关于束搜索（Beam Search）相关问题的探讨

**研究的问题**

搜索算法中的束搜索及其相关问题。

**研究背景**

在seq2seq的工作场景中，模型的输入输出均为序列（Sequence），其中最为典型的案例便是机器翻译。机器翻译的过程可以简单概述为，某种语言的句子序列通过编码器生成一个向量作为其代表，传递给解码器作为解码器的输入向量，输出另一种语言的句子序列。

语言模型对比于机器翻译模型的解码器，我们可以发现，除开语言模型的预输入向量，而机器翻译模型的解码器的输入是编码器输出的向量以外，几乎完全相同，因此机器翻译模型又叫条件语言模型（condition language model）。

语言模型告诉我们的是一个句子的概率，机器翻译解码器的作用在编码器传递的向量的基础上，随机抽样得到多种翻译，再通过某种算法找出句子概率最大的一个，即最合适的翻译结果。

在此间，我们就需要用到搜索的算法进行语句的处理（翻译）。

**研究意义**

借由不断优化的搜索算法找出某些问题的最优解，提高搜索效率。

**搜索的基本概念**

依靠经验，利用已有知识，根据问题的实际情况，不断寻找可利用知识，从而构造一条代价最小的推理路线，使问题得以解决的过程称为搜索，其主要包含两个维度：找到从初始事实到问题最终答案的一条推理路径与找到的这条路径在时间和空间上复杂度最小。

搜索适用于不良结构或非结构化问题、难以获得求解所需的全部信息以及没有现成的算法可供求解使用等具体情况。

依据是否使用启发式信息，搜索可以分为盲目搜索（Blind Search，又名无信息搜索）、启发式搜索（Heuristic Search）两类。

**规定**

在任意时间步，解码器输出的概率取决于时间步之前的输出子序列和对输入序列的信息进行编码得到的上下文变量c。为了量化计算代价，用Y表示输出词表，其中包含“<eos>”， 所以这个词汇集合的基数就是词表的大小。我们还将输出序列的最大词元数指定为。因此，我们的目标是从所有个可能的输出序列中寻找理想的输出。当然，对于所有输出序列，在“<eos>”之后的部分（非本句） 将在实际输出中丢弃。

**贪心搜索（Greedy Algorithm）与穷举搜索（Exhaustive Search）**

贪心算法

对于输出序列的每一时间步， 我们都将基于贪心搜索从Y中找到具有最高条件概率的词元，即：

公式 1

一旦输出序列包含了“<eos>”或者达到其最大长度，则输出完成。



图 1 贪婪算法下所得序列

如图1中，假设输出中有四个词元“A”、“B”、“C”和“<eos>”。 每个时间步下的四个数字分别表示在该时间步生成“A”、“B”、“C”和“<eos>”的条件概率。在每个时间步，贪心搜索选择具有最高条件概率的词元。 因此，将在图1中预测输出序列“A”、“B”、“C”和“<eos>”。这个输出序列的条件概率是0.5\*0.4\*0.4\*0.6=0.048。

然而在现实中，最优序列（optimal sequence）应该是最大化值的输出序列，这是基于输入序列生成输出序列的条件概率。然而，贪心搜索无法保证得到最优序列。



图 2 某种可能的现实情况下的最优序列

图2中的另一个例子阐述了这个问题。与图1不同，在时间步2中，我们选择图2中的词元“C”，它具有第二高的条件概率。由于时间步3所基于的时间步1和2处的输出子序列已从图1中的“A”和“B”改变为图2中的“A”和“C”，因此时间步处的每个词元的条件概率也在图2中改变。假设我们在时间步选择词元“B”，于是当前的时间步基于前三个时间步的输出子序列“A”、“C”和“B”为条件，这与图1中的“A”、“B”和“C”不同。因此，在图2中的时间步生成每个词元的条件概率也不同于图1中的条件概率。结果，图2中的输出序列“A”、“C”、“B”和“<eos>”的条件概率为0.5\*0.3\*0.6\*0.6=0.054，这大于图1中的贪心搜索的条件概率。 这个例子说明：贪心搜索获得的输出序列 “A”、“B”、“C”和“<eos>” 不一定是最佳序列。

