链滴

4 月 22 日 -5 月 7 日腾讯 nlp 算法实习面试题 14 道

作者：julyedu
原文链接：https://ld246.com/article/1623062353029
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

1.介绍下word2vec，有哪两种实现，可以用什么方法提性能，分层softmax介绍一下原理，负采样怎么做，，负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。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θ，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。

2.介绍LSTM和GRU有什么不同?

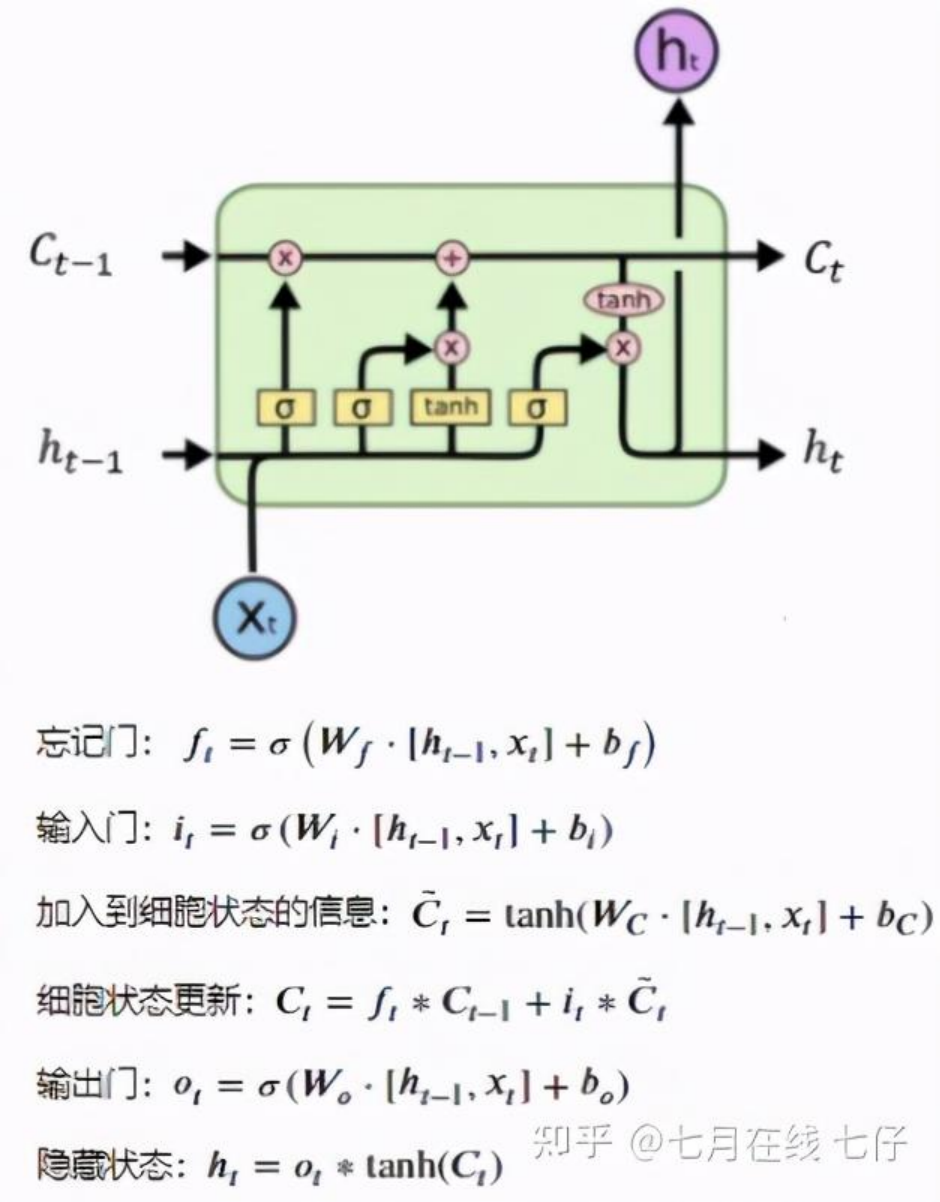
（1）LSTM和GRU的性能在很多任务上不分伯仲；

（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。

3.介绍一下LSTM单元里面的运算过程。



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：

$$P_{total} = P_1 + P_2 + ... + P_N = e^{S_1} + e^{S_2} + ... + e^{S_N}$$

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

知乎 @七月在线 七仔

原文链接：4 月 22 日 -5 月 7 日腾讯 nlp 算法实习面试题 14 道

5.transformer介绍一下原理，transformer为什么可以行，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Decode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。

