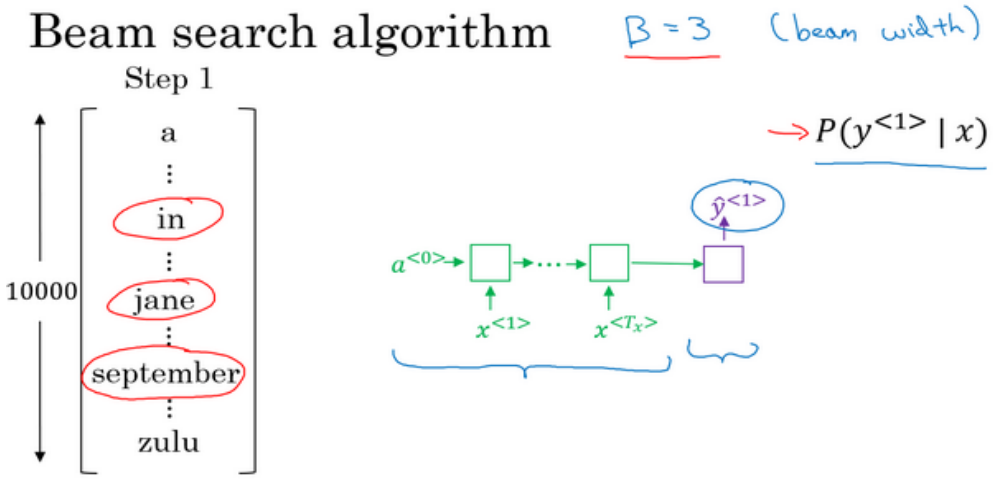
3.3 集束搜索（Beam Search）

这节视频中你会学到集束搜索（**beam search**）算法，上节视频中我们讲了对于机器翻译来说，给定输入，比如法语句子，你不会想要输出一个随机的英语翻译结果，你想要一个最好的，最可能的英语翻译结果。对于语音识别也一样，给定一个输入的语音片段，你不会想要一个随机的文本翻译结果，你想要最好的，最接近原意的翻译结果，集束搜索就是解决这个最常用的算法。这节视频里，你会明白怎么把集束搜索算法应用到你自己的工作中，就用我们的法语句子的例子来试一下集束搜索吧。

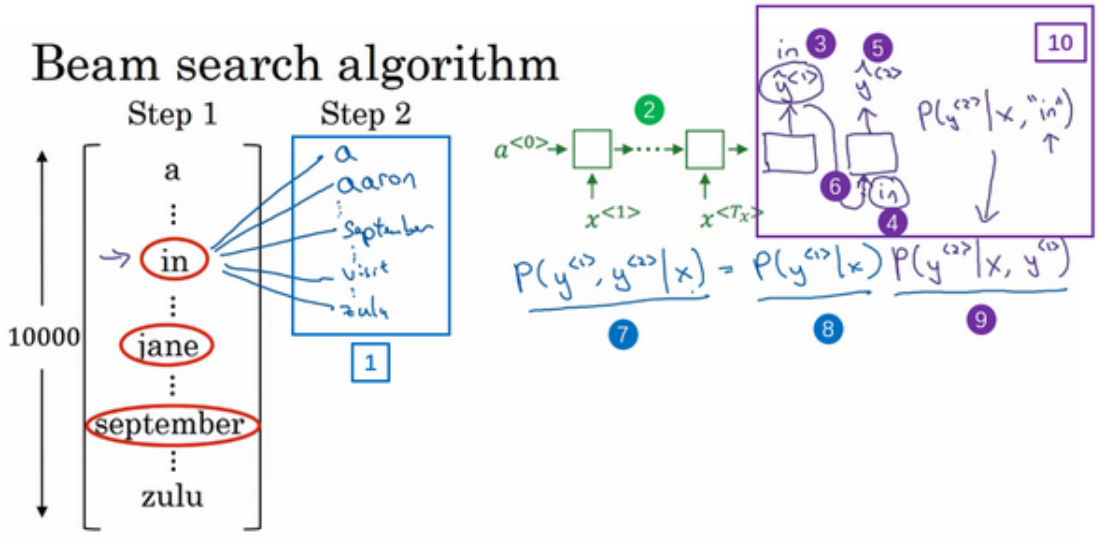
“Jane visite l'Afrique en Septembre.”（法语句子），我们希望翻译成英语，**"Jane is visiting Africa in September"**.（英语句子），集束搜索算法首先做的就是挑选要输出的英语翻译中的第一个单词。这里我列出了 10,000 个词的词汇表（下图编号 1 所示），为了简化问题，我们忽略大小写，所有的单词都以小写列出来。在集束搜索的第一步中我用这个网络部分，绿色是编码部分（下图编号 2 所示），紫色是解码部分（下图编号 3 所示），来评估第一个单词的概率值，给定输入序列x，即法语作为输入，第一个输出y的概率值是多少。



