp，其中 dp[i][j] 表示 text1[0： i] 和 text2[0： j] 的最长公共子序列的长度。

考虑动态规划的边界情况：

当 i = 0 时，text1[0： i]为空，故对于任意 0 ≤ j ≤ n，有 dp[0][j] = 0；

当 j = 0 时，text2[0： j]为空，故对于任意 0 ≤ i ≤ m，有 dp[i][0] = 0；

因此动态规划的边界情况是：当 i = 0 或 j = 0 时，dp[i][j] = 0。

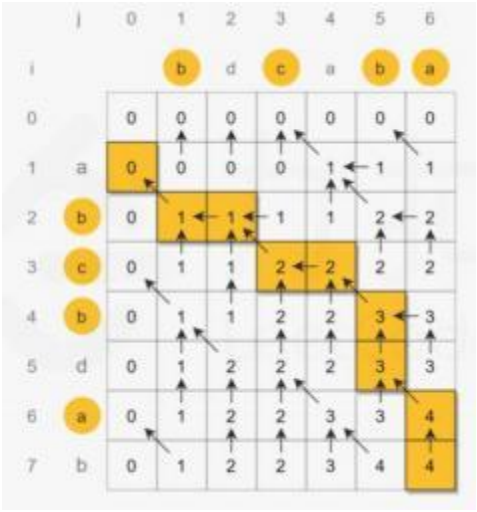
因此动态规划的边界情况是：当 i = 0 或 j = 0 时，dp[i][j] = 0。

当 i > 0 且 j > 0 时，考虑 dp[i][j] 的计算为如下状态转移方程：

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & text_1[i-1] = text_2[j-1] \\ \max(dp[i-1][j], dp[i][j-1]), & text_1[i-1] \neq text_2[j-1] \end{cases}$$

即：当两个字符相同时，公共字符长度加 1；当两个字符不相同时，取长度较长的一项。

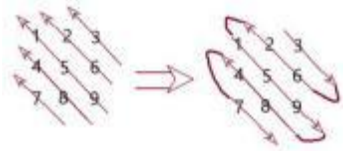
最终计算得到 dp[m][n] 即为 text 1 和 text 2 的最长公共子序列的长度。



代码如下：

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
        return dp[m][n]
```

5.10 二维数组从右上角开始 按照对角线的方式打印输出



假设 m 行 n 列的数组为 arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]) 如上图，

解题思路：

首先从右上角对角线开始，打印出所有的对角线元素；

每条对角线的端点坐标：（0，n-1）=>（m-1, n-1）=>(m-1, 0)

假设端点坐标为(i, j)，则对角线元素为：(i, j) => (i-1, j-1) => (0, *) 或者 (*, 0)

然后将奇数对角线的元素顺序反转；

参考代码：

```
import numpy as np
```

```
arr = np.array([
          [ 1, 2, 3 ],
          [ 4, 5, 6 ],
          [ 7, 8, 9 ]
      ])
m, n = arr.shape
# 保存最后结果
ret = []
for row in list(range(m+n-1)):
    # 反转元素的开始位置
    ret_count = len(ret)
    # i, j 每个对角线的开始位置
    i = row if row + 1 <= m else m - 1
    j =  n-1 if row + 1 <= m else (m + n - 1) - row - 1
    # 获取每条对角线上的元素
    while i >= 0 and j >= 0:
        ret.append(arr[i][j])
        i -= 1
        j -= 1
    # 奇数行顺序反转
    if row % 2 == 0:
        ret = ret[: ret_count] + ret[ret_count: ][: : -1]
print(ret)])
```

第六篇 2024 届作业帮提前批面试题 2 道

6.1 介绍项目

介绍逻辑：

a. 项目背景， 要解决什么问题；

b. 具体思路， 怎么解决问题；

c. 困难， 遇到什么困难， 怎么解决的；

d. 收获；

介绍项目的时候也基本按照这个套路介绍， 关键还是如何解决困难。

要注意， 每个自己说到的 Highlight， 都有可能被深挖， 被深挖的点如下：

CNN

（1）**pooling 的作用**

答案： Pooling 层的作用有如下 3 个：

增大感受野

增加网络平移不变性

减少参数量， 有利于网络收敛速度。

（2）卷积的本质是什么， 卷积怎么提取特征， 为什么能提取特征

在信号系统中， 卷积的本质是一种积分运算， 它可以用来描述线性时不变系统的输入和输出的关系， 即 t 时刻的输出应该为 t 时刻之前系统响应函数在各个时刻响应的叠加；

深度学习系统中， 卷积运算指的是卷积核(滤波器)和在特征图上对应元素相乘求和， 并已 stride 大小滑动遍历整个特征图。

神经网络由卷积层、池化层、激活层构成； 传统的特征提取是人工根据经验设置好卷积模板， 比如 Hog 特征、直方图特征、Sobel 边缘提取特征； 然后和图像进行卷积进而提取特征； 深度学习阶段， 卷积核起到的是模板的作用， 只不过模板的参数是网络根据反向传播学习到的， 网络训练好以后， 卷积核就具有了特征提取的作用； 传统的卷积模板提取的特征相当于是白盒， 可以知道是什么特征； 而 CNN 提取的特征相当于是黑盒， 特征是网络自动学习出来的， 并不能清楚的描述出来。

概率：

①求 std(标准差) 为什么除以 (n-1) 而不是 n

解答：

总体标准差： $\sigma=\sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n}}$

样本标准差：$S=\sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}}$

样本方差计算公式里分母为 n － 1， n−1 的目的是为了让方差的估计是无偏的。

标准差其实有两个公式：

一个是针对总体而言的,公式中是除以 n.

另一个是针对样本而言的, 公式中是除以 n-1， 称为样本标准差， 分母除以 n-1 是由于(xi−x) 的自由度为 n-1,即(xi−x) 中只需确定 n － 1 个数值,另外一个数值也被确定.

需要指出的是， 实际生活中的数据基本都是样本,一般我们处理数据用的都是样本标准差.

6.2 手撕代码-编辑距离

（1）编辑距离 给你两个单词 word1 和 word2， 请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符

删除一个字符

替换一个字符

示例：

输入： word1 = "horse", word2 = "ros"

输出： 3

解释：

horse -> rorse (将 'h' 替换为 'r')

rorse -> rose (删除 'r')

rose -> ros (删除 'e')

解题思路：

编辑距离样例网址 ： https：//alchemist-al.com/algorithms/edit-distance

		G	C	C	G	T
	0	1	2	3	4	5
T	1	1	2	3	4	4
A	2	2	2	3	4	5
G	3	2	3	3	3	4
G	4	3	3	4	3	4
A	5	4	4	4	4	4

图中数字表示序列“TAGGA”依次更改为“GCCCGT”对应的编辑距离。

：表示"T" => "GC"的编辑距离为 2 ，即用"G"替换"T" 再插入"C";

：表示"T" => "GCC"的编辑距离为 3 ，即用"G"替换"T" 再 2 次插入"C";

： 表示"TA" => "GC"的编辑距离为 2 ，即用"G"替换"T" 再用"C"替换"A";

由上述内容推理： S1 ="TA" => S2 ="GCC"： 分为如下两种情况：

如果 S1[-1] == S2[-1]，那么问题可以转换为 S1[： -1] => S2[-1], 即计算除最后一位之外的编辑距离， 也就是左上角位置对应的编辑距离。

如果 S1[-1] != S2[-1]，即图中的情况 "A" != "C"，此时对应的编辑距离为： （ ，， ）最小值 然后加 1 ，即为 S1 => S2 的编辑距离；

如果最小值为表示： 在 S1[： -1] => S2[： -1] 的基础上再将 S1[-1]替换为 S2[-1]；

如果最小值为表示： 在 S1[： -1] => S2 的基础上再删除 S1[-1]；

如果最小值为表示： 在 S1 => S2[： -1] 的基础上再插入 S2[-1]；

第一行数字和第一列数字表示将空字符串变成相应字符所需要的编辑距离；

上述解题思路对应的代码如下：