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

知乎 @七月有雪 七仔

● 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。

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

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，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。

6 BERT介绍一下原理，怎么用BERT计算文本相似度，有两种计算方法？

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。

7 transformer里自注意力机制的计算过程，为什么要进缩放，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。

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

知乎 @七月有雪七仔

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.介绍下bert位置编码和transformer的区别，哪个好，什么？

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 encodig。

原版的Transformer中，谷歌对learned position embedding和sinusoidal position encoding进行对比实验，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，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。

9.sigmoid函数的缺点，为什么会产生梯度消失？不是以为中心的话，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；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，这较大地增加训练时间。

10.Layer Normalization和Batch Normalization 的别

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。Bat h 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BN明显。

11.Leetcode 8. 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

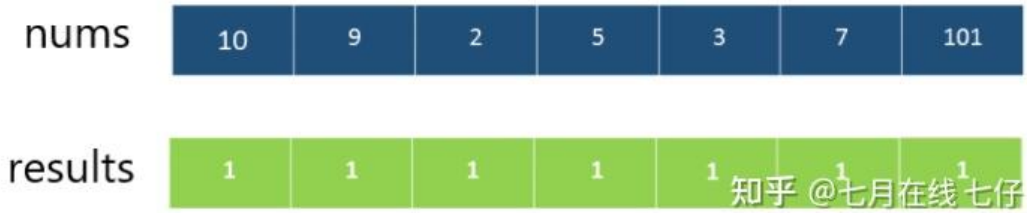
代码如下：

```
class Solution:

def myAtoi(self, s: str) -> int:

return max(min(int(*re.findall('^[\+\-]?\d+', s.lstrip())), 2**31 - 1), -2**31)
```

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；

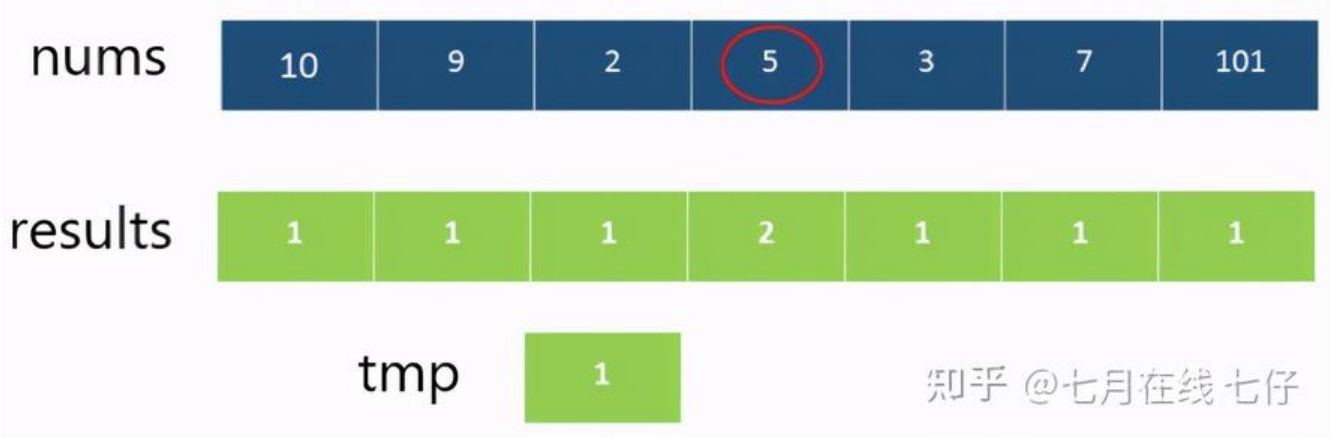
因为要找处递增序列，所以我们只需要找出nums[i]>nums[j]（其中j < i）的数，并将对应的results[j]保存在tmp临时列表中，然后找出最长的那个序列将nums[i]附加其后面；