贪婪算法只会挑出最可能的那一个单词，然后继续。而集束搜索则会考虑多个选择，集束搜索算法会有一个参数 **B**，叫做集束宽（**beam width**）。在这个例子中我把这个集束宽设成 3，这样就意味着集束搜索不会只考虑一个可能结果，而是一次会考虑 3 个，比如对第一个单词有不同选择的可能性，最后找到 **in**、**jane**、**september**，是英语输出的第一个单词的最可能的三个选项，然后集束搜索算法会把结果存到计算机内存里以便后面尝试用这三个词。如果集束宽设的不一样，如果集束宽这个参数是 10 的话，那么我们跟踪的不仅仅 3 个，而是 10 个第一个单词的最可能的选择。**所以要明白，为了执行集束搜索的第一步，你需要**

输入法语句子到编码网络，然后会解码这个网络，这个 softmax 层（上图编号 3 所示）会输出 10,000 个概率值，得到这 10,000 个输出的概率值，取前三个存起来。

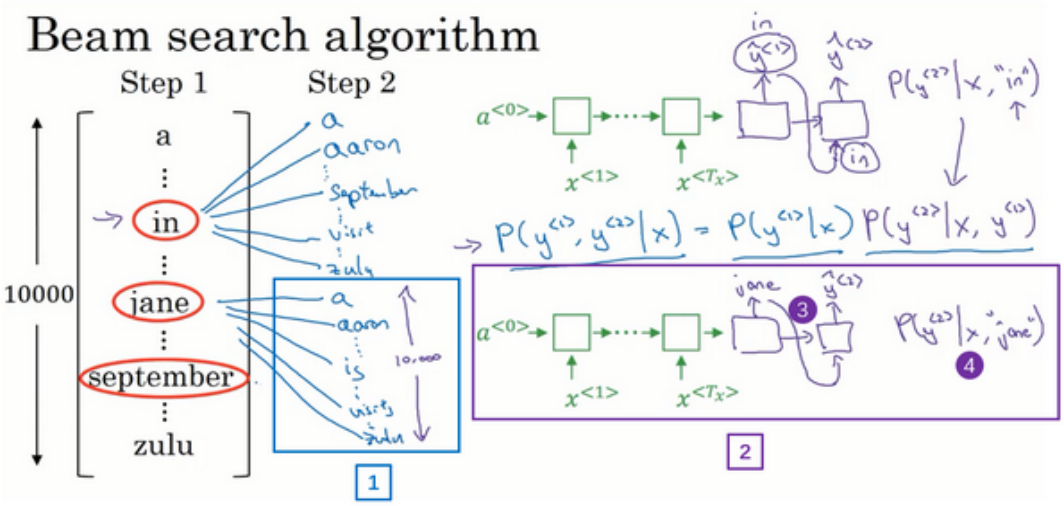
让我们看看集束搜索算法的第二步，已经选出了 **in**、**jane**、**september** 作为第一个单词三个最可能的选择，集束算法接下来会针对每个第一个单词考虑第二个单词是什么，单词 **in** 后面的第二个单词可能是 **a** 或者是 **aaron**，我就是从词汇表里把这些词列了出来，或者是列表里某个位置，**september**，可能是列表里的 **visit**，一直到字母 **z**，最后一个单词是 **zulu**（下图编号 1 所示）。