```
1.  class Solution:
2.      def minDistance(self, word1:  str, word2:  str) -> int:
3.          n1 = len(word1)
4.          n2 = len(word2)
5.          # 存储当前元素的编辑距离
6.          dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
7.          # 第一行
8.          for j in range(1, n2 + 1):
9.              dp[0][j] = dp[0][j-1] + 1
10.         # 第一列
11.         for i in range(1, n1 + 1):
12.             dp[i][0] = dp[i-1][0] + 1
13.
14.         for i in range(1, n1 + 1):
15.             for j in range(1, n2 + 1):
16.                 # 相等时 copy 左上角的值
17.                 if word1[i-1] == word2[j-1]:
18.                     dp[i][j] = dp[i-1][j-1]
19.                 # 不相等时：mmin(上，左、左上角) + 1
20.                 else:
21.                     dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-
22.                      1] ) + 1
23.         #print(dp)
24.         return dp[-1][-1]
```

（2）求出每种编辑操作的次数分别是多少次

解题思路： 不同操作的时候记录下来， 而后分别记录每种操作的次数；

替换操作：

dp[i][j] =dp[i−1][j−1] +1

插入操作：

dp[i][j] =dp[i][j−1] +1

删除操作：

dp[i][j] =dp[i-1][j]+1

只需要在上题代码中加入操作的计数代码即可：

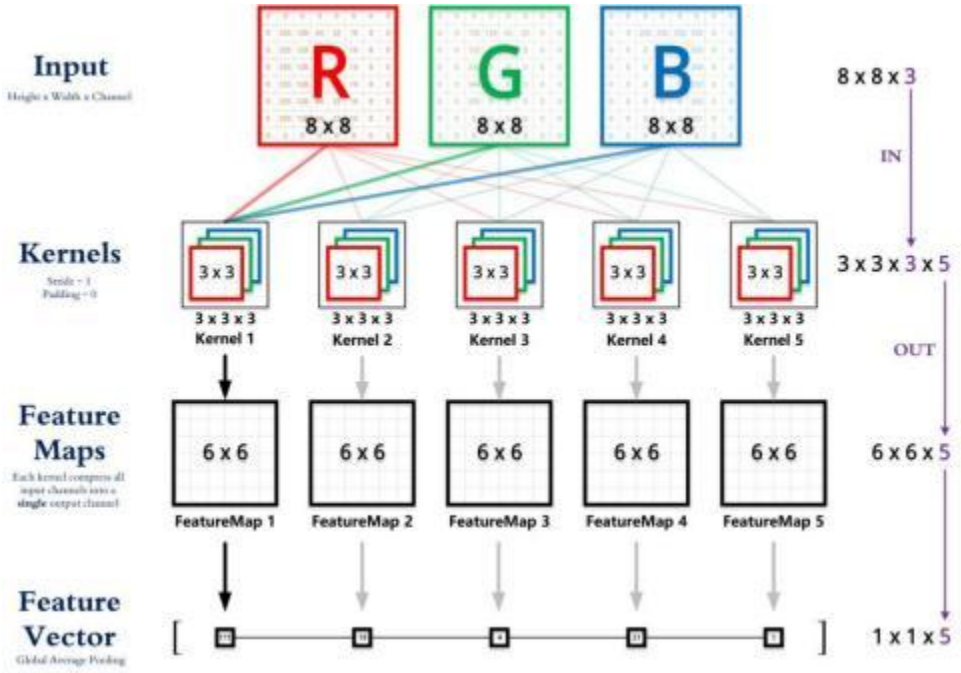
```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        n1 = len(word1)
        n2 = len(word2)
        # 存储当前元素的编辑距离
        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        # 第一行
        for j in range(1, n2 + 1):
            dp[0][j] = dp[0][j-1] + 1
        # 第一列
        for i in range(1, n1 + 1):
            dp[i][0] = dp[i-1][0] + 1
        del_num, repalce_num, inter_num = 0, 0, 0
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                # 相等时 copy 左上角的值
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                # 不相等时：mmin(上，左、左上角) + 1
                else:
                    dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-
                     1] ) + 1
                     if dp[i][j] == dp[i][j-1]:  repalce_num += 1
                     elif dp[i][j] == dp[i-1][j]: del_num += 1
                     elif dp[i][j] == dp[i-1][j-1]: inter_num += 1
        return repalce_num，del_num，inter_num
```

第七篇 2024 年春节后商汤面试题 7 道

视频理解研究员， 校招。 四面技术面。已 offer， 4 月入职。商汤一直以来都是自己最仰慕的公司，从 2017 年第一次进实验室， 我做的所有 项目 的所有结果， 都要差于商汤。拿了商汤 offer 后， 我立马拒绝了各种 offer， 我真的太爱商汤了。

7.1 普通卷积计算量，深度可分离卷积计算量

普通卷积计算：



(图像来源 ： https：//zhuanlan.zhihu.com/p/29119239)

定义 ： 输入图像大小 ： Hn*win*c in ； 输出图像大小 ： H,*W。*C,

卷积核大小 ： k*k*DK*c,其中 ： DK为卷积核的厚度等于cin,C_o为卷积核的个数, 也是输出特征图的通道数。

标准卷积计算过程如上图所示， 其中

输出图像大小计算公式为 ：

$$H_o = W_o = [(W - F + 2 * P)/S] + 1 = [8 - 3 + 2 * 0]/1] + 1 = 6$$

参数量(不考虑偏置项) ：

$$Para = k * k * C_{in} * C_o$$

参数量即可训练的参数数量， 卷积计算过程中参数共享， 而且不同通道的卷积核参数也不一样， 所以每个卷积核的参数量为k米k米c， 共有co个卷积核。

计算量：

$$Cal = [k*k*C_{in} + (k*k-1)*C_{in} + (C_{in}-1)]*H_o*W_o*C_o = (2*k*k*C_{in}-1)*H_o*W_o*C_o$$

计算量包括加减乘除操作次数， 其中 1 个卷积核 1 次卷积的计算量为：

k*k*cin十(k*k一1)*cn十(cin一1)

第 1 部分是卷积核在各通道进行 1 次乘法操作，

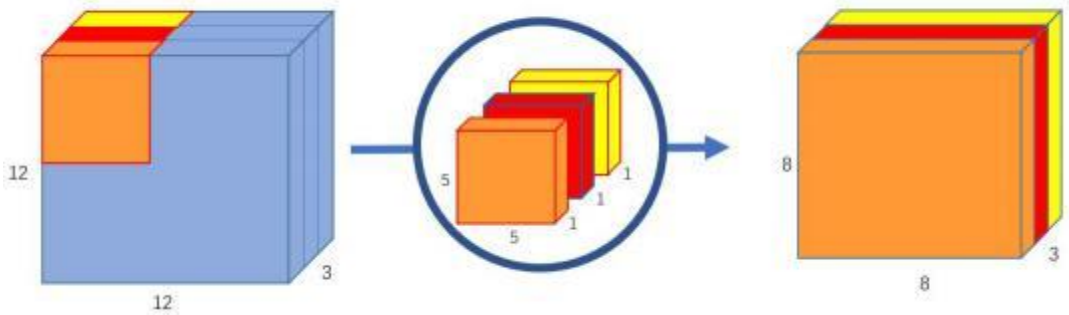
第 2 部分是乘法之后卷积核内部的加合操作，

第 3 部分是各通道间的加合操作； 一个卷积核在一个输出通道上共进行H,*WO次卷积操作， 共有C_o各卷积核。

深度可分离卷积：

深度可分离卷积包括： 深度卷积（deep wise ）和逐点卷积（ point wise ）

深度卷积（ Deep Wise ）



$$H_{in} = W_{in} = 12, C_{in} = 3; k = 5, D_k = 1; H_o = W_o = 8, C_o = C_{in} = 3$$

与标准卷积网络不一样的是， 深度卷积将卷积核拆分成单通道形式， 在不改变输入特征图通道数的情况下， 对每一通道进行卷积操作， 这样就得到了和输入特征图通道数一致的输出特征图。如上图， 输入 12*12*3 的特征图， 经过 5*5*1*3 的深度卷积之后， 得到了 8*8*3 的输出特征图。输入和输出的维度都是 3。

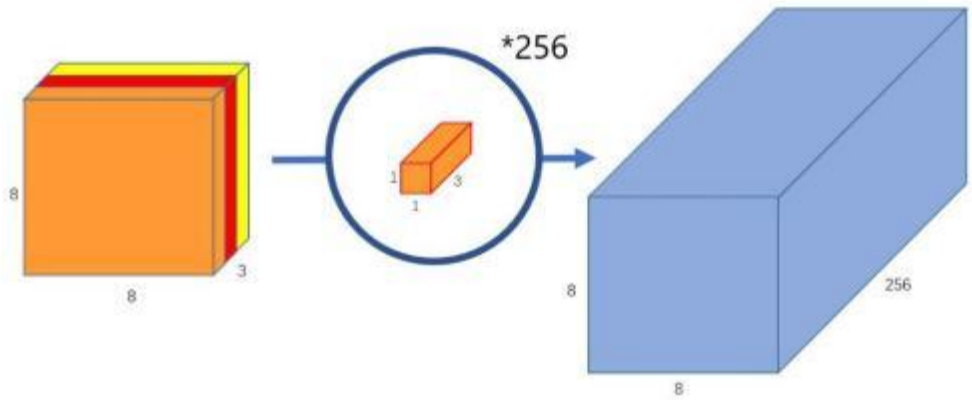
参数量：

$$P_{dw} = k*k*D_k*C_{in} = k*k*C_{in}$$

计算量：

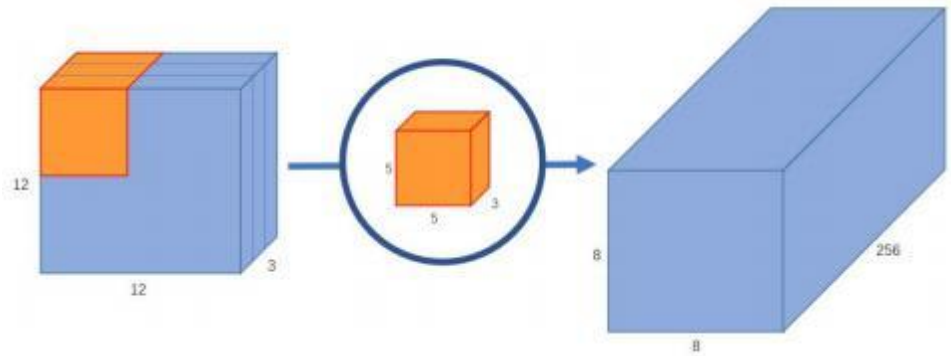
calaw= [k*k*DK+(k*k-1)]*cin*H,*w,= (2*k*k-1)*cin*H,*w

逐点卷积（Point Wise）



$$H_{in} = W_{in} = H_o = W_o = 8, C_{in} = 3; k = 1, D_k = 3; C_o = 256$$

使用 256 个 1*1*3 的卷积核对 DepthWise 输出特征图进行卷积操作，输出的特征图和标准的卷积操作一样都是 8*8*256 了，DepthWise+PointWise 对应的标标准卷积计算为：



逐点卷积其实和标注卷积是一样的，只不过卷积核大小为1*1，因此：

参数量：

Pw=k*k*c in*c,=c*c,

计算量：

caly w=[k*k*cn十(k*k一1)*c in十(c in一1)*H,*w*c,=(2*c in一1)*H,*w,*c,

综上 深度可分离卷积总的参数量和计算量分别为：

para=pa+RW=口 Dk cin=的cn+cm c,

cal=ca law+cal p= I(2k"-I)c in+(2cn-1)]H,w,

标准卷积和深度可分析卷积参数量对比：

$$\frac{Para_{DSC}}{Para_{std}} = \frac{k^2 C_{in} + C_{in} C_o}{k^2 C_{in} C_o} = \frac{1}{C_0} + \frac{1}{k^2} \approx \frac{1}{k^2}$$

同样的卷积效果， 标准卷积参数量是深度可分离卷积参数量的k^2倍； 卷积核一般为 3*3 大小， 参数量下降到原来的$\frac{1}{9}$到$\frac{1}{8}$。

7.2 Sigmoid 、MSELoss 及 CELoss 的来源

Sigmoid 是 Softmax 函数在二分类时的特殊情况， 而 Softmax 是根据最大熵原理推导出来的。

MSELoss： 根据误差服从正态分布， 用似然函数用于参数估计， 如何可推导处均方误差损失。

https：//blog.csdn.net/weixin_42065178/article/details/106312229

CELoss： 交叉熵损失函数是根据 KL 散度推导出来的。

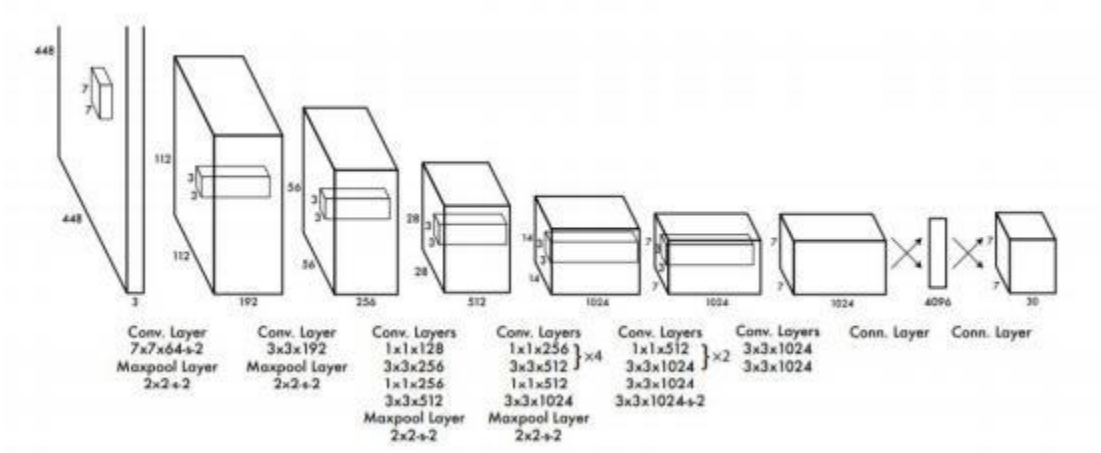
7.3 讲一下目标检测 OneStage、 TwoStage 以及 YOLOv1

two stage 代表算法： Fast RCNN、 Faster RCNN、Mask RCNN 等； 优点是模型精度高， 缺点是速度慢， 因为网络需要先经过一个区域生成网络， 生产待检测目标候选区域， 再进行目标分离和边框回归任务；

one stage 代表算法： YOLOv1-v5、 SSD、 RetinaNet 等； 优点是模型速度快， 缺点是精度比 TwoStage 算法略低； 原始是 OneStage 算法不需要经过区域生成网络， 直接进行目标分离和回归任务， 一步完成， 所以速度快；

YOLOv1 算法简介：

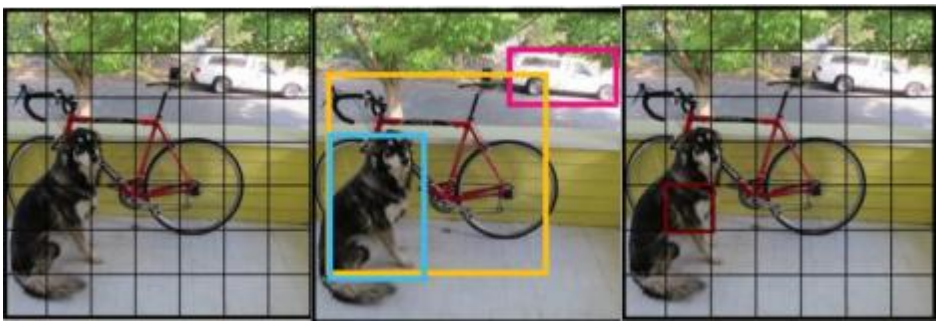
Yolov1 的核心思想就是利用整张图作为网络的输入， 然后输出层回归 bounding box 位置和目标所属的类别。



原理分析：

网络结构图如上所示， 模型主干网络是 GoogleNet， 如何又接了 7 层卷积网络进行处理， 最后输出特征图的维度为（ 7*7*30）：每个 cell 预测 2 个 Box， 每个 Box 有(x,y,w,h,score) 5 个分数， 再加上 VOC 的 20 个类别， 所以最后是 7*7*(2*5+20)大小的矩阵；

每个像素点映射回原图可得到下左图所示：



每一个像素点映射回原图是一个矩形区域，相当于把图像分成了 7*7 大小的网格 cell，每个网格 cell 都可以输出物体的类别和 bounding box 坐标，如图上中图所示。

将输入图像按照模型的输出网格（比如 7*7）进行划分，预测目标中心落在哪个 cell 里面，那个 cell 就负责预测这个目标。上图右，狗的中心落在了红色的 cell 内，这个 cell 负责预测狗。这个过程分为两个阶段：

（1）训练阶段：在训练阶段，如果目标中心落在这个 cell，那么就给这 cell 打上这个目标的 label（包括 xywh 和类别）。通过这种方法来设置训练 label，然后训练 cell 预测待检测目标。

（2）测试阶段：在训练阶段已经训练了 cell 去预测中心落在 cell 的目标，cell 在测试中也是这么做的。

置信度预测：

公式：score=pr(object)*I O U

每个 cell 预测 2 个边框，每个边框有(x, y, w, h, score, class)信息，置信度 score 包括两部分：P,(object)表示边框内含有待检测目标的概率，IOU_{pred}^{truth} 表示预测框准确度；

训练阶段：

如果待检测目标中心没有落在 cell 中，则该 cell 对应的边框p,(object)=0，此时不需要考虑 IOU_{pred}^{truth}；如果 cell 是待检测目标中心，则p,(object f)=1，此时$score = \text{IOU}_{\text{pred}}^{\text{truth}}$，$IOU_{pred}^{truth}$ 是边框真实值和预测值之间的重叠比，训练中随预测值的改变而改变。

预测阶段：

预测阶段的置信度 score 值是网络直接输出的，因为上面已经训练好了让网络预测每个 cell 对应的边框置信度大小；

类别预测：

在 YOLO 中，每个格子只有一个 Class 类别，即相当于忽略了 B 个 bounding boxes，每个 cell 只判断一次类别，这是 YOLOv1 的缺点。

对于 VOC 数据集，网络对每个边框预测 20 个 class 类别概率，判定边框的类别。

目标类别其实是一个条件概率：pr(class f object)：即边框内有待检测目标的条件下，目标属于某一类别的概率；

训练阶段：

如果 cell 是待检测目标(类别 A)的中心，则该 cell 对应边框的类别 A 概率为 1，其余为 0，即 $\Pr(\text{Class}_{i==A}/\text{Object})=1,\Pr(\text{Class}_{i!=A}/\text{Object})=0$

测试阶段：

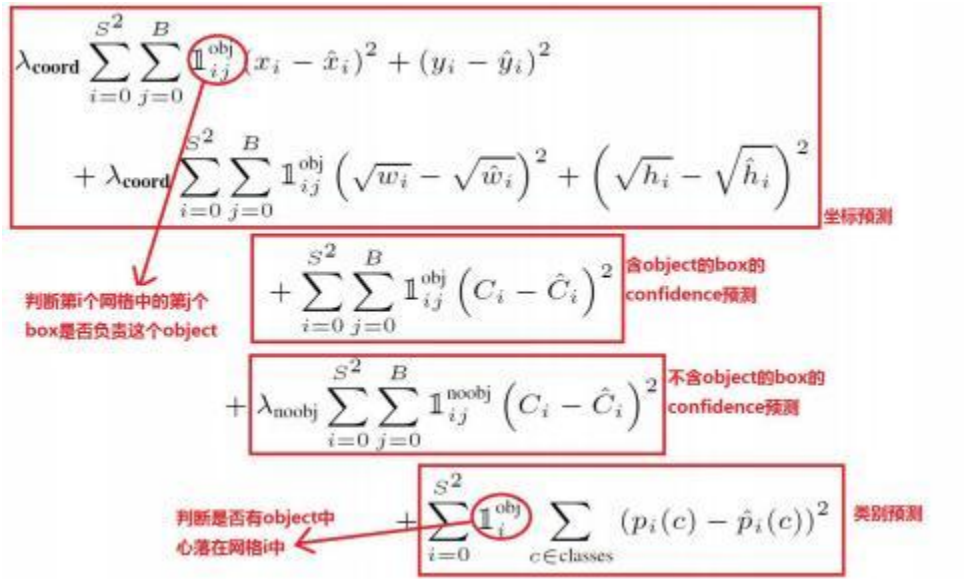
网络直接输出概率值pr(class f object)；代表有目标存在的条件下类别概率是多

少，测试阶段还把这个概率乘上了置信度。计算公式如下：

$$\Pr(\text{Class}_i \mid \text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

即表答了属于某个目标类别的概率又表达了目标边框的准确度两层含义。

损失函数：



YOLOv1 的损失函数设计时主要考虑了以下 3 个方面

(1) bounding box(x, y, w, h)的坐标预测损失：考虑到有大小不同的边框，所以对 w 和 h 求平方根进行回归。

(2) bounding box 的置信度损失：由于绝大部分网格中不包含待检测目标，大部分 box 的置信度为 0，所以在设计置信度误差时对于在不含 object 的边框置信度增加惩罚系数λnoobj=0.5。另外为了增加边框坐标损失的权重，设置权重系数λcoord=5。

(3) 分类损失：即每个 box 属于什么类别，采用的均方误差损失函数；需要注意一个 cell 只预测一次类别，即默认每个 cell 中的所有 B=2 个 bounding box 都是同一类。

关于损失函数：

有目标中心落入的 cell 需要计算分类损失，两个预测边框都需要计算置信度损失，预测的两个 bounding box 分别与 ground truth 计算 IoU 值，值较大的那个 bbox 参与计算边框回归损失；

没有目标中心落入的 cell 只需要计算置信度损失。

公式中每一项 loss 的计算都是 L2 loss(分离损失函数)，所以 YOLOv1 是把分类问题转化为了回归问题。

7.4 写一下 Softmax CrossEntropy 的反向传播推导过程

损失函数的反向传播过程即对损失函数的求导。

在目标分类的任务中，交叉熵损失函数是最长用的一个损失，下面我就推导一下他的反向传播公式，假设网络输出的 n 维特征向量为：=[1,2,xg,…, zl，然后经过 SoftMax 函数得到一个可以表示概率形式的向量，再用交叉熵函数计算 loss。

根据Softmax公式：$softmax(\mathbf{z}_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$ 其中$j = 1, \ldots, K$ 得到概率化的特征向量：

$$\text{softmax}(x) = \left[\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x2}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_3}}{\sum_{i=1}^{n} e^{x_i}}, \cdots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{xi}} \right]$$

简写为：

$$softmax(x) = [a_1, a_2, a_3, \cdots, a_n] 其中 a_k = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{xi}}; k \in [1, n]$$

交叉熵损失函数为：

$$\text{Loss} = -\sum_{i=1}^{n} y_i \log a_i$$

其中 y_i 是真实标签对应的onehot向量；

为求交叉熵的反向传播，先对Softmax公式进行求导，加入我们对 x_k 求导，公式如下：

$$\frac{\partial softmax(x)}{\partial x_k} = \left[\frac{\partial a_1}{\partial x_k}, \frac{\partial a_2}{\partial x_k}, \ldots, \frac{\partial a_k}{\partial x_k}, \cdots, \frac{\partial a_n}{\partial x_k} \right]$$

对于 $\frac{\partial a_k}{\partial x_k}$ 项求导计算如下：

$$\frac{\partial a_k}{\partial x_k} = \frac{\partial \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}}{\partial x_k} = \frac{e^{x_k}\left(\sum_{i=1}^{n} e^{x_i}\right) - e^{x_k}\left(e^{x_k}\right)}{\left(\sum_{i=1}^{n} e^{x_i}\right)^2} = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} \cdot \left(1 - \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}\right) = a_k \cdot (1 - a_k)$$

对于 $a_{j \neq k}$ 项求导计算如下：

$$\frac{\partial a_j}{\partial x_k} = \frac{\partial \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}}{\partial x_k} = \frac{-e^{x_j}\left(e^{x_k}\right)}{\left(\sum_{i=1}^{n} e^{x_i}\right)^2} = -\frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \cdot \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} = -a_j \cdot a_k$$

又因为 $\frac{\partial L}{\partial a_i} = -y_i \frac{1}{a_i}$,带入上式可得：

$$\begin{aligned}\frac{\partial L}{\partial x_k} &= a_k\{\sum_{i=1}^{n} \frac{\partial L}{\partial a_i} \cdot (-a_i) + \frac{\partial L}{\partial a_k}\} \\ &= a_k\{\sum_{i=1}^{n} -y_i \frac{1}{a_i} \cdot (-a_i) + (-y_k)\frac{1}{a_k}\} \\ &= a_k\{\sum_{i=1}^{n} y_i - y_k \frac{1}{a_k}\} \\ &= a_k(\sum_{i=1}^{n} y_i) - y_k\end{aligned}$$

又因为标签y是一个onehot变量，所以 y_i 中只有1个1，其余为0，所以 $\sum_{i=1}^{n} y_i = 0$

因此上式可继续化简为：

$$\frac{\partial L}{\partial x_k} = a_k(\sum_{i=1}^{n} y_i) - y_k = a_k - y_k = \frac{e^{x_1}}{\sum_{i=k}^{n} e^{x_i}} - y_k$$

综上交叉熵损失函数的反向传播(求导)为：

$$\frac{\partial L}{\partial x_k} = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}} - y_k$$

对交叉熵损失进行求导：

$$\begin{aligned}\frac{\partial L}{\partial x_k} &= \sum_{i=1}^{n} \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial x_k} \\ &= \sum_{i=1,i\neq k}^{n} \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial x_k} + \frac{\partial L}{\partial a_k} \cdot \frac{\partial a_k}{\partial x_k} \\ &= \sum_{i=1,i\neq k}^{n} \frac{\partial L}{\partial a_i} \cdot (-a_i \cdot a_k) + \frac{\partial L}{\partial a_k} \cdot (a_k \cdot (1 - a_k)) \\ &= a_k \left\{ \sum_{i=1,i\neq k}^{n} \frac{\partial L}{\partial a_i} \cdot (-a_i) + \frac{\partial L}{\partial a_k} \cdot (1 - a_k) \right\} \\ &= a_k \left\{ \sum_{i=1,i\neq k}^{n} \frac{\partial L}{\partial a_i} \cdot (-a_i) + \frac{\partial L}{\partial a_k} \cdot (-a_k) + \frac{\partial L}{\partial a_k} \right\} \\ &= a_k \left\{ \sum_{i=1}^{n} \frac{\partial L}{\partial a_i} \cdot (-a_i) + \frac{\partial L}{\partial a_k} \right\}; \text{这一步已经包括} i = k \text{了}\end{aligned}$$

7.5 lc5 最长子串

方法： 使用中心扩展法

遍历每一个字符串， 分两种情况进行中心扩展， 一种为奇数， 一种为偶数， 如果两边字母相同， 就继续扩展； 如果不相同就停止扩展， 对所有满足要求的长度求出最大值， 得到最终答案。

代码如下：

```
class Solution:
    def expandAroundCenter(self, s, left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    def longestPalindrome(self, s: str) -> str:
        start, end = 0, 0
        for i in range(len(s)):
            left1, right1 = self.expandAroundCenter(s, i, i)
            left2, right2 = self.expandAroundCenter(s, i, i + 1)
            if right1 - left1 > end - start:
                start, end = left1, right1
            if right2 - left2 > end - start:
                start, end = left2, right2
        return s[start: end + 1]
```

7.6 lc206 反转链表 （递归、递推）

本题是一道基本的链表滚动赋值， 通过设置 pre 和 cur 和临时 tmp 三个指针， 进行指针交替滚动，完成翻转。

代码如下：

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        cur,pre = head,None
        while cur:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        return pre
```

7.7 lc215 第 k 个最大元素

解题思路： 堆排序

总体思路： 维护一个大小为 k 的最小堆， 堆顶是这 k 个数里的最小的， 遍历完数组后返回堆顶元素

即可。

其时间复杂度为 O(nlogk)，因此堆排序这种解法在总体数据规模 n 较大，而维护规模 k 较小时对时间复杂度的优化会比较明显。

代码如下（Python3 可以用 heapq 模块实现堆，Java 可以用优先队列实现）

```
1.  class Solution:
2.      def findKthLargest(self, nums:  List[int], k:  int) -> int:
3.          heap = []
4.          for num in nums:
5.              heapq.heappush(heap, num)
6.              if len(heap) > k:
7.                  heapq.heappop(heap)
8.          return heap[0]
```