**穷举搜索**

如果目标是获得最优序列，我们可以考虑使用穷举搜索（Exhaustive Search）：穷举地列举所有可能的输出序列及其条件概率，然后计算输出条件概率最高的一个。

虽然我们可以使用穷举搜索来获得最优序列，但其计算量可能高的惊人。例如，当和时，我们需要评估序列，现有的计算机几乎不可能计算它。然而，贪心搜索的计算量要显著地小于穷举搜索。例如，当和时，我们只需要评估个序列。

针对上述两种算法的天然劣势，我们考虑设计一种更加高效且具有更低错误概率的算法予以替代之。

**束搜索Beam Search（Beam Search集束搜索）**

束搜索（Beam Search）是一种启发式搜索算法，通常用在图的解空间比较大的情况下。为了减少搜索所占用的空间和时间，束搜索在每一步深度扩展的时候，剪掉一些质量比较差的结点，保留下一些质量较高的结点。这样减少了空间消耗，并提高了时间效率，但缺点就是有可能存在潜在的最佳方案被丢弃，因此Beam Search算法是不完全的，一般用于解空间较大的系统中。

束搜索于20世纪70年代中期首先被应用于人工智能领域,1976年Lowerre 在其称为HARPY的语音识别系统中第一次使用了束搜索方法,他的目标是并行地搜索几个潜在的最优决策路径以减少回溯,并快速地获得一个解。

更准确的说，集束搜索是一种类似于迭代最佳改进（Iterative Best Improvement）的方法。它使用启发式方法估计k 个最好的路径,仅从这k 个路径出发向下搜索,即每一层只有满意的结点会被保留,其它的结点则被永久抛弃,从而比分枝定界法能大大节省运行时间。束搜索于20世纪70年代中期首先被应用于人工智能领域,1976年Lowerre 在其称为HARPY的语音识别系统中第一次使用了束搜索方法,他的目标是并行地搜索几个潜在的最优决策路径以减少回溯,并快速地获得一个解。



图 3 束搜索过程

束搜索（Beam Search）是贪心搜索的一个改进版本。 它有一个超参数，名为束宽（Beam Size）k。在时间步1，我们选择具有最高条件概率的k个词元。 这k个词元将分别是k个候选输出序列的第一个词元。 在随后的每个时间步，基于上一时间步的k个候选输出序列， 我们将继续从k|Y|个可能的选择中挑出具有最高条件概率的k个候选输出序列。

上图（图3）演示了束搜索的过程。假设输出的词表只包含五个元素：Y={A,B,C,D,E}， 其中有一个是“<eos>”。设置束宽为2，输出序列的最大长度为3。在时间步1，假设具有最高条件概率P(y1∣c)的词元是A和C。在时间步2，我们计算所有y2∈Y为：

P(A,y2∣c)=P(A∣c)P(y2∣A,c),

P(C,y2∣c)=P(C∣c)P(y2∣C,c),

公式 2

从这十个值中选择最大的两个， 比如P(A,B∣c)和P(C,E∣c)。 然后在时间步3，我们计算所有y3∈Y为：

P(A,B,y3∣c)=P(A,B∣c)P(y3∣A,B,c),

P(C,E,y3∣c)=P(C,E∣c)P(y3∣C,E,c),

公式 3

从这十个值中选择最大的两个， 即P(A,B,D∣c)和P(C,E,D∣c)， 我们会得到六个候选输出序列：（1）A；（2）C；（3）A,B；（4）C,E；（5）A,B,D；（6）C,E,D。

最后，基于这六个序列（例如，丢弃包括“<eos>”和之后的部分）， 我们获得最终候选输出序列集合。然后我们选择其中条件概率乘积最高的序列作为输出序列：



公式 4

其中L是最终候选序列的长度，通常设置为0.75。 因为一个较长的序列在公式4的求和中会有更多的对数项， 因此分母中的用于惩罚长序列。

束搜索的计算量为， 这个结果介于贪心搜索和穷举搜索之间。 实际上，贪心搜索可以看作是一种束宽为1的特殊类型的束搜索。 通过灵活地选择束宽，束搜索可以在正确率和计算代价之间进行权衡。