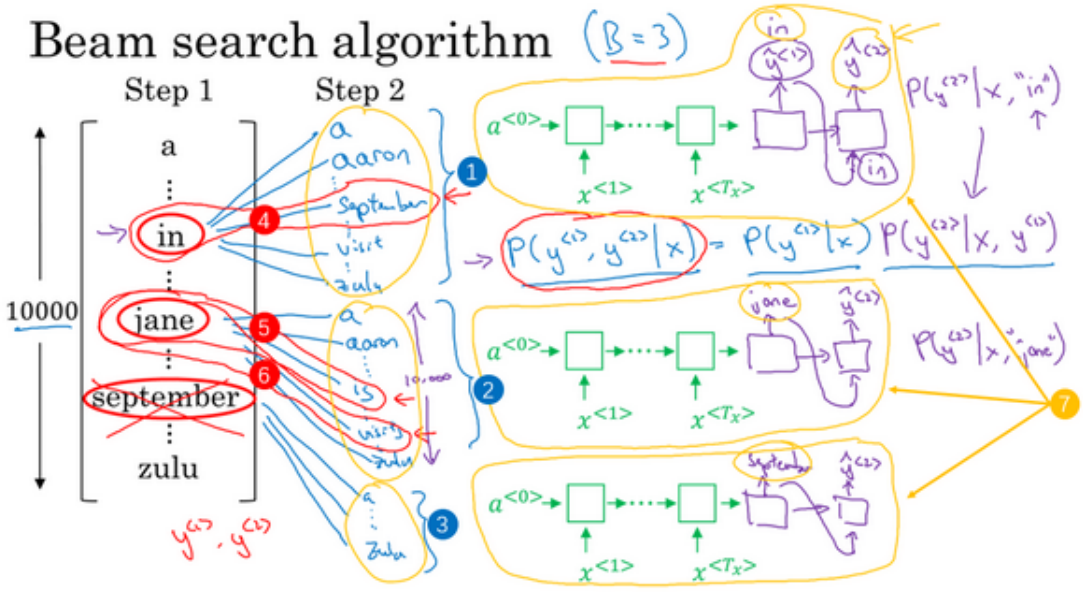
为了评估第二个词的概率值，我们用这个神经网络的部分，绿色是编码部分（上图编号 2 所示），而对于解码部分，当决定单词 **in** 后面是什么，别忘了解码器的第一个输出$y^{<1>}$，我把$y^{<1>}$设为单词 **in**（上图编号 3 所示），然后把它喂回来，这里就是单词 **in**（上图编号 4 所示），因为它的目的是努力找出第一个单词是 **in** 的情况下，第二个单词是什么。这个输出就是$y^{<2>}$（上图编号 5 所示），有了这个连接（上图编号 6 所示），就是这里的第一个单词 **in**（上图编号 4 所示）作为输入，这样这个网络就可以用来评估第二个单词的概率了，在给定法语句子和翻译结果的第一个单词 **in** 的情况下。

注意，在第二步里我们更关心的是要找到最可能的第一个和第二个单词对，所以不仅仅是第二个单词有最大的概率，而是第一个、第二个单词对有最大的概率（上图编号 7 所示）。按照条件概率的准则，这个可以表示成第一个单词的概率（上图编号 8 所示）乘以第二个单词的概率（上图编号 9 所示），这个可以从这个网络部分里得到（上图编号 10 所示），对于已经选择的 **in**、**jane**、**september** 这三个单词，你可以先保存这个概率值（上图编号 8 所示），然后再乘以第二个概率值（上图编号 9 所示）就得到了第一个和第二个单词对的概率（上图