第八篇 2024 年 4 月 26 日-4 月 30 日腾讯 NLP 算法实习面试题 14 道

8.1 介绍下 word2vec

介绍下 word2vec， 有哪两种实现， 可以用什么方法提高性能， 分层 softmax 介绍一下原理， 负采样怎么做，， 负采样和原始做法的优缺点比较。

参考答案：

在进行最优化的求解过程中， 从隐藏层到输出的 softmax 层放入计算量很大， 因为要计算所有词的 softmax 概率， 再去找概率最大的值， 可以使用层次 softmax 和负采样两种方法；

层次 softmax 采用对输出层进行优化的策略， 输出层从原始模型利用 softmax 计算概率值改为利用 huffman 树进行计算概率值， 其优点： 由于是二叉树， 之前计算量为 V， 现在变为了 logV； 由于使用 Huffman 树是高频的词靠近树根， 这样高频词需要更少的时间被找到， 符合贪心优化思想； 其缺点： 如果训练样本中的中心词 w 是一个很生僻的词， 那么就需要在 Huffman 树中向下走很久。

负采样过程： 比如我们有一个训练样本， 中心词是 w,它周围上下文共有 2c 个词， 记为 context(w)。由于这个中心词 w 和 context(w)相关存在， 因此它是一个真实的正例。通过 Negative Sampling 采样，我们得到 neg 个和 w 不同的中心词 wi,i=1,2,...neg， 这样 context(w)和 wi 就组成了 neg 个并不真实存在的负例。利用这一个正例和 neg 个负例， 我们进行二元逻辑回归， 得到负采样对应每个词 wi 对应的模型参数θi， 和每个词的词向量。

负采样相比原始做法： 每次只需要更新采样的词的权重， 不需要更新所有词的权重， 可以减少了训练过程的计算负担， 加速模型的计算， 另一方面也可以保证模型训练的效果， 提高了其结果词向量的质量。

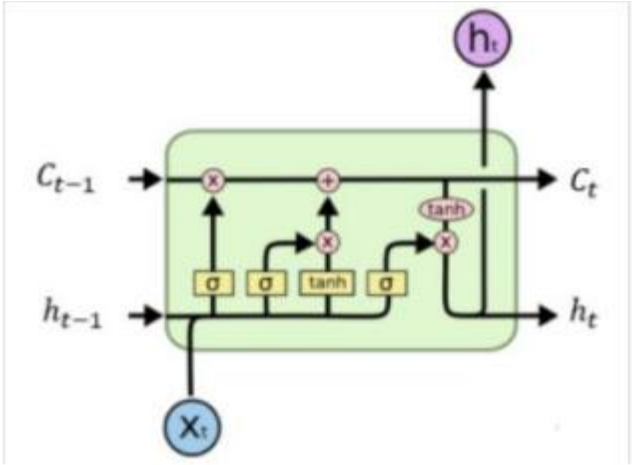
8.2 介绍 LSTM 和 GRU 有什么不同

（1）LSTM 和 GRU 的性能在很多任务上不分伯仲；

（2）GRU 参数更少， 因此更容易收敛， 但是在大数据集的情况下， LSTM 性能表现更好；

（3）从结构上说， GRU 只有两个门， LSTM 有三个门， GRU 直接将 hidden state 传给下一个单元， 而 LSTM 则用 memory cell 把 hidden state 包装起来。

8.3 介绍一下 LSTM 单元里面的运算过程



忘记门：$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$

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

加入到细胞状态的信息：$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

细胞状态更新：$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

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

隐藏状态：$h_t = o_t * \tanh(C_t)$

8.4 CRF 的损失函数是什么，具体怎么算？

在训练过程中， CRF 损失函数只需要两个分数： 真实路径的分数和所有可能路径的总分数。**所有可能路径的分数中， 真实路径分数所占的比例会逐渐增加。**

损失函数：

$$LossFunction = \frac{P_{RealPath}}{P_1 + P_2 + ... + P_N}$$

实际路径得分：

Ra a=R+R+…+p=esi+e□ +…+e

S_i由两部分组成$S_i = EmissionScore + TransitionScore$

8.5 transformer 介绍一下原理

transformer 介绍一下原理，transformer 为什么可以并行，它的计算代价瓶颈在哪？多头注意力机制计算过程？

Transformer 结构（原理）：

Transformer 本身还是一个典型的 encoder-decoder 模型，Encoder 端和 Decoder 端均有 6 个 Block，Encoder 端的 Block 包括两个模块，Decoder 端的 Block 包括三个模块，需要注意：Encoder 端和 Decoder 端中的每个模块都有残差层和 Layer Normalization 层。

Encoder 端的 Block：多头 self-attention 模块以及一个前馈神经网络模块，

多头 self-attention 模块

query、key、value 的纬度均为 64，输出为 values 的加权和，其中分配给每个 value 的权重由 query 与对应 key 的相似性函数计算得来。

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

前馈神经网络模块

由两个线性变换组成，中间有一个 ReLU 激活函数。输入和输出的纬度均为 64，内层维度为 2048。

$$\text{FFN}(x)=\max\left(0,xW_1+b_1\right)W_2+b_2$$

Decoder 端的 Block：多头 self-attention 模块，多头 Encoder-Decoder attention 交互模块，以及一个前馈神经网络模块；

多头 self-attention 模块

与 Encoder 基本一致，但注意需要做 mask。

多头 Encoder-Decoder attention 交互模块

Q 来源于 Decoder 自身子模块的输出，K，V 来源于整个 Encoder 端的输出。

前馈神经网络模块

该部分与 Encoder 端的一致。

Transformer 为什么可以并行？

Encoder 支持并行化：

自注意力机制就是利用 xi 之间两两相关性作为权重的一种加权平均将每一个 xi 映射到 zi，在计算每一个 zi 时只需要 xi，并不依赖 zi-1。

Decoder 支持部分并行化的原因：

teacher force：是指在每一轮预测时，不使用上一轮预测的输出，而强制使用正确的单词。

masked self attention

Decoder 的并行化仅在训练阶段， 在测试阶段， 因为我们没有正确的目标语句， t 时刻的输入必然依赖 t-1 时刻的输出， 这时跟之前的 seq2seq 就没什么区别了。

transformer 的计算瓶颈：

在传统的 Transformer 中， 每个 block 中都有 Multi-head Attention 和全连接层， 随着序列长度 N 的增加， 全连接层的计算量是线性增长， 而 attention 的计算量则是平方增长， 因此， 当序列长度比较长的时候， attention 就会有很大的计算量。

多头注意力机制计算过程？

多次 attention 综合的结果至少能够起到增强模型的作用， 也可以类比 CNN 中同时使用多个卷积核的作用， 直观上讲， 多头的注意力有助于网络捕捉到更丰富的特征/信息。

多头 self-attention 模块， 则是将 Q,K,V 通过参数矩阵映射后(给 Q,K,V 分别接一个全连接层)，然后再做 self-attention， 将这个过程重复 6 次， 最后再将所有的结果拼接起来， 再送入一个全连接层即可。

8.6 BERT 介绍一下原理

BERT 介绍一下原理，怎么用 BERT 计算文本相似度，有哪两种计算方法？

参考答案：

BERT 原理：

BERT 句向量： 取最后一两层 embedding 的平均或直接用[CLS]对应的 embedding， 使用 BERT 计算文本相似度的效果是不好的。

原因：

一： BERT 句向量的空间分布是不均匀的， 受到词频的影响。因为词向量在训练的过程中起到连接上下文的作用， 词向量分布受到了词频的影响， 导致了上下文句向量中包含的语义信息也受到了破坏。

二： BERT 句向量空间是各向异性的， 高频词分布较密集且整体上更靠近原点， 低频词分布较稀疏且整体分布离原点相对较远。因为高词频的词和低频词的空间分布特性， 导致了相似度计算时， 相似度过高或过低的问题。在句子级： 如果两个句子都是由高频词组成， 那么它们存在共现词时， 相似度可能会很高，而如果都是由低频词组成时， 得到的相似度则可能会相对较低； 在单词级： 假设两个词在语义上是等价的， 但是它们的词频差异导致了它们空间上的距离偏差， 这时词向量的距离就不能很好的表征语义相关度。

解决办法：

BERT-Flow： 利用标准化流(Normalizing Flows)将 BERT 句向量分布变换为一个光滑的、各向同性的标准高斯分布。

Sentence-Bert： 在微调的时候直接把损失函数改写为余弦相似度， 然后采用平方损失函数。

8.7 transformer 里自注意力机制的计算过程

transformer 里自注意力机制的计算过程,为什么要进行缩放,为什么要用多头?

参考答案：

自注意力机制的计算过程：

query、key、value 的纬度均为 64，输出为 values 的加权和，其中分配给每个 value 的权重由 query 与对应 key 的相似性函数计算得来。

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

进行缩放的原因：

除以缩放因子，不让较大的值经过 softmax 后变得非常大

避免出现梯度消失

使用多头的原因：

多次 attention 综合的结果至少能够起到增强模型的作用，也可以类比 CNN 中同时使用多个卷积核的作用，直观上讲，多头的注意力有助于网络捕捉到更丰富的特征/信息。

8.8 介绍下 bert 位置编码和 transformer 的区别

参考答案：

Transformer 解决并行计算问题的法宝，就是 Positional Encoding，简单点理解就是，对于一句文本，每一个词语都有上下文关系，而 RNN 类网络由于其迭代式结构，天然可以表达词语的上下文关系，但 transformer 模型没有循环神经网络的迭代结构，所以我们必须提供每个字的位置信息给 transformer，才能识别出语言中的顺序关系，为了解决这个问题，谷歌 Transformer 的作者提出了 position encoding。

原版的 Transformer 中，谷歌对 learned position embedding 和 sinusoidal position encoding 进行了对比实验，结果非常相近。而 Sinusoidal encoding 更简单、更高效、并可以扩展到比训练样本更长的序列上，因此成为了 Transformer 的默认实现。

对于机器翻译任务，encoder 的核心是提取完整句子的语义信息，它并不关注某个词的具体位置是什么，只需要将每个位置区分开（三角函数对相对位置有帮助）；而 Bert 模型对于序列标注类的下游任务，是要给出每个位置的预测结果的。

8.9 sigmoid 函数的缺点

sigmoid 函数的缺点，为什么会产生梯度消失？不是以 0 为中心的话，为什么会收敛慢？

参考答案：

sigmoid 函数

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。

8.10 Layer Normalization 和 Batch Normalization 的区别

参考答案：

Batch Normalization 的处理对象是对一批样本，Layer Normalization 的处理对象是单个样本。Batch Normalization 是对这批样本的同一维度特征做归一化，Layer Normalization 是对这单个样本的所有维度特征做归一化。

BatchNorm 的缺点：

1.需要较大的 batch 以体现整体数据分布

2.训练阶段需要保存每个 batch 的均值和方差，以求出整体均值和方差在 infrence 阶段使用

3.不适用于可变长序列的训练，如 RNN

Layer Normalization

Layer Normalization 是一个独立于 batch size 的算法，所以无论一个 batch 样本数多少都不会影响参与 LN 计算的数据量，从而解决 BN 的两个问题。LN 的做法是根据样本的特征数做归一化。

LN 不依赖于 batch 的大小和输入 sequence 的深度，因此可以用于 batch-size 为 1 和 RNN 中对边长的输入 sequence 的 normalize 操作。但在大批量的样本训练时，效果没 BN 好。

实践证明，LN 用于 RNN 进行 Normalization 时，取得了比 BN 更好的效果。但用于 CNN 时，效果并不如 BN 明显。

8.11 Leetcode8 字符串转换整数 (atoi)，考虑科学计数法

解析：有三种方法：正常遍历、有限状态机和正则表达式，这里只提供正则表达式的参考答案。

s.lstrip()表示把开头的空格去掉

使用正则表达式：

^：匹配字符串开头

[\+\-]：代表一个+字符或-字符

?：前面一个字符可有可无

\d：一个数字

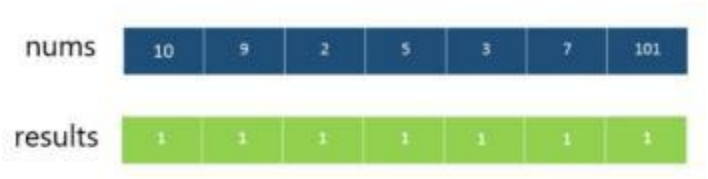
+：前面一个字符的一个或多个

max(min(数字, 2**31 - 1), -2**31) 用来防止结果越界

代码如下：

```
1.  class Solution:
2.      def myAtoi(self, s:  str) -> int:
3.          return max(min(int(*re.findall('^[\+\-]?\d+', s.lstrip())), 2**31 -
    1), -2**31)
```

8.12 最长递增子序列



如上图所示，用 nums 表示原数组，results[i]表示截止到 nums[i] 当前最长递增子序列长度为 results[i]，初始值均为 1；

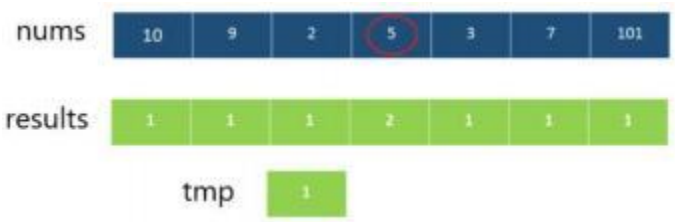
因为要找处递增序列，所以我们只需要找出 nums[i]>nums[j]（其中 j < i）的数，并将对应的 results[j]保存在 tmp 临时列表中，然后找出最长的那个序列将 nums[i]附加其后面；

示例：假设 nums[i] = 5, i = 3

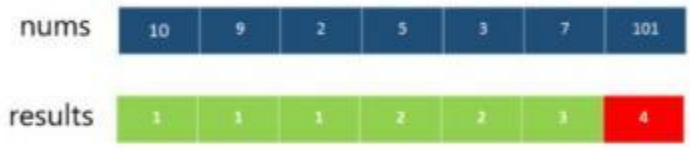
更新 results[i]时只需要考虑前面小于 results[i]的项，所以 results[0-2]均为 1；

此时 nums[i]前面小于 5 的项有 nums[2]，需要将 results[2]保存报 tmp 中，然后选取 tmp 中最大的数值并加 1 后赋值给 results[i], 即 results[5] = 2

tmp 是临时列表更新 results 之后需要清空；



根据上述流程更新所有 results[i]，即可得到下图：



此时我们只需要找出 results 中最大的那个值，即为最长递增子序列的长度。

参考代码如下：

```
1.  class Solution:
2.      def lengthOfLIS(self, nums:  List[int]) -> int:
3.          # 特殊情况判断
4.          if len(nums) == 0:  return 0
5.
6.          results = [1]*len(nums)
7.          for i in range(1, len(nums)):
8.              tmp = []
9.              for j in range(i):
10.                 if nums[i] > nums[j]:
11.                     tmp.append(results[j])
12.             if tmp:  results[i] = max(tmp) + 1
13.         return max(results)
```

8.13 全排列

解析 1：

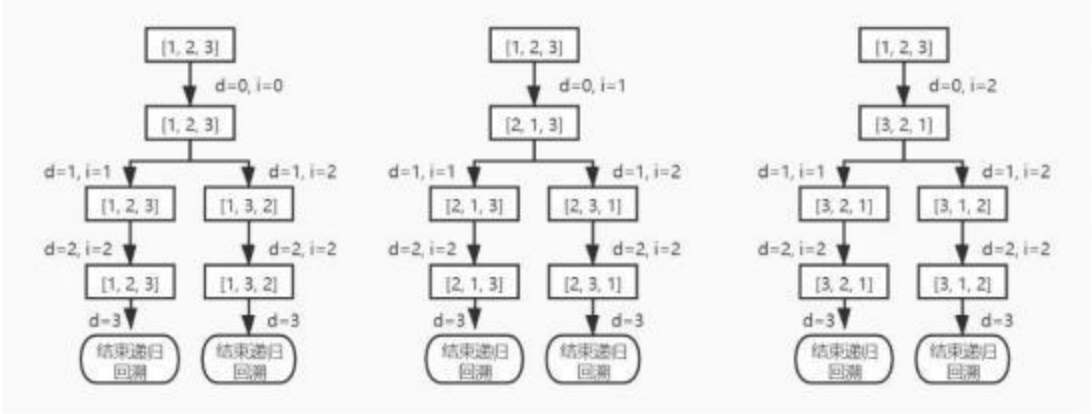
定义一个长度为 len(nums)的空表 output，从左往右一次填入 nums 中的数字，并且每个数只能使用一次。可以枚举所有困难，从左往右每一个位置都一次填入一个数，用 used 表记录 nums[i]是否已填入 output 中，如果 nums[i]木有填入 output 中则填入并标记，当 output 的长度为 len(nums)时，得到一个满足条件的结果，将 output 存入最终结果。然后将 output 回溯到未添加 nums[i]状态，used 回溯为未标记 nums[i]状态，继续下一次尝试。

```
1.  class Solution:
2.      def permute(self, nums:  List[int]) -> List[List[int]]:
3.          # 特殊判断
4.          if len(nums) == 0:  return nums
5.          result = []
6.          depth, n, output, used = 0, len(nums), [], [False]*len(nums)
7.          self.backtrack(depth, n, output, used, nums, result)
8.          return result
9.
10.     def backtrack(self, depth, n, output, used, nums, result):
11.         # 递归终止条件:
12.         if depth == n:
13.             result.append(output[: ])
14.         else:
15.             # 遍历数组中的每一个数
16.             for i in range(0, n):
17.                 # 如果该数字已使用(已在 output 中),  则进入下一个数字
```

```
                if used[i]:  continue
                # 该数字还未出现在 output 中，则添加并标记
                output.append(nums[i])
                used[i] = True
                # 递归进入下一层，选择下一个数字
                self.backtrack(depth+1, n, output, used, nums, result)
                # 回溯操作：去除添加、去除标记
                output.pop()
                used[i] = False
```

解析 2：

在解析 1 中，我们使用了两个辅助空间，output 和 used；为节省空间，我们可以除掉他们；我们可以将给定的 nums 数组分成左右两个部分，用 n 表示数组个数，depth 表示当前需要填充的位置，左边表示已经填好的内容[0~depth)，右边表示待填充的内容[depth~n)，假设当前填充的位置是 i，填充后为了保持上述结构，需要 nums[i]和 nums[depth]交换，在完成一次填充后的回溯过程，还需要再次交换，保持原有内容。整个流程如下图所示：