示例：假设nums[i] = 5, i = 3

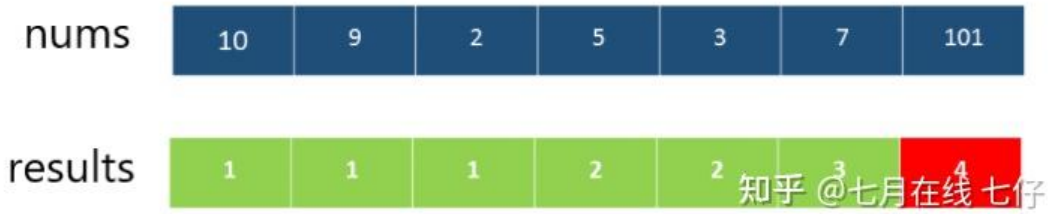
更新results[i]时只需要考虑前面小于results[i]的项，所以results[0-2]均为1；

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：

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        # 特殊情况判断
        if len(nums) == 0: return 0

        results = [1]*len(nums)
        for i in range(1, len(nums)):
            tmp = []
            for j in range(i):
                if nums[i] > nums[j]:
                    tmp.append(results[j])
            if tmp: results[i] = max(tmp) + 1
        return max(results)
```

13.全排列

解析1：

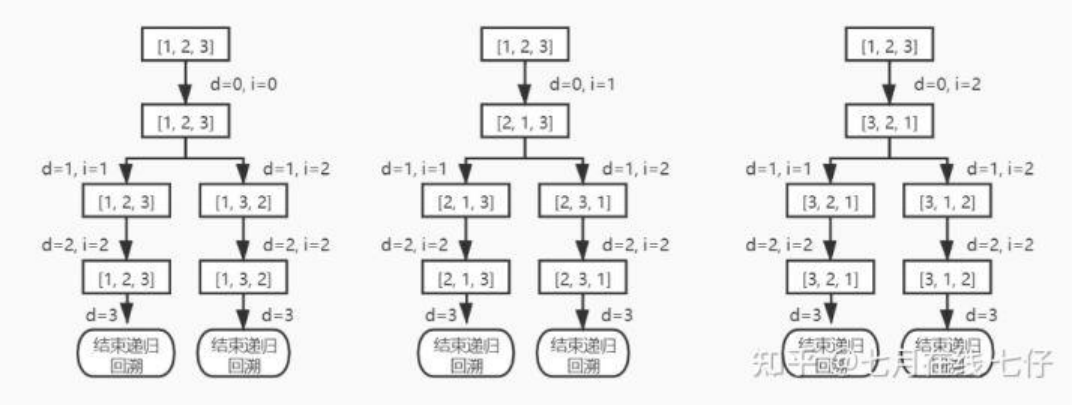
定义一个长度为len(nums)的空表output，从左往右一次填入nums中的数字，并且每个数只能使用次。可以枚举所有困难，从左往右每一个位置都一次填入一个数，用used表记录nums[i]是否已填入otput中，如果nums[i]木有填入output中则填入并标记，当output的长度为len(nums)时，得到一个足条件的结果，将output存入最终结果。然后将output回溯到未添加nums[i]状态，used回溯为未标nums[i]状态，继续下一次尝试。

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        # 特殊判断
        if len(nums) == 0: return nums
        result = []
        depth, n, output, used = 0, len(nums), [], [False]*len(nums)
        self.backtrack(depth, n, output, used, nums, result)
        return result


    def backtrack(self, depth, n, output, used, nums, result):
        # 递归终止条件：
        if depth == n:
            result.append(output[:])
        else:
            # 遍历数组中的每一个数
            for i in range(0, n):
                # 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]
```

14.合并两个有序数组并去重

```python
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+
```