编号 7 所示）。



现在你已经知道在第一个单词是 in 的情况下如何评估第二个单词的概率，现在第一个单词是 **jane**，道理一样，句子可能是"**jane a**"、"**jane aaron**"，等等到"**jane is**"、"**jane visits**"等等（上图编号 1 所示）。你会用这个新的网络部分（上图编号 2 所示），我在这里画一条线，代表从$y^{<1>}$，即 **jane**，$y^{<1>}$连接 **jane**（上图编号 3 所示），那么这个网络部分就可以告诉你给定输入x和第一个词是 **jane** 下，第二个单词的概率了（上图编号 4 所示），和上面一样，你可以乘以$P(y^{<1>}|x)$得到$P(y^{<1>}, y^{<2>}|x)$。

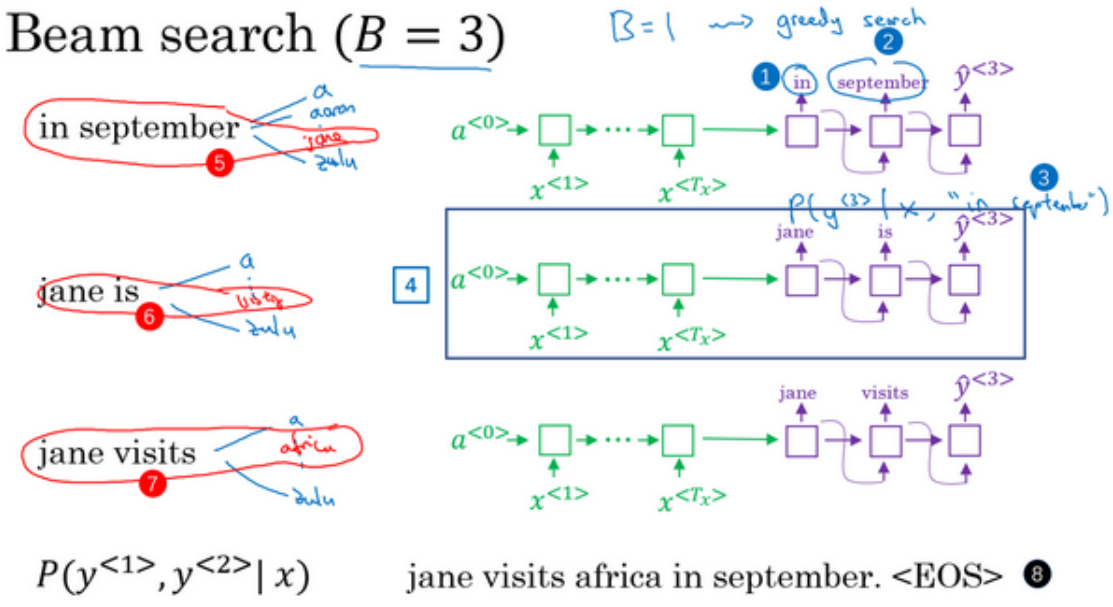


针对第二个单词所有 10,000 个不同的选择，最后对于单词 **september** 也一样，从单词 **a** 到单词 **zulu**，用这个网络部分，我把它画在这里。来看看如果第一个单词是 **september**，第二个单词最可能是什么。所以对于集束搜索的第二步，由于我们一直用的集束宽为 3，并且词汇表里有 10,000 个单词，那么最终我们会有 3 乘以 10,000 也就是 30,000 个可能的结

果，因为这里（上图编号 1 所示）是 10,000，这里（上图编号 2 所示）是 10,000，这里（上图编号 3 所示）是 10,000，就是集束宽乘以词汇表大小，你要做的就是评估这 30,000 个选择。按照第一个词和第二个词的概率，然后选出前三个，这样又减少了这 30,000 个可能性，又变成了 3 个，减少到集束宽的大小。假如这 30,000 个选择里最可能的是**“in September”**（上图编号 4 所示）和**“jane is”**（上图编号 5 所示），以及**“jane visits”**（上图编号 6 所示），画的有点乱，但这就是这 30,000 个选择里最可能的三个结果，集束搜索算法会保存这些结果，然后用于下一次集束搜索。