参考代码：

```
class Solution:
    def permute(self, nums:  List[int]) -> List[List[int]]:
        if len(nums) == 0 :  return nums
        ret = []
        depth, n = 0, len(nums)
        self.backtrack(nums, ret, depth, n)
        return ret
    # 递归
    def backtrack(self, nums, ret, depth, n):
        # 递归终止条件
        if depth == n:
            ret.append(nums[: ])
        else:
            # depth 表示 nums 中第 depth 位置及之后的内容是需要确认的
            for i in range(depth, n):
                nums[depth], nums[i] = nums[i], nums[depth]
                # 递归
                self.backtrack(nums, ret, depth+1, n)
```

```
                # 递归之后回溯得到原始数组
                nums[depth], nums[i] = nums[i], nums[depth]
```

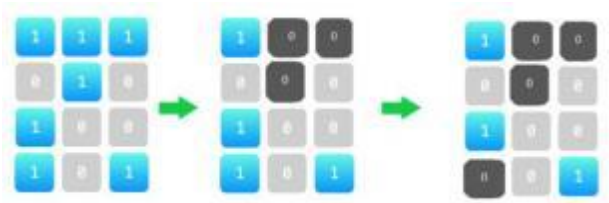
8.14 合并两个有序数组并去重

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        # 方法 1: 直接合并，然后排序
        # nums1[m: ] = nums2
        # nums1.sort()

        # 方法 2: 双指针法
        # 因为两个列表均已是有序，所以可以比较 nums1 和 nums2 的首端元素，将较小者插入到前面
        # p1: nums1 的指针  p2: nums2 的指针
        p1, p2 = 0, 0
        nums1_ = [nums1[i] for i in range(m)]

        i = 0
        while p1 <= m and p2 <= n:
            # 判断 nums1 是否已遍历完
            if p1 == m:
                nums1[m+p2: ] = nums2[p2: ]
                break
            # 判断 nums2 是否已遍历完
            elif p2 == n:
                nums1[n+p1: ] = nums1_[p1: ]
                break

            elif nums1_[p1] <= nums2[p2]:
                nums1[i] = nums1_[p1]
                p1, i = p1+1, i+1
            else:
                nums1[i] = nums2[p2]
                p2, i = p2+1, i+1
```

第九篇 2024 年 5 月 10 日-5 月 16 日滴滴算法工程师面试题 5 道

9.1 岛屿个数（dfs）

题干：给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。此外，你可以假设该网格的四条边均被水包围。



样例：grid = [["1","1","1"], ["0","1","0"], ["1","0","0"], ["1","0","1"]]

解题思路-深度优先：题干要求的是岛屿数量 num, 可以按照行列遍历所有元素，第一次遇到“1”时岛屿数量+1，然后按照深度优先遍历后面连续的“1”，并将“1”赋值为“0”；然后继续遍历，再次遇到“1”时岛屿数量+1，然后再将其链接的“1”改为“0”；重复该过程直至遍历所有元素；参考代码如下：

```
1.  class Solution:
2.      def numIslands(self, grid:  List[List[str]]) -> int:
3.          # 遍历行列元素 如果遇到 1 则岛屿数量加 1，并开始深度优先遍历 1 元素，直至上下左右都是 0
4.          row, column = len(grid), len(grid[0])
5.          numResult = 0
6.          for i in range(row):
7.              for j in range(column):
8.                  if grid[i][j] == "1":
9.                      # 遇到新岛屿，岛屿数量+1
10.                     numResult += 1
11.                     # 使用 dfs 方法将链接的 1 置 0;
12.                     self.dfs(i, j, grid)
13.         return numResult
14.     # 遍历陆地 gride[r, c]的上下左右 判断是否为陆地，是则置零；使用递归的方法链接该岛屿的所有陆地;
15.     def dfs(self, r, c, grid):
16.         row, column = len(grid)-1, len(grid[0])-1
17.         grid[r][c] = 0
18.         if r < row and grid[r+1][c] == "1":  self.dfs(r+1, c, grid)
19.         if c < column and grid[r][c+1] == "1":   self.dfs(r, c+1, grid)
20.         if 0 < r and grid[r-1][c] == "1":   self.dfs(r-1, c, grid)
21.         if 0 < c and grid[r][c-1] == "1":  self.dfs(r, c-1, grid)
```

9.2 向图最短路径（引导我用动态规划解决）

https：//leetcode-cn.com/problems/network-delay-time/

每次找到离源点最近的一个点，以该点为中心，更新源点到其他源点的最短路径，贪心的思想。

参考代码：

```
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -
> int:
        graph = collections.defaultdict(list)
        for u, v, w in times:
            graph[u].append((v, w))
         # 节点 K 到所有节点的距离 没有 W 则赋值正无穷
        dist = {node: float('inf') for node in range(1, n+1)}
        seen = [False] * (n+1)
        dist[k] = 0
        # 循环计算各个点到 K 节点的距离
        while True:
            cand_node = -1
            cand_dist = float('inf')
            # not seen[i]: 遍历未走过的节点
            for i in range(1, n+1):
                # 同时满足两个条件的只有一个节点
                if not seen[i] and dist[i] < cand_dist:
                    cand_dist = dist[i]
                    cand_node = i
            # 遍历所有节点后 cand_note=-1，此时终止 while 循环
            if cand_node < 0: break
            seen[cand_node] = True  # 标记已遍历节点
            # 更新下一节点到 k 节点的最短距离
            for nei, d in graph[cand_node]:
                dist[nei] = min(dist[nei], dist[cand_node] + d)
        # 最短距离中的最大值，即节点 k 到节点 n 的最短距离
        ans = max(dist.values())
        return ans if ans < float('inf') else -1
```

9.3 NP 和 P 问题的看法

解释 1：

最简单的解释：

P： 算起来很快的问题

NP： 算起来不一定快，但对于任何答案我们都可以快速的验证这个答案对不对

NP-hard： 比所有的 NP 问题都难的问题

NP-complete： 是 NP hard 的问题，并且是 NP 问题

解释 2：

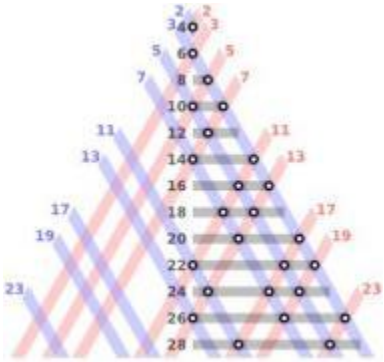
通俗的讲， P 就是能在多项式时间内解决的问题， NP 就是能在多项式时间验证答案正确与否的问题。所以 P 是否等于 NP 实质上就是在问， 如果对于一个问题我能在多项式时间内验证其答案的正确性， 那么我是否能在多项式时间内解决它。

再说说 NP-hardness 和 NP-completenes. 这里涉及一个概念， 不妨称为问题之间的归约。可以认为各个问题的难度是不同的， 表现形式为， 如果我可以把问题 A 中的一个实例转化为问题 B 中的一个实例，然后通过解决问题 B 间接解决问题 A， 那么就认为 B 比 A 更难。通过对归约过程做出限制可以得到不同类型的归约。复杂度理论里经常用到的规约叫 polynomial-time Karp' reduction。其要求是转化问题的过程必须是多项式时间内可计算的。到这为止 NP-hardness 和 NP-completeness 就很好理解了。称问题 L 是 NP-hard， 如果任意一个 NP 的问题都可以多项式规约到 L。如果一个 NP-hard 的问题 L 本身就是 NP 的， 则称 L 是 NP-complete。这个定义可以推广到所有复杂度类。所以 compleness 的直观解释就是， 我能解决这个问题就相当于具备了用相同级别的计算资源解决这个复杂度类里所有问题的能力。

9.4 验证“哥德巴赫猜想”

验证“哥德巴赫猜想”，即： 任意一个大于 2 的偶数均可表示成两个素数之和

数学领域著名的“哥德巴赫猜想”： 任何一个大于 2 的偶数总能表示为两个素数之和；



将一个偶数用两个素数之和表示的方法， 等于同一横线上， 蓝线和红线的交点数。

数值验证：

1938 年， 尼尔斯·皮平（ Nils Pipping ）验证了所有小于 10^5 的偶数。

1964 年， M·L·斯坦恩和 P·R·斯坦恩验证了小于 10^7 的偶数。

1989 年， A·格兰维尔将验证范围扩大到 10^10。

1993 年， Matti K. Sinisalo 验证了 10^11 以内的偶数。

2000 年， Jörg Richstein 验证了 10^14 内的偶数。

截至 2014 年， 数学家已经验证了 10^18 以内的偶数， 在所有的验证中， 没有发现偶数哥德巴赫猜想的反例。

下面**用 Python 进行验证**：

思路： 对于偶数 M， 遍历 0~M， 判断 x ~[0, M]是否是素数， 若是素数则判断 M-x 是否是素数， 若是则找到一个要求的答案：

假设 M＝100

```
import math
def isprime(x):
    if 0 not in [x%i for i in range(2,int(math.sqrt(x))+1)]:
        return True
    else:
        return False
M = 100
for i in range(2, M):
    if isprime(i) and isprime(M-i):
        print('{} = {} + {}'.format(M,i,M-i))
        break
```

9.5 Leetcode 无序数组第 k 大的元素

无序数组第 k 大的元素，要求给出最低时间复杂度的方法以及推导时间复杂度

https://leetcode-cn.com/problems/kth-largest-element-in-an-array/solution/

思路：快排中 partition 思想：

对于某个索引 j，nums[j] 经过 partition 操作：

nums[left] 到 nums[j - 1] 中的所有元素都不大于 nums[j]；

nums[j + 1] 到 nums[right] 中的所有元素都不小于 nums[j]

即 nums[j]元素的位置是排序好了的，我们判断索引 j 是否满足条件，若满足则直接返回结果，若不满足我只需要遍历左序列或者右序列，直至满足要求；无需对整个序列进行排序，所以减少了计算量；

参考代码：

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # 快排中分治思想
        n = len(nums)
        left, right, target = 0, n-1, n - k
        while True:
            pivote = self.getPivote(nums, left, right)
            if pivote == target: return nums[target]
            # 只需遍历 pivote 左边 或者右边 减少计算量
            elif pivote < target: left = pivote + 1
            elif pivote > target: right = pivote - 1

    # 分治思想 pivote 左边都是不大于 pivote 的数，右边是不小于 pivote 的数
    def getPivote(self, nums, left, right):
        pivote = random.randint(left, right)
        nums[left], nums[pivote] = nums[pivote], nums[left]
        pivote = nums[left]
        # 双指针法
        while left < right:
            while nums[right] >= pivote and left < right: right -= 1
            nums[left] = nums[right]
            while nums[left] <= pivote and left < right: left += 1
            nums[right] = nums[left]
        nums[left] = pivote
```

```
25.         return left
```

复杂度分析：

时间复杂度： O(N)，这里 N 是数组的长度

空间复杂度： O(1)，原地排序没有借助额外的辅助空间。

第十篇 2024 年 5 月 10 日-5 月 16 日美团优选 NLP 算法工程师 5 道

10.1 怎么处理数据不平衡

常用于解决数据不平衡的方法：

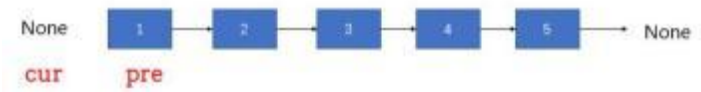
- 欠采样： 从样本较多的类中再抽取， 仅保留这些样本点的一部分；
- 过采样： 复制少数类中的一些点， 以增加其基数；
- 生成合成数据： 从少数类创建新的合成点， 以增加其基数。
- 添加额外特征： 除了重采样外， 我们还可以在数据集中添加一个或多个其他特征， 使数据集更加丰富， 这样我们可能获得更好的准确率结果。

10.2 Leetcode 206-反转链表

给你单链表的头节点 head， 请你反转链表， 并返回反转后的链表。

https：//leetcode-cn.com/problems/reverse-linked-list/

定义两个节点 cur =None 和 pre =head



改变节点方向让 pre 的 next 指向 cur， 实现一次局部反转

cur 和 pre 向前移动一个位置

循环交换前进， 直至 pre 为空， 遍历结束， 完成反转， 此时 cur 节点为开始节点 head；

参考代码：

```python
class Solution:
    def reverseList(self, head:  ListNode) -> ListNode:
        cur, pre = None, head
        while pre != None:
            next_node = pre.next
            pre.next = cur
            cur, pre = pre, next_node
        return cur
```

10.3 连续子数组的最大乘积

https：//leetcode-cn.com/problems/maximum-product-subarray/

思路：

遍历数组时计算当前最大值、最小值， 不断更新

当前最大值为 ans_max = max(ans_max * nums[i], nums[i])

当前最小值为 ans_min = min(ans_min * nums[i], nums[i])

由于存在负数， 那么会导致最大的变最小的， 最小的变最大的。

当前最大值为 ans_max = max(ans_min * nums[i], nums[i])

当前最小值为 ans_min = min(ans_max * nums[i], nums[i])

参考代码：

```python
class Solution:
    def maxProduct(self, nums:  List[int]) -> int:
        if len(nums) == 1:  return nums[0]
        ans_max = nums[0]   # 当前最大值
        ans_min = nums[0]    #当前最小值
        ans = nums[0]  # 当前最大乘积
        # nums[i]分两种情况：大于 0， 小于 0
        for i in range(1, len(nums)):
            # 小于 0 时，
            if nums[i] < 0:
                tmp = ans max
                ans_max = max(ans_min*nums[i], nums[i])
                ans_min = min(tmp*nums[i], nums[i])
                ans = max(ans_max, ans)
            # 大于等于 0 时
            else:
                ans_max = max(ans_max*nums[i], nums[i])
                ans_min = min(ans_min*nums[i], nums[i])
                ans = max(ans_max, ans)
        return ans
```

10.4 最大子数组和

https：//leetcode-cn.com/probl ms/maximum-subarray/description/

解题思路：

遍历数组， 遍历的时候记录两个值： 当前子数组的和 tmpSum， 最大值 res

```
1.  class Solution:
2.      def maxSubArray(self, nums:  List[int]) -> int :
3.          tmp_sum = 0
4.          res = nums[0]
5.          for num in nums:
6.              tmp_sum = max(tmp_sum + num, num)
7.              res = max(res, tmp_sum)
8.          return res
```

10.5 抛硬币游戏

抛硬币游戏， 如果在连续抛出三次正面之前不要停下来， 那么我们总计抛硬币的期望次数是多少

假设期望是 x

假设第一抛是反面， 那么就浪费了一步， 平均一共需要 x+1 步(概率是 1/2)

假设第一抛是正面， 在此基础上如果第二抛是反面， 又浪费了， 平均一共需要 x+2 步（概率是 1/4)

在此基础上如果第二抛是正面

假设第三抛反面， 浪费， 平均一共 x+3 步（概率是 1/8 ）

假设第三抛正面， 完成， 只用了 3 步（概率是 1/8 ）

所以 x 的期望即 x =(1/2)(x+1) +(1/4)(x+2) +(1/8)(x+3) +(1/8)*3

解得 x =14

第十一篇 阿里蚂蚁金服算法岗实习面试题（一二面）6 道

11.1 使用 Word2vec 算法计算得到的词向量之间为什么能够表征词语之间的语义近似关系?

参考答案：

word2vec 是一种高效实现 word embedding 的算法， word2vec 模型其实就是一个简单化的神经网络， 输入是 One-Hot 向量， Hidden Layer 没有激活函数， 也就是线性的单元。Output Layer 维度跟 Input Layer 的维度一样， 用的是 Softmax 回归。word2vec 得出的词向量其实就是训练后的一个神经网络的隐层的权重矩阵， 经过 CBOW 或 Skip-Gram 模型的训练后， 此意相近的词语就会获得更为接近的权重， 因此可以用向量的距离来衡量词的相似度。

11.2 在样本量较少的情况下如何扩充样本数量？

参考答案：

1. 同义词替换（ SR： Synonyms Replace ）：不考虑 stopwords， 在句子中随机抽取 n 个词， 然后从同义词词典中（ wordnet ）随机抽取同义词， 并进行替换。

2. 随机插入(RI： Randomly Insert)： 不考虑 stopwords， 随机抽取一个词， 然后在该词的同义词集合中随机选择一个， 插入原句子中的随机位置。该过程可以重复 n 次。

3. 随机交换(RS： Randomly Swap)： 句子中， 随机选择两个词， 位置交换。该过程可以重复 n 次。

4. 随机删除(RD： Randomly Delete)： 句子中的每个词， 以概率 p 随机删除。

11.3 介绍一下 Python 的装饰器。

参考答案：

装饰器本质上是一个 Python 函数或类， 它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能， 装饰器的返回值也是一个函数/类对象。

装饰器的作用是装饰函数， 即在不改变原有函数的基础上， 增加新的函数功能， 让函数更加强大。

装饰器适用的两个场景： 增强被装饰函数的行为； 代码复用。

11.4 什么是梯度消失和梯度爆炸？

参考答案：

根据损失函数计算的误差通过梯度反向传播的方式对深度网络权值进行更新时， 得到的梯度值接近 0 或特别大， 也就是梯度消失或爆炸。梯度消失或梯度爆炸在本质原理上其实是一样的。

11.5 leetcode46 全排列

预备知识

回溯法： 一种通过探索所有可能的候选解来找出所有的解的算法。如果候选解被确认不是一个解（或者至少不是最后一个解），回溯算法会通过在上一步进行一些变化抛弃该解，即回溯并且再次尝试。

方法一： 回溯思路和算法这个问题可以看作有 n 个排列成一行的空格，我们需要从左往右依此填入题目给定的 n 个数，每个数只能使用一次。那么很直接的可以想到一种穷举的算法，即从左往右每一个位置都依此尝试填入一个数，看能不能填完这 n 个空格，在程序中我们可以用「回溯法」来模拟这个过程。我们定义递归函数 backtrack(first, output) 表示从左往右填到第 first 个位置，当前排列为 output。