**随机集束搜索（束搜索的一种变种）**

随机集束搜索是集束搜索的替代方案。由于局部束搜索会很快的集中在状态空间的某个小区域，使得搜索代价比爬山法更昂贵；而随机束搜索则模仿随机爬山法，有助于缓解这个问题；它不是在完成表中选择k个最好后继节点，而是以选择后继节点的概率是其值的递增函数，来随机的选择k个节点。

在这种情况下，质量更好的节点更有可能被选中（有点类似于自然随机的过程）。 这可以通过使用吉布斯分布或玻尔兹曼分布来确定概率分布。

因此，随机集束搜索倾向于允许k节点比集束搜索更多样化。 在生物学的进化方面，评价函数反映了节点的适应性； 对于节点而言，适应的越好的节点更有可能将其所包含的那部分搜索结果传递给下一代。 随机集束搜索就像无性繁殖; 每个人给出轻微变异的孩子，然后随机集束搜索进行，适者生存。

**束搜索与贪心搜索的比较**

相较于贪心搜索，束搜索索是站在整个词序列整体的角度上使其概率最大化。典型的结果是，通过贪心搜索得到的翻译结果可能会包含一些日常使用中词频较高的词。

**束搜索的瓶颈**

束搜索的第一个问题就是由它于在每一步深度扩展的时候都会丢掉一部分节点，最佳解决方案有可能也会被丢弃。另外超参数B（即束宽）的选择有些主观，过小的B可考虑的选择少，搜索结果不好，过大的B太过于费时。

**发展方向**

通过考虑引入多元变量对排序结构产生影响，使得解码结果更具有多样性；

由束搜索的自身性质，考虑在每次解码前利用自动判决选取的束宽对判决过程做先行处理；

通过结合深度优先算法，引入回溯，比对获得最优解。

**附伪代码展示介绍**

/\* initialization \*/

g = 0;

hash\_table = { start };

BEAM = { start };

/\* main loop \*/

while(BEAM ≠ ∅){ // loop until the BEAM contains no nodes

 SET = ∅; // the empty set

 /\* generate the SET nodes \*/

 for(each state in BEAM){

 for(each successor of state){

 if(successor == goal) return g + 1;

 SET = SET ∪ { successor }; // add successor to SET

 }

 }

 BEAM = ∅; // the empty set

 g = g + 1;

 /\* fill the BEAM for the next loop \*/

 while((SET ≠ ∅) AND (B > |BEAM|)){ // set is not empty and the number of nodes in BEAM is less than B

 state = successor in SET with smallest h value;

 SET = SET \ { state }; // remove state from SET

 if(state ∉ hash\_table){ // state is not in the hash\_table

 if(hash\_table is full) return ∞;

 hash\_table = hash\_table ∪ { state }; // add state to hash\_table

 BEAM = BEAM ∪ { state }; // add state to BEAM

 }

 }

}

// goal was not found, and BEAM is empty - Beam Search failed to find the goal

return ∞;

**思考**

通过本篇文章章节，我对于搜索的概念有了进一步的加强，同时对于搜索的优化有了进一步的认知

**习题带做**

4.1

a)爬山法

b)宽度优先。‎因为它在添加下一层之前添加一个完整的节点层。从一个状态开始，该算法本质上与广度优先搜索相同，只是每个图层都是一次性生成的。‎

‎ c）所有时刻T = 0模拟退火：首选爬山算法，因为每个向下的继任者都会被拒绝，概率为1。‎

‎ d）所有时刻T = ∞模拟退火：随机游走搜索：它始终接受新状态。‎

‎ e）群体规模N = 1的遗传算法：如果群体规模为1，则两个选定的父母将是同一个体;交叉产生个体的精确副本;那么突变的机会很小。因此，该算法在个体空间中执行随机游走。