注意一件事情，如果集束搜索找到了第一个和第二个单词对最可能的三个选择是**“in September”**或者**“jane is”**或者**“jane visits”**，这就意味着我们去掉了 **september** 作为英语翻译结果的第一个单词的选择，所以我们的第一个单词现在减少到了两个可能结果，但是我们的集束宽是 3，所以还是有$y^{<1>}$，$y^{<2>}$对的三个选择。

在我们进入集束搜索的第三步之前，我还想提醒一下因为我们的集束宽等于 3，每一步我们都复制 3 个，同样的这种网络来评估部分句子和最后的结果，由于集束宽等于 3，我们有三个网络副本（上图编号 7 所示），每个网络的第一个单词不同，而这三个网络可以高效地评估第二个单词所有的 30,000 个选择。所以不需要初始化 30,000 个网络副本，只需要使用 3 个网络的副本就可以快速的评估 **softmax** 的输出，即$y^{<2>}$的 10,000 个结果。



让我们快速解释一下集束搜索的下一步，前面说过前两个单词最可能的选择是**“in September”**和**“jane is”**以及**“jane visits”**，对于每一对单词我们应该保存起来，给定输入x，即法语句子作为x的情况下，$y^{<1>}$和$y^{<2>}$的概率值和前面一样，现在我们考虑第三个单词是什么，可以是**“in September a”**，可以是**“in September aaron”**，一直到**“in September zulu”**。为

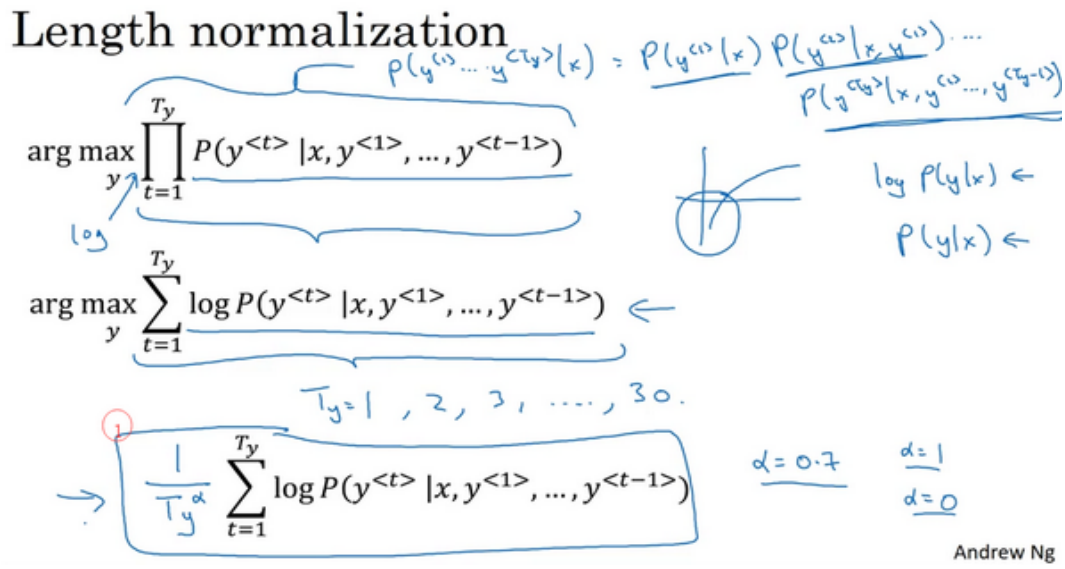
了评估第三个单词可能的选择，我们用这个网络部分，第一单词是 **in**（上图编号 1 所示），第二个单词是 **september**（上图编号 2 所示），所以这个网络部分可以用来评估第三个单词的概率，在给定输入的法语句子x和给定的英语输出的前两个单词**"in September"**情况下（上图编号 3 所示）。对于第二个片段来说也一样，就像这样一样（上图编号 4 所示），对于**"jane visits"**也一样，然后集束搜索还是会挑选出针对前三个词的三个最可能的选择，可能是**"in september jane"**（上图编号 5 所示），**"Jane is visiting"**也很有可能（上图编号 6 所示），也很可能是**"Jane visits Africa"**（上图编号 7 所示）。