那么整个递归函数分为两个情况：

如果 first==n，说明我们已经填完了 n 个位置（注意下标从 0 开始），找到了一个可行的解，我们将 output 放入答案数组中，递归结束。

如果 first<n，我们要考虑这第 first 个位置我们要填哪个数。根据题目要求我们肯定不能填已经填过的数，因此很容易想到的一个处理手段是我们定义一个标记数组 vis[]来标记已经填过的数，那么在填第 first 个数的时候我们遍历题目给定的 n 个数，如果这个数没有被标记过，我们就尝试填入，并将其标记，继续尝试填下一个位置，即调用函数 backtrack(first + 1, output)。回溯的时候要撤销这一个位置填的数以及标记，并继续尝试其他没被标记过的数。

使用标记数组来处理填过的数是一个很直观的思路，但是可不可以去掉这个标记数组呢？毕竟标记数组也增加了我们算法的空间复杂度。

答案是可以的，我们可以将题目给定的 n 个数的数组 nums 划分成左右两个部分，左边的表示已经填过的数，右边表示待填的数，我们在回溯的时候只要动态维护这个数组即可。具体来说，假设我们已经填到第 first 个位置，那么 nums 数组中[0,first−1][0,first−1] 是已填过的数的集合，[first,n−1][first,n−1] 是待填的数的集合。我们肯定是尝试用[first,n−1][first,n−1] 里的数去填第 first 个数，假设待填的数的下标为 ii ，那么填完以后我们将第 ii 个数和第 first 个数交换，即能使得在填第 first+1 个数的时候 nums 数组的[0,first][0,first] 部分为已填过的数，[first+1,n−1][first+1,n−1] 为待填的数，回溯的时候交换回来即能完成撤销操作。

举个简单的例子，假设我们有 [2, 5, 8, 9, 10] 这 5 个数要填入，已经填到第 3 个位置，已经填了 [8,9] 两个数，那么这个数组目前为 [8, 9 | 2, 5, 10] 这样的状态，分隔符区分了左右两个部分。假设这个位置我们要填 10 这个数，为了维护数组，我们将 2 和 10 交换，即能使得数组继续保持分隔符左边的数已经填过，右边的待填 [8, 9, 10 | 2, 5] 。当然善于思考的读者肯定已经发现这样生成的全排列并不是按字典序存储在答案数组中的，如果题目要求按字典序输出，那么请还是用标记数组或者其他方法。

```
1.  class Solution:
2.      def permute(self, nums):
3.          """
4.          :type nums: List[int]
5.          :rtype: List[List[int]]
6.          """
7.          def backtrack(first = 0):
8.              # 所有数都填完了
9.              if first == n:
10.                 res.append(nums[:])
11.             for i in range(first, n):
12.                 # 动态维护数组
13.                 nums[first], nums[i] = nums[i], nums[first]
14.                 # 继续递归填下一个数
```

```
                backtrack(first + 1)
                # 撤销操作
                nums[first], nums[i] = nums[i], nums[first]

        n = len(nums)
        res = []
        backtrack()
        return res
```

11.6 在两个排列数组中各取一个数，使得两个数的和为 m

参考答案：

思路：

最容易想到的方法是枚举数组中的每一个数 x，寻找数组中是否存在 target - x。

当我们使用遍历整个数组的方式寻找 target - x 时，需要注意到每一个位于 x 之前的元素都已经和 x 匹配过，因此不需要再进行匹配。而每一个元素不能被使用两次，所以我们只需要在 x 后面的元素中寻找 target - x。

代码：

```
class Solution:
    def twoSum(self, nums:  List[int], target:  int) -> List[int]:
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                if nums[i] + nums[j] == target:
                    return [i, j]
        return []
```

第十二篇 腾讯应用研究岗暑期实习面试题 12 道

12.1 决策树有多少种，分别的损失函数是什么？

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：

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

Cart 树损失函数：

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

12.2 决策树的两种剪枝策略分别是什么？

决策树的剪枝基本策略有 预剪枝 (Pre-Pruning) 和 后剪枝 (Post-Pruning)。

预剪枝核心思想：

在每一次实际对结点进行进一步划分之前， 先采用验证集的数据来验证如果划分是否能提高划分的准确性。如果不能， 就把结点标记为叶结点并退出进一步划分； 如果可以就继续递归生成节点。

后剪枝核心思想：

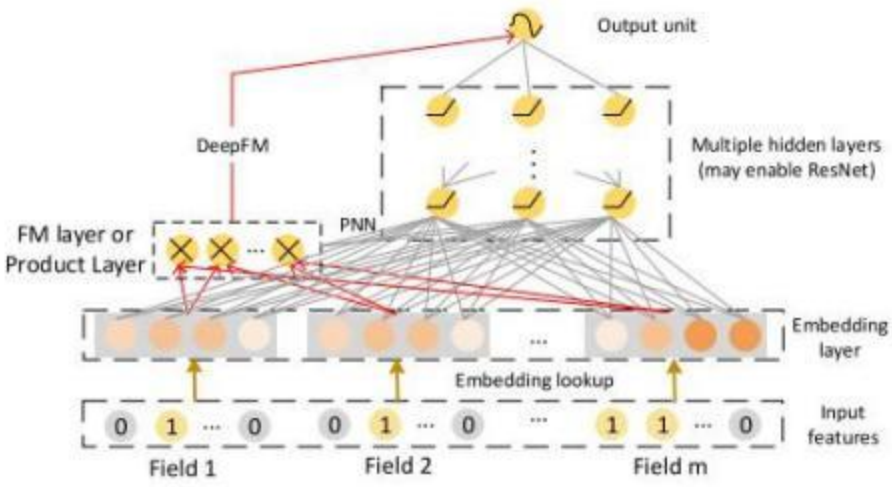
后剪枝则是先从训练集生成一颗完整的决策树， 然后自底向上地对非叶结点进行考察， 若将该结点对应的子树替换为叶结点能带来泛化性能提升， 则将该子树替换为叶结点。

12.3 信息增益比跟信息增益相比，优势是什么？

以信息增益作为划分训练集的特征选取方案， 存在偏向于选取值较多的特征的问题。

信息增益比可以解决该问题。

12.4 介绍 XdeepFM 算法、XdeepFM 跟 DeepFM 算法相比，优势是什么？



上图为 xDeepFM 的总体结构， 有三个分支： Linear（稀疏的 01 向量作为输入）、 DNN（经过 embedding 的稠密向量作为输入）、CIN（压缩感知层）。

xDeepFM 如果去掉 CIN 分支， 就等同于 Wide & Deep。

xDeepFM 将基于 Field 的 vector-wise 思想引入 Cross， 并且保留了 Cross 的优势， 模型结构也很 elegant， 实验效果也提升明显。如果说 DeepFM 只是“Deep & FM”， 那么 xDeepFm 就真正做到了” Deep” Factorization Machine。xDeepFM 的时间复杂度会是其工业落地的一个主要性能瓶颈， 需要重点优化。

12.5 对于长度较长的语料，如何使用 Bert 进行训练？

对于长文本， 有两种处理方式， 截断和切分。

- 截断： 一般来说文本中最重要的信息是开始和结尾， 因此文中对于长文本做了截断处理。

> head-only： 保留前 510 个字符

> tail-only： 保留后 510 个字符

> head + tail： 保留前 128 个和后 382 个字符

- 切分 ： 将文本分成 k 段， 每段的输入和 Bert 常规输入相同， 第一个字符是[CLS]表示这段的加权信息。文中使用了 Max-pooling, Average pooling 和 self-attention 结合这些片段的表示。

12.6 请介绍 k-mean 算法的原理

1、选取 K 个点做为初始聚集的簇心

2、分别计算每个样本点到 K 个簇核心的距离（这里的距离一般取欧氏距离或余弦距离），找到离该点最近的簇核心， 将它归属到对应的簇

3、所有点都归属到簇之后， M 个点就分为了 K 个簇。之后重新计算每个簇的重心（平均距离中心），将其定为新的“簇核心” ；

4、反复迭代 2 - 3 步骤， 直到达到某个中止条件。

12.7 逻辑回归怎么分类非线性数据？

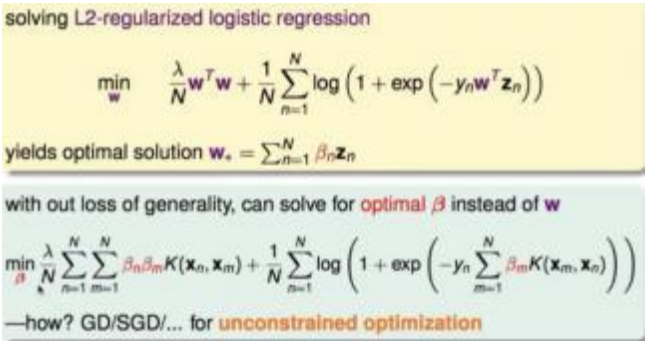
可以， 只要使用 kernel trick。

不过， 通常使用的 kernel 都是隐式的， 也就是找不到显式地把数据从低维映射到高维的函数， 而只能计算高维空间中数据点的内积。在这种情况下， logistic regression 模型就不能再表示成 "ztb 的形式（ primal form ），而只能表示成 $\sum_i a_i \langle x_i, x \rangle + b$ 的形式（ dual form ）。忽略那个 b 的话， primal form 的模型的参数只有 w ， 只需要一个数据点那么多的存储量； 而 dual form 的模型不仅要存储各个 a_i ，还要存储训练数据 x_i 本身， 这个存储量就大了。

SVM 也是具有上面两种形式的。不过， 与 logistic regression 相比， 它的 dual form 是稀疏的——只有支持向量的 a_i 才非零， 才需要存储相应的 x_i 。所以， 在非线性可分的情况下， SVM 用得更多。

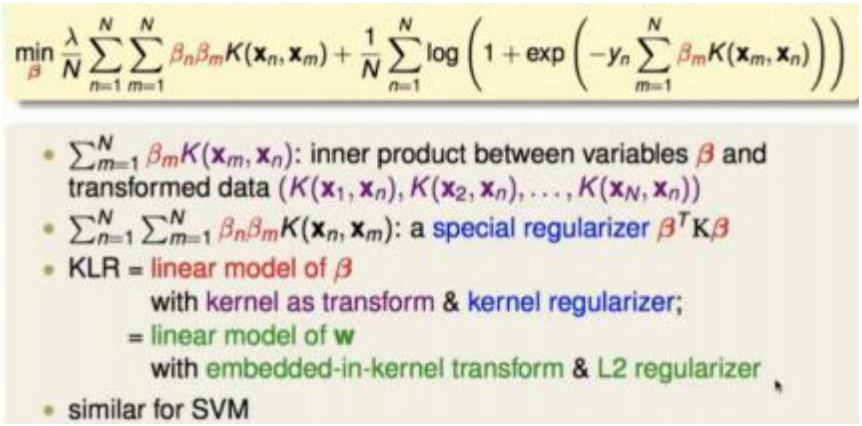
12.8 逻辑回归引入核方法后损失函数如何求导？

用核函数的方法来解决一个 L2 正则化的逻辑回归如下图：



我们直接将 W*表示成β的形式带到我们最佳化的问题中， 然后就得到一个关于β的无条件的最佳化问题。这时我们可以用梯度下降法或随机梯度下降法来得到问题的最优解。

再仔细观察核函数逻辑回归之后会发现它可以是一个关于β的线性模型：



其中 kernel 函数既充当了转换的角色有充当了正则化的角色， 这种角度同样适用于 SVM 演算法。需要注意的是： SVM 的解α大多都是 0， 核函数逻辑回归的解β大多都不是 0 这样我们会付出计算上的代价。

12.9 请介绍几种常用的参数更新方法

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。

12.10 请介绍 Wide&Deep 模型

Memorization 和 Generalization

Wide&Deep Mode 就是希望计算机可以像人脑一样，可以同时发挥 memorization 和 generalization 的作用。--Heng-Tze Cheng(Wide&Deep 作者)

Wide 和 Deep

同样，这两个词也是通篇出现，究竟什么意思你明白了没？

其实，Wide 也是一种特殊的神经网络，他的输入直接和输出相连。属于广义线性模型的范畴。Deep 就是指 Deep Neural Network，这个很好理解。Wide Linear Model 用于 memorization；Deep Neural Network 用于 generalization。左侧是 Wide-only，右侧是 Deep-only，中间是 Wide & Deep：

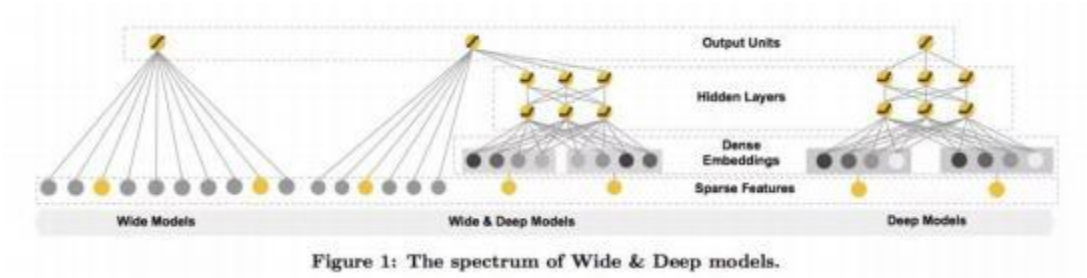


Figure 1: The spectrum of Wide & Deep models.

Cross-product transformation

Wide 中不断提到这样一种变换用来生成组合特征，也必须搞懂才行哦。它的定义如下：

$$\phi_k(\mathbf{x}) = \prod_{i=1}^{d} x_i^{c_{ki}} \quad c_{ki} \in \{0, 1\}$$

k 表示第 k 个组合特征。i 表示输入 X 的第 i 维特征。C_ki 表示这个第 i 维度特征是否要参与第 k 个组合特征的构造。d 表示输入 X 的维度。那么到底有哪些维度特征要参与构造组合特征那？这个是你之前自己定好的，在公式中没有体现。

饶了一大圈，整这么一个复杂的公式，其实就是我们之前一直在说的 one-hot 之后的组合特征：仅仅在输入样本 X 中的特征 gender=female 和特征 language=en 同时为 1，新的组合特征 AND(gender=female, language=en)才为 1。所以只要把两个特征的值相乘就可以了。

Cross-product transformation 可以在二值特征中学习到组合特征， 并且为模型增加非线性

Wide & Deep Model

Memorization： 之前大规模稀疏输入的处理是： 通过线性模型 + 特征交叉。所带来的 Memorization 以及记忆能力非常有效和可解释。但是 Generalization（泛化能力 ）需要更多的人工特征工程。

Generalization： 相比之下， DNN 几乎不需要特征工程。通过对低纬度的 dense embedding 进行组合可以学习到更深层次的隐藏特征。但是， 缺点是有点 over-generalize（过度泛化）。推荐系统中表现为： 会给用户推荐不是那么相关的物品， 尤其是 user-item 矩阵比较稀疏并且是 high-rank（高秩矩阵 ）

两者区别： Memorization 趋向于更加保守， 推荐用户之前有过行为的 items。相比之下， generalization 更加趋向于提高推荐系统的多样性（diversity ）。

Wide & Deep： Wide & Deep 包括两部分： 线性模型 + DNN 部分。结合上面两者的优点， 平衡 memorization 和 generalization。原因： 综合 memorization 和 generalizatio 的优点， 服务于推荐系统。相比于 wide-only 和 deep-only 的模型， wide & deep 提升显著（这么比较脸是不是有点大）

12.11 Xgboost、lightGBM 和 Catboost 之间的异同？

树的特征

三种算法基学习器都是决策树， 但是树的特征以及生成的过程仍然有很多不同

CatBoost 使用对称树， 其节点可以是镜像的。CatBoost 基于的树模型其实都是完全二叉树。

XGBoost 的决策树是 Level-wise 增长。Level-wise 可以同时分裂同一层的叶子， 容易进行多线程优化， 过拟合风险较小， 但是这种分裂方式也有缺陷， Level-wise 对待同一层的叶子不加以区分， 带来了很多没必要的开销。实际上很多叶子的分裂增益较低， 没有搜索和分裂的必要。

LightGBM 的决策树是 Leaf-wise 增长。每次从当前所有叶子中找到分裂增益最大的一个叶子（通常来说是数据最多的一个）， 其缺陷是容易生长出比较深的决策树， 产生过拟合， 为了解决这个问题， LightGBM 在 Leaf-wise 之上增加了一个最大深度的限制。

对于类别型变量

调用 boost 模型时， 当遇到类别型变量， xgboost 需要先处理好， 再输入到模型， 而 lightgbm 可以指定类别型变量的名称， 训练过程中自动处理。

具体来讲， CatBoost 可赋予分类变量指标， 进而通过独热最大量得到独热编码形式的结果（独热最大量： 在所有特征上， 对小于等于某个给定参数值的不同的数使用独热编码； 同时， 在 CatBoost 语句中设置“跳过” ， CatBoost 就会将所有列当作数值变量处理）。

LighGBM 也可以通过使用特征名称的输入来处理属性数据； 它没有对数据进行独热编码， 因此速度比独热编码快得多。LGBM 使用了一个特殊的算法来确定属性特征的分割值。（注： 需要将分类变量转化为整型变量； 此算法不允许将字符串数据传给分类变量参数）

和 CatBoost 以及 LGBM 算法不同， XGBoost 本身无法处理分类变量， 只接受数值数据， 这点和 RF 很相似。实际使用中， 在将分类数据传入 XGBoost 之前， 必须通过标记编码、均值编码或独热编码等各种编码方式对数据进行处理。

12.12 情景题：找到词频高的词

有一个大小为 1G 的文件， 文件中每行有一个词， 每个词最大为 16kb； 现有内存为 1M 的计算机， 找出词频前 100 的词。

参考答案：

1.分而治之/hash 映射

顺序读取文件， 对于每个词 x， 取 hash(x)%5000， 然后把该值存到 5000 个小文件（记为 x0,x1,...x4999 ）中。这样每个文件大概是 200k 左右。当然， 如果其中有的小文件超过了 1M 大小， 还可以按照类似的方法继续往下分， 直到分解得到的小文件的大小都不超过 1M。

2.hash_map 统计

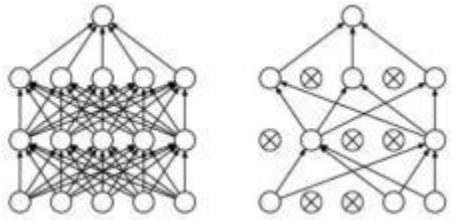
对每个小文件， 采用 trie 树/hash_map 等统计每个文件中出现的词以及相应的频率。

3.堆/归并排序

取出出现频率最大的 100 个词（可以用含 100 个结点的最小堆）后， 再把 100 个词及相应的频率存入文件， 这样又得到了 5000 个文件。最后就是把这 5000 个文件进行归并（类似于归并排序）的过程了。

第十三篇 2024 年 5 月 26 日好未来机器学习算法工程师暑期实习面试题 2 道

13.1 讲讲 dropout 原理



Dropout 目标： 缓解过拟合、减少收敛时间

Dropout 原理可以总结位： 网络前向传播的时， 某个神经元的激活值以一定的概率值 p 停止工作，这样可以使网络不太依赖某些局部特征， 可以使模型泛化性更强；

P： 表示丢弃概率， 即使神经元失活的概率； 为了保证训练和测试时神经元数量分布一致， 有两种模式： upscale_in_train'和'downscale_in_infer'

upscale_in_train ： 在训练时增大输出结果。

train ： out = input * mask / (1.0 - p)

inference ： out = input

downscale_in_infer： 在预测时减小输出结果

train ： out = input * mask

inference ： out = input * (1.0 - p)

举例说明： 比如网络共有 10 个可训练参数， 训练时， 丢弃概率设置为 0.2， 那么有 8 个参数训练。但是测试时不使用 Dropout 就会有 10 个参数， 为保证两者相等， 可以训练时保留的 8 个神经元权重乘以 1/(1-0.2) =8 或者测试时所有神经元权重参数乘 0.8 （ 10*0.8 =8 ） .

13.2 算法题：判断是否是平衡二叉树

平衡二叉树定义： 二叉树每个节点的左右子树高度差不超过 1；

思路： 平衡二叉树满足根节点及其子节点都是平衡二叉树， 所以需要递归计算每个节点左右子树的高度， 然后判断高度差是否不超过 1； 参考代码如下：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
```

```
#         self.right = right
class Solution:
    def isBalanced(self, root:  TreeNode) -> bool:
        if not root :  return True
        result = abs(self.getHeight(root.right) -
 self.getHeight(root.left)) <= 1 and self.isBalanced(root.left) and self.isBa
lanced(root.right)
        return result
    def getHeight(self, root:  TreeNode):
        if not root:  return 0
        return max(self.getHeight(root.left), self.getHeight(root.right)) + 1
```

第十四篇 2024 年 4 月百度 机器学习/数据挖掘/NLP 算法工程师实习面试题 8 道

14.1 编程题旋转有序数组，查找元素是否存在

思路：

暴力破解：遍历整个数组，查找元素是否存在；

二分查找：旋转后局部数组依然是有序的，所以此时依然可以使用二分查找算法；

参考代码：

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # 方法 1：保留破解
        # return nums.intdex(target) if target in nums else -1

        # 方法 2：二分查找
        if not nums: return -1

        left, right = 0, len(nums) - 1

        while left <= right:
            medium = (left + right) // 2
            if nums[medium] == target: return medium

            if nums[left] <= nums[medium]:
                if nums[left] <= target < nums[medium]:
                    right = medium - 1
                else:
                    left = medium + 1
            else:
                if nums[medium] < target <= nums[right]:
                    left = medium + 1
                else:
                    right = medium - 1
        return -1
```

14.2 实现余弦相似度计算

余弦相似度：用两个向量夹角判断其相似程度。向量夹角越大，距离越远，最大距离就是两个向量夹角 180°；向量夹角越小，距离越近，最小距离就是两个向量夹角 0°，完全重合。

所以余弦相似度越大，向量越相似；

计算公式：

$$\cos(\theta) = \frac{a \cdot b}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \times \sqrt{\sum_{i=1}^{n} (b_i)^2}}$$

求余弦相似度方法：

Numpy：

```
def cal_cosineSimilarity_numpy(a,b):
    dot = a*b
    a_len = np.linalg.norm(a,axis=1)
    b_len = np.linalg.norm(b,axis=1)
    cosineSimilarity = dot.sum(axis=1)/(a_len*b_len)
```

Pytorch

```
def methodOne(a,b):
    d = torch.mul(a, b)#计算对应元素相乘
    a_len = torch.norm(a,dim=1)#2 范数：模长
    b_len = torch.norm(b,dim=1)
    cosineSimilarity = torch.sum(d, dim=1)/(a_len*b_len)#得到相似度

def methodTwo(a,b):
    cosineSimilarity = torch.cosine_similarity(a,b,dim=1)
```

Sklearn

```
from sklearn.metrics.pairwise import cosine_similarity
def cal_cosineSimilarity_sklearn(a, b)
    cosineSimilarity = cosine_similarity(a,b)
```

14.3 验证二叉搜索树(BST)

二叉搜索树具有如下**特征**：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

思路：

根据二叉搜索树的特征可知： 如果二叉树的左子树不为空， 则左子树上所有节点的值均小于它根节点的值； 若它的右子树不空， 则右子树上所有节点的值均大于它根节点的值； 并且它的左右子树也为二叉搜索树。

可以设计一个递归函数 check(root, max_num, min_num) ， 函数表示考虑以 root 为根的子树， 判断子树中所有节点的值是否都在 (max_num, min_num) 的范围内。如果 root 节点的值 val 不在 (max_num, min_num)的范围内说明不满足条件直接返回 False， 否则继续递归， 检查它的左右子树是否满足， 都满足则说明这是一棵二叉搜索树。

注意： 在递归调用左子树时， 需要把上界 max_num 改为 root.val； 递归调用右子树时， 需要把下界 min_num 改为 root.val.

函数递归调用的入口为 check(root, float("inf"), -float("inf")) ， float("inf")表示一个无穷大的值。

参考代码：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        return self.valTree(root, float("inf"), -float("inf"))
    def valTree(self, root: TreeNode, max_num, min_num):
        if not root: return True
        val = root.val
        if val >= max_num or val <= min_num: return False
        if not self.valTree(root.left, val, min_num): return False
        if not self.valTree(root.right, max_num, val): return False
        return True
```

14.4 用 randomInt(5)实现 randomInt(7)，只用讲思路

randomInt(5)：等概率生成整数[1, 2, 3, 4, 5]

randomInt(7) ：等概率生成整数[1, 2, 3, 4, 5, 6, 7]

思路：

用(randomInt(5) - 1)构造等概率整数数组： [0, 1, 2 , 3, 4],

用(randomInt(5) - 1)*5 构造整数数组： [0, 5, 10, 15, 20],

上面的两个整数组可以构造等概率的新数组： [0, 1, 2, 3, 4, 5, 6, 7, ... , 24];

(如果第 2 个数组选择 2 倍或者 3 倍， 4 倍则无法构造新的等概率数组)

选择新数组[0, 1, 2, 3 .... , 20]21 个元组即可构造等概率的数组[1, 2, 3, 4, 5, 6, 7]

参考代码：

```
import random

def rand5():
    return random.randint(1, 5)

def rand7():
    while True:
        tmp_num = (rand5() - 1)*5 + (rand5() - 1)
        if tmp_num <= 20:
            break
    return tmp_num % 7 + 1

# 随机生成 10000 个 1~7 的整数，验证 0~7 各个数字生成的概率；
ls = []
for _ in range(10000):
    ls.append(rand7())
```

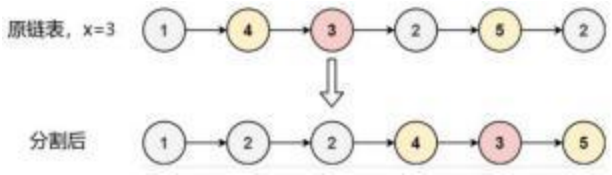
```
for i in range(1, 8):
    print(ls.count(i)/len(ls), end=" ")
# out: 0.144 0.1459 0.14 0.1438 0.1444 0.1373 0.1446
# 观察可知 0~7 各个数字生成的概率是等价的
```

14.5 编程题： 分割链表问题

分割链表：

给定一个链表的头节点 head 和一个特定值 x，然后对链表进行分隔，使得所有小于 x 的节点都出现在大于或等于 x 的节点之前。同时保留两个分区中每个节点的初始相对位置。

如下图所示：



思路：

需维护两个链表 small 和 large，small 链表按顺序存储所有小于 x 的节点，large 链表按顺序存储所有大于等于 x 的节点。遍历完原链表后，我们只要将 small 链表尾节点指向 large 链表的头节点即能完成对链表的分隔。

参考代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def partition(self, head:  ListNode, x:  int) -> ListNode:
        small = ListNode()
        large = ListNode()
        smallHead, largeHead = small, large
        while head:
            if head.val < x:
                small.next = head
                small = small.next
            else:
                large.next = head
                large = large.next
            head = head.next
        large.next = None
        small.next = largeHead.next
        return smallHead.next
```

14.6 怎么解决过拟合？怎么做图像增广？

常见缓解过拟合的方法：

- 降低模型复杂度
- 增加更多的训练数据： 使用更大的数据集训练模型
- 数据增强
- 正则化： L1、L2、添加 BN 层
- 添加 Dropout 策略
- Early Stopping
- 重新清洗数据： 把明显异常的数据剔除
- 使用集成学习方法： 把多个模型集成在一起， 降低单个模型的过拟合风险

常见的数据增广方法：

- 水平/垂直翻转
- 随机旋转
- 随机缩放
- 随机剪切
- 颜色、对比度增强
- CutOut
- CutMix
- Mixup
- Mosaic
- Random Erasing

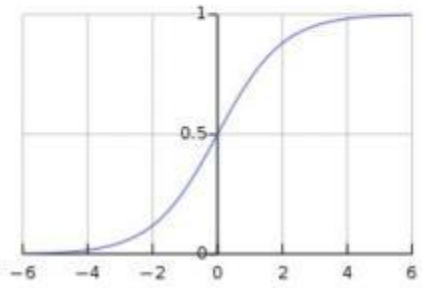
14.7 梯度下降方法有哪些？

梯度下降算法有如下 3 种：

- 随机梯度下降法： SGD
- 批量梯度下降法： BGD
- min-batch 小批量梯度下降法： MBGD

14.8 Sigmoid 有哪些特性？激活函数了解多少？

$$S\left(x\right)=\frac{1}{1+e^{-x}}$$



Sigmod 函数性质：

定义域：（−∞，+∞）；

值域：（− 1，1）；

函数在定义域内为连续光滑函数；

处处可导，导数为：$S'(x)=\frac{e^{-x}}{(1+e^{-x})^2}=S(x)(1-S(x))$；

函数的取值在 0 到 1 之间，在 0.5 处呈中心对称，且越靠近 x = 0 的取值斜率越大。

常见激活函数：

Sigmoid

Tanh

Relu、Leaky Relu、P-Relu (Parametric ReLU)

Elu、Gelu

Swich

Selu

第十五篇 5 月 24 日-5 月 27 日，好未来算法实习面试题 8 道

15.1 lr 为什么要用极大似然？

参考答案：

因为我们想要让每一个样本的预测都要得到最大的概率，即将所有的样本预测后的概率进行相乘都最大，也就是极大似然函数。

对极大似然函数取对数以后相当于对数损失函数，由梯度更新的公式可以看出，对数损失函数的训练求解参数的速度是比较快的，而且更新速度只和 x，y 有关，比较的稳定

为什么不用平方损失函数？如果使用平方损失函数，梯度更新的速度会和 sigmod 函数的梯度相关，sigmod 函数在定义域内的梯度都不大于 0.25，导致训练速度会非常慢。而且平方损失会导致损失函数是 theta 的非凸函数，不利于求解，因为非凸函数存在很多局部最优解

15.2 讲一下 lgb 的直方图是怎么用的？

参考答案：

基本思想：先把连续的浮点特征值离散化成 k 个整数，同时构造一个宽度为 k 的直方图。

在遍历数据时：

a. 根据离散化后的值作为索引在直方图中累积统计量。

b. 当遍历一次数据后，直方图累积了需要的统计量。

c. 然后根据直方图的离散值，遍历寻找最优的分割点。

优点：节省空间。假设有 个样本，每个样本有 个特征，每个特征的值都是 32 位浮点数。

a. 对于每一列特征，都需要一个额外的排好序的索引（32 位的存储空间）。则 pre-sorted 算法需要消耗 字节内存。

b. 如果基于 histogram 算法，仅需要存储 feature bin value （离散化后的数值），不需要原始的 feature value，也不用排序。而 bin value 用 unit8_t 即可，因此 histogram 算法消耗 字节内存，是预排序算法的 。

缺点：不能找到很精确的分割点，训练误差没有 pre-sorted 好。但从实验结果来看，histogram 算法在测试集的误差和 pre-sorted 算法差异并不是很大，甚至有时候效果更好。

- 实际上可能决策树对于分割点的精确程度并不太敏感，而且较“粗”的分割点也自带正则化的效果。
- 采用 histogram 算法之后，寻找拆分点的算法复杂度为：

a. 构建 histogram：$O(Nx\ n)$。

b. 寻找拆分点：$O(NX\ K)$，其中 k 为分桶的数量。

- 与其他算法相比：

a. scikit-learn GBDT、gbm in R 使用的是基于 pre-sorted 的算法。

b. pGBRT 使用的是基于 histogram 的算法。

c. xgboost 既提供了基于 pre-sorted 的算法，又提供了基于 histogram 的算法。

d. lightgbm 使用的是基于 histogram 的算法。

15.3 链表判断有无环

给定一个链表，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。

思路：

- 哈希表法：用哈希表存储已遍历过的节点，如果后续哈希表遇到相同节点，则表明有环，否则无环；
- 快慢指针法：定义快慢两个指针，快指针每次移动 2 个节点，满节点每次移动 1 个节点，如果快慢指针相遇，则有环，否则无环；

参考答案：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def hasCycle(self, head:  ListNode) -> bool:
        # 哈希表法
        # checked = set()
        # while head:
        #     if head in checked:
        #         return True
        #     checked.add(head)
        #     head = head.next
        # return False

        # 快慢指针法
        if not head or not head.next:  return False
        slow, quick = head, head.next
        # 判断快慢指针是否相遇：
        while slow != quick:
            if not quick or not quick.next:  return False
            slow, quick = slow.next, quick.next.next
        return True
```

15.4 二叉树路径

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明： 叶子节点是指没有子节点的节点。

示例:

输入:

输出: ["1->2->5", "1->3"]

解释: 所有根节点到叶子节点的路径为: 1->2->5, 1->3

思路： 可以使用深度优先搜索方法遍历整个二叉树， 对于叶子节点， 添加当前路径； 对于非叶子节点，递归遍历其子节点；

参考代码：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def binaryTreePaths(self, root:  TreeNode) -> List[str]:
        if not root:  return root
        result = []
        self.helper(root, result, "")
        return result

    def helper(self, root, result, path):
        if not root:  return result
        path += str(root.val)
        # 叶子节点
        if  not root.left and not root.right:
            result.append(path)
        # 非叶子节点
        else:
            path += "->"
            self.helper(root.left, result, path)
            self.helper(root.right, result, path)
        return result
```

15.5 集成学习 boosting 和 bagging 的概念

参考答案：

Bagging 算法(套袋发)

bagging 的算法过程如下：

- 从原始样本集中使用 Bootstraping 方法随机抽取 n 个训练样本， 共进行 k 轮抽取， 得到 k 个训练集（k 个训练集之间相互独立， 元素可以有重复）。
- 对于 n 个训练集， 我们训练 k 个模型， (这个模型可根据具体的情况而定， 可以是决策树， knn 等）
- 对于分类问题： 由投票表决产生的分类结果； 对于回归问题， 由 k 个模型预测结果的均值作为最后预测的结果（所有模型的重要性相同）。

Boosting（提升法）

boosting 的算法过程如下：

- 对于训练集中的每个样本建立权值 wi， 表示对每个样本的权重， 其关键在与对于被错误分类的样本权重会在下一轮的分类中获得更大的权重（错误分类的样本的权重增加）。
- 同时加大分类 误差概率小的弱分类器的权值， 使其在表决中起到更大的作用， 减小分类误差率较大弱分类器的权值， 使其在表决中起到较小的作用。每一次迭代都得到一个弱分类器， 需要使用某种策略将其组合， 最为最终模型， (adaboost 给每个迭代之后的弱分类器一个权值， 将其线性组合作为最终的分类器,误差小的分类器权值越大。）

Bagging 和 Boosting 的**主要区别：**

样本选择上： Bagging 采取 Bootstraping 的是随机有放回的取样， Boosting 的每一轮训练的样本是固定的， 改变的是买个样的权重。

样本权重上： Bagging 采取的是均匀取样， 且每个样本的权重相同， Boosting 根据错误率调整样本权重， 错误率越大的样本权重会变大

预测函数上： Bagging 所以的预测函数权值相同， Boosting 中误差越小的预测函数其权值越大。

并行计算： Bagging 的各个预测函数可以并行生成;Boosting 的各个预测函数必须按照顺序迭代生成.