然后继续，接着进行集束搜索的第四步，再加一个单词继续，最终这个过程的输出一次增加一个单词，集束搜索最终会找到**"Jane visits africa in september"**这个句子，终止在句尾符号（上图编号 8 所示），用这种符号的系统非常常见，它们会发现这是最有可能输出的一个英语句子。在本周的练习中，你会看到更多的执行细节，同时，你会运用到这个集束算法，在集束宽为 3 时，集束搜索一次只考虑 3 个可能结果。注意如果集束宽等于 1，只考虑 1 种可能结果，这实际上就变成了贪婪搜索算法，上个视频里我们已经讨论过了。但是如果同时考虑多个，可能的结果比如 3 个，10 个或者其他的个数，集束搜索通常会找到比贪婪搜索更好的输出结果。

你已经了解集束搜索是如何工作的了，事实上还有一些额外的提示和技巧的改进能够使集束算法更高效，我们在下个视频中一探究竟。

3.4 改进集束搜索（Refinements to Beam Search）

上个视频中，你已经学到了基本的束搜索算法(**the basic beam search algorithm**)，这个视频里,我们会学到一些技巧，能够使算法运行的更好。长度归一化（**Length normalization**）就是对束搜索算法稍作调整的一种方式，帮助你得到更好的结果，下面介绍一下它。



前面讲到束搜索就是最大化这个概率，这个乘积就是$P(y^{<1>}\ldots y^{<T_y>}|X)$，可以表示成:$P(y^{<1>}|X)\ P(y^{<2>}|X,y^{<1>})\ P(y^{<3>}|X,y^{<1>},y^{<2>})\ldots P(y^{<T_y>}|X,y^{<1>},y^{<2>}\ldots y^{<T_y-1>})$

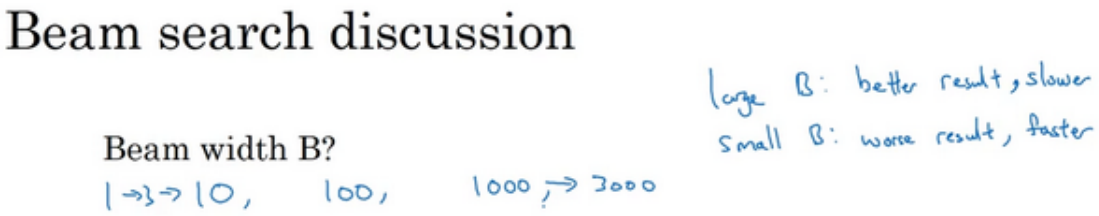
这些符号看起来可能比实际上吓人，但这就是我们之前见到的乘积概率（**the product probabilities**）。如果计算这些，其实这些概率值都是小于 1 的，通常远小于 1。很多小于 1 的数乘起来，会得到很小很小的数字，会造成数值下溢（**numerical underflow**）。数值下溢就是数值太小了，导致电脑的浮点表示不能精确地储存，因此在实践中,我们不会最大化这个乘积，而是取log值。如果在这加上一个log，最大化这个log求和的概率值，在选择最可能的句子y时，你会得到同样的结果。所以通过取log，我们会得到一个数值上更稳定的算法，不容易出现四舍五入的误差，数值的舍入误差（**rounding errors**）或者说数值下溢（**numerical underflow**）。因为log函数它是严格单调递增的函数，最大化$P(y)$，因为对数函数，这就是log函数，是严格单调递增的函数，所以最大化$logP(y|x)$和最大化$P(y|x)$结果一样。如果一个y值能够使前者最大，就肯定能使后者也取最大。所以实际工作中，我们总是记录概率的对数和（**the sum of logs of the probabilities**），而不是概率的乘积（**the production of probabilities**）。