15.6 bert 的改进版有哪些

参考答案：

RoBERTa： 更强大的 BERT

- 加大训练数据 16GB -> 160GB， 更大的 batch size， 训练时间加长
- 不需要 NSP Loss： natural inference
- 使用更长的训练 Sequence
- Static vs. Dynamic Masking
- 模型训练成本在 6 万美金以上（估算）

ALBERT： 参数更少的 BERT

- 一个轻量级的 BERT 模型
- 共享层与层之间的参数（减少模型参数）
- 增加单层向量维度

- DistilBERT：轻量版 BERT

15.7 Leetcode88- 合并两个有序数组

给出两个有序的整数数组 A 和 B，请将数组 B 合并到数组 A 中，变成一个有序的数组。注意：可以假设 A 数组有足够的空间存放 B 数组的元素，A 和 B 中初始的元素数目分别为 m 和 n。

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        # 方法 1: 直接合并，然后排序
        # nums1[m: ] = nums2
        # nums1.sort()
        # 方法 2: 双指针法
        # 因为两个列表均已是有序，所以可以比较 nums1 和 nums2 的首端元素，将较小者插入到前面
        # p1: nums1 的指针  p2: nums2 的指针
        p1, p2 = 0, 0
        nums1_ = [nums1[i] for i in range(m)]
        i = 0
        while p1 <= m and p2 <= n:
            # 判断 nums1 是否已遍历完
            if p1 == m:
                nums1[m+p2: ] = nums2[p2: ]
                break
            # 判断 nums2 是否已遍历完
            elif p2 == n:
                nums1[n+p1: ] = nums1_[p1: ]
                break

            elif nums1_[p1] <= nums2[p2]:
                nums1[i] = nums1_[p1]
                p1, i = p1+1, i+1
            else:
                nums1[i] = nums2[p2]
                p2, i = p2+1, i+1
```

15.8 1-26 对应 a-z 字符串转换

字母 a-z 对应数字 1-26，给定一个数字序列， 求所有可能的解码总数，例如 1261 对应解码为 1 2 6 1，12 6 1，1 26 1 总共为 3 种方式

参考答案：

这其实是一道字符串类型的动态规划问题，不难发现对于字符串 s 的某个位置 i 而言，我们只关心位置 i 自己能否形成独立 item 和位置 i 能够与上一位置（i-1）能否形成 item，二不关心 i-1 之前的位置。

有了以上分析，我们可以从前往后处理字符串 s ，使用一个数组记录以字符串 s 的每一位作为结尾的解码方案。即定义 f{i} 为考虑前 i 个字符的解码方案数。

对于字符串 s 的任意位置 i 而言，其存在三种情况：

. 只能由位置 i 的单独作为一个 item，设为 a，转移的前提是 a 的数值范围为[1, 9]，转移逻辑为 f[i] = f[i-1]。

. 只能由位置 i 的与前一位置（i-1）共同作为一个 item，设为 b，转移的前提是 b 的数值范围为 [10，26]，转移逻辑为 f[i] = f[i-2]。

. 位置 i 既能作为独立 item 也能与上一位置形成 item，转移逻辑为 f[i] = f[i-1] + f[i-2]。

因此我们有如下转移

$$\begin{cases} f[i] = f[i-1], 1 \leqslant a \leq 9 \\ f[i] = f[i-2], 10 \leqslant b \leqslant 26 \\ f[i] = f[i-1] + f[i-2], 1 \leqslant a \leq 9, 10 \leqslant b \leqslant 26 \end{cases}$$

其他细节：由于题目存在前导零，而前导零属于无效 item。可以进行特判，但个人习惯往只读串头部追加空格作为哨兵，追加空格既可以避免讨论前导零。也能使下标从 1 开始，简化 f[i-1] 等负数下标的判断。

代码如下：

```
1.  class Solution:
2.      def numDecodings(self, s:  str) -> int:
3.          n = len(s)
4.          s = ' ' + s
5.          f = [0] * (n + 1)
6.          f[0] = 1
7.          for i in range(1,n + 1):
8.              a = ord(s[i]) - ord('0')
9.              b = ( ord(s[i - 1]) - ord('0') ) * 10 + ord(s[i]) - ord('0')
10.             if 1 <= a <= 9:
11.                 f[i] = f[i - 1]
12.             if 10 <= b <= 26:
13.                 f[i] += f[i - 2]
14.         return f[n]
```

第十六篇 2024 年 5 月底， 好未来-算法实习面试题 5 道

16.1 讲一下xgb 与 lgb 的特点与区别

- xgboost 采用的是 level-wise 的分裂策略， 而 lightGBM 采用了 leaf-wise 的策略， 区别是 xgboost 对每一层所有节点做无差别分裂， 可能有些节点的增益非常小， 对结果影响不大， 但是xgboost 也进行了分裂， 带来了不必要的开销。 leaft-wise 的做法是在当前所有叶子节点中选择分裂收益最大的节点进行分裂， 如此递归进行， 很明显 leaf-wise 这种做法容易过拟合， 因为容易陷入比较高的深度中， 因此需要对最大深度做限制， 从而避免过拟合。
- lightgbm 使用了基于 histogram 的决策树算法， 这一点不同与 xgboost 中的 exact 算法， histogram算法在内存和计算代价上都有不小优势。
    - 内存： 直方图算法的内存消耗为(#data* #features * 1Bytes)(因为对特征分桶后只需保存特征离散化之后的值)， 而 xgboost 的 exact 算法内存消耗为： (2 * #data * #features* 4Bytes)， 因为xgboost 既要保存原始 feature 的值， 也要保存这个值的顺序索引， 这些值需要 32 位的浮点数来保存。
    - 计算 ： 预排序算法在选择好分裂特征计算分裂收益时需要遍历所有样本的特征值， 时间为(#data),而直方图算法只需要遍历桶就行了， 时间为(#bin)

16.2 讲一下xgb 的二阶泰勒展开对于 GBDT 有什么优势

- 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。

16.3 Leetcode91-解码方法

算法题： 字母 a-z对应数字 1-26， 给定一个数字序列， 求所有可能的解码总数， 例如 1261 对应解码为 1 2 6 1 ， 12 6 1， 1 26 1 总共为 3 种方式

https://leetcode-cn.com/problems/decode-ways/

参考答案：

参考答案：

这其实是一道字符串类型的动态规划问题， 不难发现对于字符串 s 的某个位置 i 而言 ， 我们只关心位置 i 自己能否形成独立 item 和 位置 i 能够与上一位置 （ i-1 ） 能否形成 item ， 二不关心 i-1 之前的位置。

有了以上分析， 我们可以从前往后处理字符串 s ， 使用一个数组记录以字符串 s 的每一位作为结尾的解码方案。即定义 f{i} 为考虑前 i 个字符的解码方案数。

对于字符串 s 的任意位置 i 而言 ， 其存在三种情况：

. 只能由位置 i 的单独作为一个 item ， 设为 a ， 转移的前提是 a 的数值范围为[1, 9] ， 转移逻辑为 f[i] = f[i-1]。

. 只能由位置 i 的与前一位置 （ i-1 ） 共同作为一个 item， 设为 b ， 转移的前提是 b 的数值范围为 [10 ， 26] ， 转移逻辑为 f[i] = f[i-2]。

. 位置 i 既能作为独立 item 也能与上一位置形成 item ， 转移逻辑为 f[i] = f[i-1] + f[i-2]。

因此我们有如下转移

$$\begin{cases} f[i] = f[i-1], 1 \leqslant a \leq 9 \\ f[i] = f[i-2], 10 \leqslant b \leqslant 26 \\ f[i] = f[i-1] + f[i-2], 1 \leqslant a \leq 9, 10 \leqslant b \leqslant 26 \end{cases}$$

其他细节： 由于题目存在前导零， 而前导零属于无效 item。可以进行特判， 但个人习惯往只读串头部追加空格作为哨兵， 追加空格既可以避免讨论前导零。也能使下标从 1 开始， 简化 f[i-1] 等负数下标的判断。

代码如下：

```
4.  class Solution:
5.      def numDecodings(self, s: str) -> int:
6.          n = len(s)
7.          s = ' ' + s
8.          f = [0] * (n + 1)
9.          f[0] = 1
10.         for i in range(1,n + 1):
11.             a = ord(s[i]) - ord('0')
12.             b = ( ord(s[i - 1]) - ord('0') ) * 10 + ord(s[i]) - ord('0')
13.             if 1 <= a <= 9:
14.                 f[i] = f[i - 1]
15.             if 10 <= b <= 26:
16.                 f[i] += f[i - 2]
17.         return f[n]
```

16.4 介绍 xgboost 和 gbdt 的区别

- 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。

16.5 算法题：给定一个 int 数字，将其转换为中文描述

例如， 1010 - > 一千零一十

https://blog.csdn.net/rensihui/article/details/94590075

参考代码：

```
import random
# number_to_chinese  讲整数转化为汉字
num_dict = { 0: "零", 1: "一", 2: "二", 3: "三", 4: "四",
             5: "五", 6: "六", 7: "七", 8: "八", 9: "九" }
unit_map = [ ["", "十", "百", "千"],       ["万", "十万", "百万", "千万"],
             ["亿", "十亿", "百亿", "千亿"], ["兆", "十兆", "百兆", "千兆"] ]
unit_step = ["万", "亿", "兆"]

class int_number2chinese():
    def __init__(self):
        self.result = ""
    def number_to_str_10000(self, data_str):
        # 一万以内的数转成大写
        res = []
        count = 0
        # 倒转
        str_rev = reversed(data_str)
        for i in str_rev:
            if i is not "0":
                count_cos = count // 4  # 行
                count_col = count % 4   # 列
                res.append(unit_map[count_cos][count_col])
                res.append(num_dict[int(i)])
                count += 1
```

```
            else:
                count += 1
                if not res: res.append("零")
                elif res[-1] is not "零":
                    res.append("零")
        res.reverse()
        # 去掉"一十零"这样整数的“零”
        if res[-1] is "零" and len(res) is not 1:
            res.pop()
        return "".join(res)
    def number_to_str(self, data):
        # 每 4 位数字进行一次分割
        len_data = len(data)
        count_cos = len_data // 4  # 行
        count_col = len_data-count_cos*4  # 列
        if count_col > 0: count_cos += 1
        res = ""
        for i in range(count_cos):
            if i==0:  data_in = data[-4:]
            elif i==count_cos-1 and count_col>0:
                data_in = data[:count_col]
            else:
                data_in = data[-(i+1)*4:-(i*4)]
            res_ = self.number_to_str_10000(data_in)
            res = res_ + unit_map[i][0] + res
        return res
    def decimal_chinese(self, data):
        assert type(data) == int, "请输入整数"
        data_str = str(data)
        res = self.number_to_str(data_str)
        return res
if __name__=="__main__":
    print(n2c.decimal_chinese(23))
```

第十七篇 2024 年 6 月 7 日-6 月 8 日， TP-LINK 提前批（图像算法岗）面试题 6 道

17.1 adaboost 和随机森林有什么区别

- 相同点：
  a. 二者都是 bootsrap 自助法选取样本。
  b. 二者都是要训练很多课决策树。
- 不同点：
  a. Adaboost 后面树的训练， 其在变量抽样选取的时候， 对于上一颗树分错的样本， 抽中的概率会加大。
  b. 随机森林在训练每一棵树的时候， 随机挑选了部分变量作为拆分变量， 而不是所有的变量都去作为拆分变量。
  c. 预测新数据时， adaboost 中所有的树加权投票来决定因变量的预测值， 每棵树的权重和错误率有关； 随机森林按照所有树中少数服从多数树的分类值来决定因变量的预测值。

17.2 LBP 特征和 SIFT 的特征的意义

- SIFT： Scale-Invariant Feature Taransform 尺度不变特征变换

SIFT 算法分解为如下四步：

1. 尺度空间极值检测： 搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。

2. 关键点定位： 在每个候选的位置上， 通过一个拟合精细的模型确定位置和尺度。关键点的选择依据于它们的稳定程度。

3. 方向确定： 基于图像局部的梯度方向， 分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换， 从而提供对于这些变换的不变性。旋转不变性，利用梯度的方法求取局部结构的稳定性。

4. 关键点描述（128 维）：在每个关键点周围的邻域内， 在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示， 这种表示允许比较大的局部形状的变形和光照变化。描述子是关键点领域高斯图像梯度统计结果的一种表示， 通过对关键点周围图像区域分块， 计算块内梯度直方图， 生成具有独特性的向量。

- LBP： Local Binary Pattern 局部二值模式

对灰度图像进行二值化， 经典 LBP 算法窗口是 3*3 的正方形窗口， 以窗口中心像素为阈值， 相邻 8 领域像素灰度和中心像素值比较， 若大于这标 1 或者则标 0， 则中心点可以由 8 位二进制来表示， 共有 2 ^8 = 256 种 LBP 值， 中心像素的 LBP 值反映了该像素周围区域的纹理信息。

由于 LBP 无尺度和旋转不变性， 后续有研究针对这两点进行改进。如圆形 LBP 算子达到旋转不变性的要求。

17.3 颜色校正、 白平衡的过程

- 此处举例 Local Color Correction 局部颜色较正算法的步骤：
  - 根据输入图像计算出掩膜图像；
  - 结合输入图像和掩模图像计算出最终结果

掩膜图像一般根据彩色图像各个通道的图像灰度值获得。假设 RGB 图像各个通道的像素灰度值为 R，G， B， 则掩膜图像可以表示为 I = (R + G + B) / 3I=(R+G+B)/3， 之后对掩膜图像进行高斯滤波：$M(x,y) = (Gaussian * (255 - I))(x,y)$, 高斯滤波时， 选取较大值进行滤波， 以保证对比度不会沿着边缘方向过度减小。上述的输出结果表明： 图像哪部分需要提亮， 哪部分需要减暗。最后输出图像为：output(z, y)= $255(\frac{Input(x,y)}{255})^{2^{\frac{128-M(x,y)}{128}}}$ ，如果掩膜图像大于 128， 将得到一个大于 1 的指数， 并对图像该点的亮度移植， 反之增加亮度。如果等于 128， 则不改变该像素点亮度。

代码样例：

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

Mat LCC(const Mat& src)
{
    int rows = src.rows;
    int cols = src.cols;
    int** I = new int* [rows];
    for (int i = 0; i < rows; i++) {
        I[i] = new int[cols];
    }
    int** inv_I;
    inv_I = new int* [rows];
    for (int i = 0; i < rows; i++) {
        inv_I[i] = new int[cols];
    }
    # 掩膜
    Mat Mast(rows, cols, CV_8UC1);
    for (int i = 0; i < rows; i++) {
        uchar* data = Mast.ptr<uchar>(i);//? ? ?
        for (int j = 0; j < cols; j++) {
            //获取图像的掩膜
            I[i][j] = (src.at<Vec3b>(i, j)[0] + src.at<Vec3b>(i, j)[1] + src.
at<Vec3b>(i, j)[2]) / 3.0;
            inv_I[i][j] = 255;
            *data = inv_I[i][j] - I[i][j];
            data++;
        }
```

```
    }
    //高斯滤波
    GaussianBlur(Mast, Mast, Size(41, 41), BORDER_DEFAULT);

    Mat dst(rows, cols, CV_8UC3);
    for (int i = 0; i < rows; i++) {
        uchar* data = Mast.ptr<uchar>(i);
        for (int j = 0; j < cols; j++) {
            for (int k = 0; k < 3; k++) {
                float Exp = pow(2, (128 - data[j]) / 128.0);
                //输出结果
                int value = int(255 * pow(src.at<Vec3b>(i, j)[k] / 255.0, Exp));
                dst.at<Vec3b>(i, j)[k] = value;
            }
        }
    }
    return dst;
}

int main()
{
    Mat img, result;
    img = imread("july.jpg", IMREAD_COLOR);

    namedWindow("ResultImg", WINDOW_AUTOSIZE);
    namedWindow("OriginImg", WINDOW_AUTOSIZE);

    result = LCC(img);
    if (result.empty())
    {
        cout << "Error! THE IMAGE IS EMPTY.." << endl;
        return -1;
    }
    else
    {
        imshow("OriginImg", img);
        imshow("ResultImg", result);
        imwrite("july_lcc.jpg", result);
    }
    waitKey(0);
    return 0;
}
```

- 白平衡：

人的视觉系统具有颜色恒常性，能从变化的光照环境和成像条件下获取物体表面颜色的不变特性，但成像设备不具有这样的调节功能，不同的光照环境会导致采集的图像颜色与真实颜色存在一定程度的偏差，需要选择合适的颜色平衡（校正）算法，消除光照环境对颜色显现的影响。灰度世界算法是最常用平衡算法。

灰度世界算法以灰度世界假设为基础，该假设认为：对于一幅有着大量色彩变化的图像，RGB 三个分量的平均值趋于同一灰度值 Gray。从物理意义上讲，灰色世界法假设自然界景物对于光线的平均反射的均值在总体上是个定值，这个定值近似地为“灰色”。颜色平衡算法将这一假设强制应用于待处理图像，可以从图像中消除环境光的影响，获得原始场景图像。

原始图像　　颜色校正之后的图像

注意左图中较亮（6 周年附近）和较暗（右下角盆栽）区域和右图中对应区域的区别；

算法执行步骤：

- 一般有两种方法确定 Gray: 要么取固定值（如最亮灰度值的一半， 八位显示的话即为 128 ）,要么通过计算图像， RGB 三通道的均值$\bar{R}, \bar{G}, \bar{B},$ ，取$Gr^-ay = \frac{\bar{R}+\bar{G}+\bar{B}}{3}$
- 计算 RGB 三通道的增益系数:

$$k_r = \frac{\text{Gray}}{\bar{R}}; k_g = \frac{\text{Gray}}{\bar{G}}; k_b = \frac{\text{Gray}^-}{\bar{B}};$$

- 根据 Von Kries 对角模型， 对于图像中的每个像素 C， 调整其 RGB 分量：

$$\begin{cases} C(R') = C(R) * k_r \\ C(G') = C(G) * k_g \\ C(B') = C(B) * k_b \end{cases}$$

可以 colorcorrect 库实现：

```
1.  from PIL import Image
2.  import colorcorrect.algorithm as cca
```

```
3.  from colorcorrect.util import from_pil, to_pil
4.
5.  img = Image.open("street.png")
6.  to_pil(cca.grey_world(from_pil(img))).show()
```

原图：

使用灰度世界算法进行白平衡变换后：

17.4 HOG 特征的意义

Hog: Histograms of oriented gradients

HOG 通过计算局部图像提取的方向信息统计值来统计图像的梯度特征， 它跟 EOH、 SIFT 及 shape contexts 有诸多相似之处， 但是它有明显的不同之处： HOG 特征描述子是在一个网格密集、大小统一的单元上进行计算， 而且为了提高性能， 它还采用了局部对比度归一化思想。它的出现， 使得目标检测技术在静态图像的人物检测、车辆检测等方向得到大量应用。

在传统目标检测中， HOG 可以称得上是经典中的经典， 它的 HOG+SVM+归一化思想对后面的研究产生深远的影响， HOG 的出现， 奠定了 2005 之后的传统目标检测的基调和方向。

17.5 决策树有哪些种类

常见的决策树算法主要有：

- ID3、C4.5 和 CART 树；
- 随机森林、Adaboost、GBDT
- Xgboost 和 LightGBM

17.6 SVM 的 KKT 条件是啥

一般地， 一个最优化数学模型能够表示成下列标准形式：

$$\begin{aligned} \min. \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & h_j(\mathbf{x}) = 0, j = 1, \ldots, p, \\ & g_k(\mathbf{x}) \leq 0, k = 1, \ldots, q, \\ & \mathbf{x} \in \mathbf{X} \subset \Re^n \end{aligned}$$

其中， f(x)是需要最小化的函数， h(x)是等式约束， g(x)是不等式约束， p 和 q 分别为等式约束和不等式约束的数量。

同时， 得明白以下两点：

- 凸优化的概念： 为一凸集 ， 为一凸函数。凸优化就是要找出一点 ， 使得每一满足 。
- KKT 条件的意义： 它是一个非线性规划（ Nonlinear Programming ）问题能有最优化解法的必要和充分条件

而 KKT 条件就是指上面最优化数学模型的标准形式中的最小点 x* 必须满足下面的条件：

$$1.\ h_j(\mathbf{x}_*) = 0, j = 1, \ldots, p, g_k(\mathbf{x}_*) \leq 0, k = 1, \ldots, q,$$

$$2.\quad \nabla f(\mathbf{x}_*) + \sum_{j=1}^{p} \lambda_j \nabla h_j(\mathbf{x}_*) + \sum_{k=1}^{q} \mu_k \nabla g_k(\mathbf{x}_*) = \mathbf{0},$$

$$\lambda_j \neq 0, \mu_k \geq 0, \mu_k g_k(\mathbf{x}_*) = 0$$

经过论证， 我们这里的问题是满足 KKT 条件的（首先已经满足 Slater 条件， 再者 f 和 gi 也都是可微的， 即 L 对 w 和 b 都可导） ， 因此现在我们便转化为求解第二个问题。

也就是说， 原始问题通过满足 KKT 条件， 已经转化成了对偶问题。而求解这个对偶学习问题， 分为 3 个步骤： 首先要让 L(w， b， a) 关于 w 和 b 最小化， 然后求对α的极大， 最后利用 SMO 算法求解对偶问题中的拉格朗日乘子。

第十八篇 2024 年 6 月 6 日--6 月 16 日，拼多多算法面试题 8 道

18.1 GBDT 原理

参考答案：

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。

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。

更多请看七月在线题库里的这题：https://www.julyedu.com/questions/interview-detail?quesId=2591&cate=%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0&kp_id=23

18.2 连续数组最大和（DP）

参考答案：

本题为剑指 offer42 题，和 Leetcode 第 53 题相同。

题目描述给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

解析：本题在面试时要求使用动态规划的方法，因此需要找到对应的状态定义（子问题）和状态转移方程（子问题之间的关系）

状态定义：dp[i]：表示以 nums[i] 结尾的连续子数组的最大和。

状态转移方程：

分两种情况考虑：dp[i-1] <= 0 和 dp[i-1] > 0

如果 dp[i - 1] > 0，那么可以把 nums[i] 直接接在 dp[i - 1] 表示的那个数组的后面，得到和更大的连续子数组；

如果 dp[i - 1] <= 0，那么 nums[i] 加上前面的数 dp[i - 1] 以后值不会变大。于是 dp[i]「另起炉灶」，此时单独的一个 nums[i] 的值，就是 dp[i]。

得到状态转移方程如下：

$$dp[i] = \begin{cases} dp[i-1] + nums[i], & if \quad dp[i-1] > 0 \\ nums[i], & if \quad dp[i-1] \leq 0 \end{cases}$$

整理可得：

$$dp[i] = \max\{nums[i],\ dp[i-1] + nums[i]\}$$

代码如下：

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        size = len(nums)
        pre = 0
        res = nums[0]
        for i in range(size):
            pre = max(nums[i], pre + nums[i])
            res = max(res, pre)
        return res
```

18.3 ROC 含义、AUC 含义

参考答案：

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

18.4 XGB 原理，正则化操作

参考答案：

XGB 原理：

首先需要说一说 GBDT，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。

xgboost 使用了如下的正则化项：

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

γ和λ，这是xgboost 自己定义的，在使用xgboost 时，可以设定它们的值，显然，γ越大，表示越希望获得结构简单的树，因为此时对较多叶子节点的树的惩罚越大。λ越大也是越希望获得结构简单的树。

18.5 L1 与 L2 区别，效果差异

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。

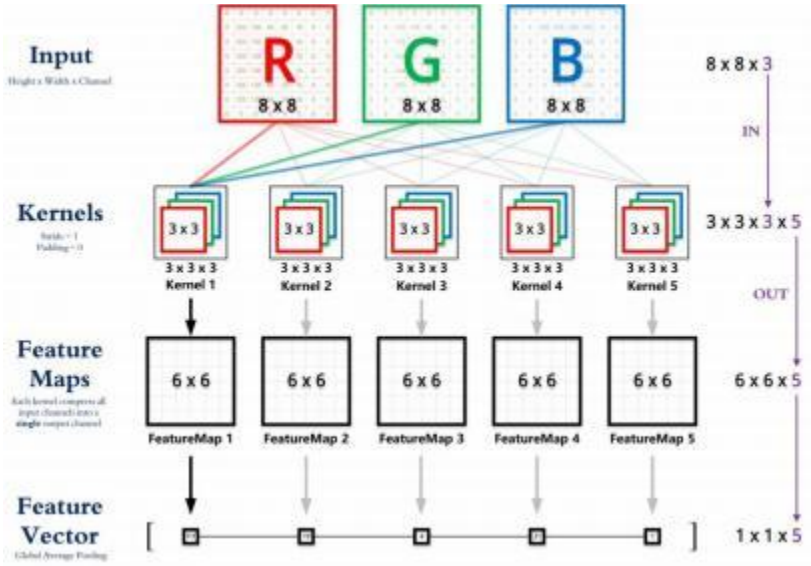
L1 和 L2 的差别，为什么一个让绝对值最小，一个让平方最小，会有那么大的差别呢？看导数一个是 1 一个是 w 便知，在靠进零附近，L1 以匀速下降到零，而 L2 则完全停下来了. 这说明 L1 是将不重要的特征(或者说，重要性不在一个数量级上)尽快剔除，L2 则是把特征贡献尽量压缩最小但不至于为零. 两者一起作用，就是把重要性在一个数量级(重要性最高的)的那些特征一起平等共事(简言之，不养闲人也不要超人)。

更多请看七月在线题库里的这题：https://www.julyedu.com/questions/interview-detail?kp_id=23&cate=%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0&quesId=995

18.6 卷积计算过程

参考答案：

普通卷积计算：



(图像来源 ： https：//zhuanlan.zhihu.com/p/29119239)

定义： 输入图像大小： H in*win*c in ； 输出图像大小： H,*w,*c。

卷积核大小： k*k*DK*c,其中： DK为卷积核的厚度等于cin,C_o为卷积核的个数, 也是输出特征图的通道数。

标准卷积计算过程如上图所示， 其中

输出图像大小计算公式为：

$$H_o = W_o = [(W - F + 2 * P)/S] + 1 = [8 - 3 + 2 * 0]/1] + 1 = 6$$

参数量(不考虑偏置项)：

$$Para = k * k * C_{in} * C_o$$

参数量即可训练的参数数量， 卷积计算过程中参数共享， 而且不同通道的卷积核参数也不一样， 所以每个卷积核的参数量为k米k米c， 共有C_o个卷积核。

计算量：

$$Cal = [k * k * C_{in} + (k * k - 1) * C_{in} + (C_{in} - 1)] * H_o * W_o * C_o = (2 * k * k * C_{in} - 1) * H_o * W_o * C_o$$

计算量包括加减乘除操作次数， 其中 1 个卷积核 1 次卷积的计算量为：

k*k*c in十(k*k一1)*cn十(c in一1)

第 1 部分是卷积核在各通道进行 1 次乘法操作，

第 2 部分是乘法之后卷积核内部的加合操作，

第 3 部分是各通道间的加合操作； 一个卷积核在一个输出通道上共进行H,*WO次卷积操作， 共有C_o个卷积核。

18.7 session 与 graph 区别

参考答案：

graph 即 tf.Graph()， session 即 tf.Session()，是两个完全独立的概念。一个 graph 可以供多个 session 使用，而一个 session 不一定需要使用 graph 的全部，可以只使用其中的一部分。

graph 定义了计算方式，是一些加减乘除等运算的组合。它本身不会进行任何计算，也不保存任何中间计算结果。

session 用来运行一个 graph，或者运行 graph 的一部分。它类似于一个执行者，给 graph 灌入输入数据，得到输出，并保存中间的计算结果。同时它也给 graph 分配计算资源（如内存、显卡等）。

18.8 最长不重复连续字符长度（DP）

参考答案：

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。

```
1.  class Solution:
2.      def lengthOfLongestSubstring(self, s: str) -> int:
3.          record = {}
4.          start,res = 0,0
5.          for end in range(len(s)):
6.              if s[end] in record:
7.                  start = max(start, record[s[end]] + 1)
8.              record[s[end]] = end
9.              res = max(res, end - start + 1)
10.         return res
```

时间复杂度： O(n)

空间复杂度： O(1)

第十九篇 6.28-7.04 TP-LINK 提前批网络安全算法工程师面试题 3 题

19.1 TCP 和 UDP 的特点

TCP： 面向连接

TCP 面向连接通信,所以握手过程会消耗资源,过程为可靠连接,不会丢失数据,适合大数据量交换 ； “面向连接”就是在正式通信前必须要与对方建立起连接。

TCP 协议能为应用程序提供可靠的通信连接， 使一台计算机发出的字节流无差错地发往网络上的其他计算机， 对可靠性要求高的数据通信系统往往使用 TCP 协议传输数据。

TCP 支持的应用协议： Telnet(远程登录)、FTP(文件传输协议)、SMTP(简单邮件传输协议)。

UDP： 面向非连接的

UDP 面向非可靠连接,会丢包,没有校验,速度快,无须握手过程； "面向非连接”就是在正式通信前不必与对方先建立连接， 不管对方状态就直接发送。

UDP 支持的应用协议： NFS(网络文件系统)、SNMP(简单网络管理系统)、DNS(主域名称系统)、TFTP(通用文件传输协议)等。

总结：

TCP： 面向连接、传输可靠(保证数据正确性,保证数据顺序)、用于传输大量数据(流模式)、速度慢， 建立连接需要开销较多(时间， 系统资源)。

UDP： 面向非连接、传输不可靠、用于传输少量数据(数据包模式)、速度快。

19.2 智力推理题：25 匹马 5 个跑道

25 匹马 5 个跑道,每次只能跑 5 匹,至少需要多少次才能选出最快的前 3 匹?

将马分成 A、B、C、D、E 五组。

第 1-5 次比赛： 各组分别进行比赛， 决出各组名次,取每组前三名

A1、A2、A3

B1、B2、B3

C1、C2、C3

D1、D2、D3

E1、 E2、E3

第 6 次比赛： A1、B1、C1、D1、E1， 即每组的第一名进行比赛；

假设得到的排名结果是 A1、B1、C1、D1、E1； 其中 A1 是跑的最快的， 那么 A 组 A2、A3， B 组 B1、B2 以及 C 组的 C1 都是有机会进入前三的， C2 没有希望冲进前 3 了， 因为 C1 是比赛的名次已经是第 3 名了， 此外 D 组 E 组都没有希望进入前 3。现在已经知道 A1 肯定是第 1 名， 剩下 A2、A3、B1、B2、C1 是有希望冲进前三的， 那么让他们再进行一场比赛即可；

第 7 次比赛： A2、A3、B1、B2、C1 比赛求出第 2 ， 第 3 即可。
所以公共需要 7 厂比赛， 就可以选出最快的前 3 匹马；

19.3 用 rand5 实现 rand7

rand5(5)： 等概率生成整数[1, 2, 3, 4, 5]
rand5(7): 等概率生成整数[1, 2, 3, 4, 5, 6, 7]
思路：
用(rand5(5) - 1)构造等概率整数数组： [0, 1, 2 , 3, 4],
用(rand5(5) - 1)*5 构造整数数组： [0, 5, 10, 15, 20],
上面的两个整数组可以构造等概率的新数组： [0, 1, 2, 3, 4, 5, 6, 7, ... , 24];
(如果第 2 个数组选择 2 倍或者 3 倍， 4 倍则无法构造新的等概率数组)
选择新数组[0, 1, 2, 3 .... , 20]21 个元组即可构造等概率的数组[1, 2, 3, 4, 5, 6, 7]
参考代码：

```
import random

def rand5():
    return random.randint(1, 5)

def rand7():
    while True:
        tmp_num = (rand5() - 1)*5 + (rand5() - 1)
        if tmp_num <= 20:
            break
    return tmp_num % 7 + 1

# 随机生成 10000 个 1~7 的整数，验证 0~7 各个数字生成的概率；
ls = []
for _ in range(10000):
    ls.append(rand7())
for i in range(1, 8):
    print(ls.count(i)/len(ls), end=" ")
# out：0.144 0.1459 0.14 0.1438 0.1444 0.1373 0.1446
# 观察可知 0~7 各个数字生成的概率是等价的
```

第二十篇 2024 年 6 月 28 日-vivo 推荐算法工程师一面试题 5 题

20.1 介绍下什么是 Word2vec

在介绍 Word2Vec 之前需要先理解 Word Embedding，它是将无法直接计算的、非结构化的单词转化为可计算的、结构化的数据-向量；

那么什么是 Word2Vec 呢？

Word2vec 是 Word Embedding 的方法之一。他是 2013 年由谷歌的 Mikolov 提出了一套新的词嵌入方法；

Word2Vec 有两种训练模式： CBOW(Continuous Bag-of-Words Model)和 Skip-gram (Continuous Skip-gram Model)；

CBOW： 根据上下文信息预测当前值（单词）

Skip-gram ： 根据当前值（单词）预测上下文信息

更多 Word2Vec 的讲解请参考七月题库：
请说说 word2vec 的简要理解 - AI 笔试面试题库： https://www.julyedu.com/questions/interview-detail?kp_id=30&cate=NLP&quesId=2966
如何通俗理解 Word2vec - AI 笔试面试题库： https://www.julyedu.com/questions/interview-detail?kp_id=30&cate=NLP&quesId=2761

20.2 Word2vec 负采样如何实现？

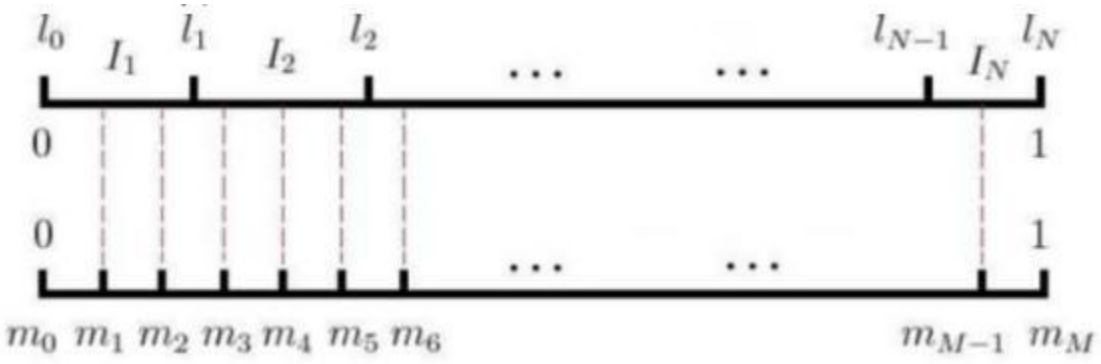
现在我们看下 Word2vec 如何通过 Negative Sampling 负采样方法得到 neg 个负例； 如果词汇表的大小为 V,那么我们就将一段长度为 1 的线段分成 V 份， 每份对应词汇表中的一个词。当然每个词对应的线段长度是不一样的， 高频词对应的线段长， 低频词对应的线段短。每个词 w 的线段长度由下式决定：

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

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。

20.3 Widedeep 为什么要分 wide deep，好处？

Wide & Deep 设计了一种融合浅层(wide)模型和深层(deep)模型进行联合训练的框架。

Wide 部分： 通过线性模型 + 特征交叉， 使模型具有“记忆能力（ Memorization ）” ，通常非常有效、可解释较强。但其”泛化能力（ Generalization ）“需要更多的人工特征工程。

Deep 部分： 只需要少量的特征工程， 深度神经网络（ DNN ）通过对稀疏特征进行学习， 可以较好地推广到训练样本中未出现过的特征组合， 使模型具有“泛化能力”。但当 user-item 交互矩阵稀疏且高阶时， 例如特定偏好的用户和小众产品， 容易出现过拟合， 导致推荐的 item 相关性差。

好处： 结合 Wide 与 Deep 模型各自的优势： Memorization 与 Generalization， Wide & Deep 模型比各自模型更加有效。

20.4 Wide&Deep 的 Deep 侧具体实现

deep 部分是 Embedding+MLP 前馈神经网络。高维离散特征会首先被转为低维稠密向量， 通常被称为 Embedding vectors， Embedding 被随机初始化， 并根据最终的 loss 来反向训练更新。隐藏层为全连接网络：

$$a^{(l+1)} = f\left(W^{(l)}a^{(l)} + b^{(l)}\right)$$

其中， a,b,w,f分别为第 l 层的输入、偏置项、参数项与激活函数。

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。深度神经网络通过特征的多次自动组合， 可以深度发掘数据中潜在的模式， 即使是非常稀疏的特征向量输入， 也能得到较稳定平滑的推荐概率。

20.5 DeepFM 为了解决什么问题？

在处理 CTR 预估问题中， 传统的方法有一个共同的缺点： 对于低阶的组合特征， 学习到的比较少； 但是低阶特征对于 CTR 也非常重要， 于是 Google 为了同时学习低阶和高阶组合特征， 提出了 Wide&Deep 模型： 混合了一个 线性模型（ Wide part ）和 Deep 模型 (Deep part)； 这两部分模型需要不同的输入， 其中 Wide part 部分的输入仍然依赖人工特征工程；

此时模型存在两个问题：

偏向于提取低阶或者高阶的组合特征， 不能同时提取这两种类型的特征；

需要专业的领域知识来做特征工程；

DeepFM 在 Wide&Deep 的基础上进行改进， 成功解决了上述这两个问题， 并做了一些优化；

优点如下：

不需要预训练 FM 得到隐向量；

不需要人工特征工程；

能同时学习低阶和高阶的组合特征；

FM 模块和 Deep 模块共享 Feature Embedding 部分， 可以更快、更精确的训练；