对于目标函数（**this objective function**），还可以做一些改变，可以使得机器翻译表现的

更好。如果参照原来的目标函数（**this original objective**），如果有一个很长的句子，那么这个句子的概率会很低，因为乘了很多项小于 1 的数字来估计句子的概率。所以如果乘起来很多小于 1 的数字，那么就会得到一个更小的概率值，所以这个目标函数有一个缺点，它可能不自然地倾向于简短的翻译结果，它更偏向短的输出，因为短句子的概率是由更少数量的小于 1 的数字乘积得到的，所以这个乘积不会那么小。顺便说一下，这里也有同样的问题，概率的log值通常小于等于 1，实际上在log的这个范围内，所以加起来的项越多，得到的结果越负，所以对这个算法另一个改变也可以使它表现的更好，也就是我们不再最大化这个目标函数了，**我们可以把它归一化，通过除以翻译结果的单词数量（normalize this by the number of words in your translation）**。这样就是取每个单词的概率对数值的平均了，这样很明显地减少了对输出长的结果的惩罚（**this significantly reduces the penalty for outputting longer translations.**）。

在实践中，有个探索性的方法，相比于直接除T_y，也就是输出句子的单词总数，我们有时会用一个更柔和的方法（**a softer approach**），**在T_y上加上指数α**，α可以等于 0.7。如果α等于 1，就相当于完全用长度来归一化，如果α等于 0，T_y的 0 次幂就是 1，就相当于完全没有归一化，这就是在完全归一化和没有归一化之间。α就是算法另一个超参数（**hyper parameter**），需要调整大小来得到最好的结果。不得不承认，这样用α实际上是试探性的，它并没有理论验证。但是大家都发现效果很好，大家都发现实践中效果不错，所以很多人都会这么做。你可以尝试不同的α值，看看哪一个能够得到最好的结果。

总结一下如何运行束搜索算法。当你运行束搜索时，你会看到很多长度等于 1 的句子，很多长度等于 2 的句子，很多长度等于 3 的句子，等等。可能运行束搜索 30 步，考虑输出的句子可能达到，比如长度 30。因为束宽为 3，你会记录所有这些可能的句子长度，长度为 1、2、3、4 等等一直到 30 的三个最可能的选择。然后针对这些所有的可能的输出句子，用这个式子（上图编号 1 所示）给它们打分，取概率最大的几个句子，然后对这些束搜索得到的句子，计算这个目标函数。最后从经过评估的这些句子中，挑选出在归一化的log 概率目标函数上得分最高的一个（**you pick the one that achieves the highest value on this normalized log probability objective.**），有时这个也叫作**归一化的对数似然目标函数**（**a normalized log likelihood objective**）。这就是最终输出的翻译结果，这就是如何实现束搜索。这周的练习中你会自己实现这个算法。



Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find exact maximum for $\arg\max_y P(y|x)$.

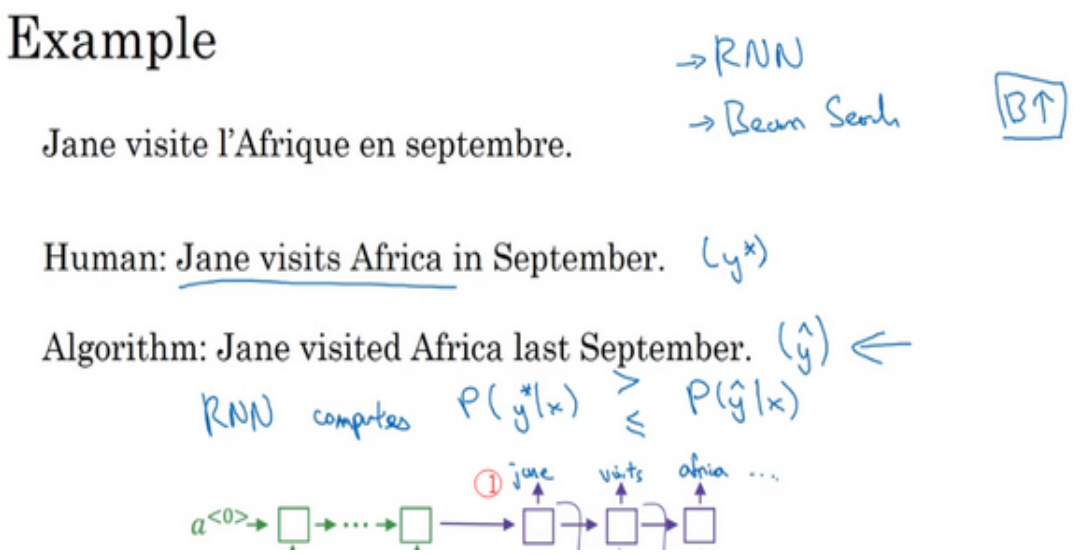
最后还有一些实现的细节，如何选择束宽 **B**。**B** 越大，你考虑的选择越多，你找到的句子可能越好，但是 **B** 越大，你的算法的计算代价越大，因为你要把很多的可能选择保存起来。最后我们总结一下关于如何选择束宽 **B** 的一些想法。接下来是针对或大或小的 **B** 各自的优缺点。如果束宽很大，你会考虑很多的可能，你会得到一个更好的结果，因为你要考虑很多的选择，但是算法会运行的慢一些，内存占用也会增大，计算起来会慢一点。而如果你用小的束宽，结果会没那么好，因为你在算法运行中，保存的选择更少，但是你的算法运行的更快，内存占用也小。在前面视频里，**我们例子中用了束宽为 3，所以会保存 3 个可能选择，在实践中这个值有点偏小。在产品中，经常可以看到把束宽设到 10，**我认为束宽为 100 对于产品系统来说有点大了，这也取决于不同应用。但是对科研而言，人们想压榨出全部性能，这样有个最好的结果用来发论文，也经常看到大家用束宽为 1000 或者 3000，这也是取决于特定的应用和特定的领域。在你实现你的应用时，尝试不同的束宽的值，当 B 很大的时候，性能提高会越来越少。对于很多应用来说，从束宽 1，也就是贪心算法，到束宽为 3、到 10，你会看到一个很大的改善。但是当束宽从 1000 增加到 3000 时，效果就没那么明显了。对于之前上过计算机科学课程的同学来说，如果你熟悉计算机科学里的搜索算法（**computer science search algorithms**），比如广度优先搜索（**BFS, Breadth First Search algorithms**），或者深度优先搜索（**DFS, Depth First Search**），你可以这样想束搜索，不像其他你在计算机科学算法课程中学到的算法一样。如果你没听说过这些算法也不要紧，但是如果你听说过广度优先搜索和深度优先搜索，不同于这些算法，这些都是精确的搜索算法（**exact search algorithms**），束搜索运行的更快，但是不能保证一定能找到 **argmax** 的准确的最大值。如果你没听说过广度优先搜索和深度优先搜索，也不用担心，这些对于我们的目标也不重要，如果你听说过，这就是束搜索和其他算法的关系。

好，这就是束搜索。这个算法广泛应用在多产品系统或者许多商业系统上，在深度学习系列课程中的第三门课中，我们讨论了很多关于误差分析（**error analysis**）的问题。事实上在束搜索上做误差分析是我发现的最有用的工具之一。有时你想知道是否应该增大束宽，我的束宽是否足够好，你可以计算一些简单的东西来指导你需要做什么，来改进你的搜索算法。我们在下个视频里进一步讨论。

3.5 集束搜索的误差分析（Error analysis in beam search）

在这五门课中的第三门课里，你了解了误差分析是如何能够帮助你集中时间做你的项目中最有用的工作，束搜索算法是一种近似搜索算法（an approximate search algorithm），也被称作启发式搜索算法（a heuristic search algorithm），它不总是输出可能性最大的句子，它仅记录着 **B** 为前 3 或者 10 或是 100 种可能。那么如果束搜索算法出现错误会怎样呢?

本节视频中，你将会学习到误差分析和束搜索算法是如何相互起作用的，以及你怎样才能发现是束搜索算法出现了问题，需要花时间解决，还是你的 **RNN** 模型出了问题，要花时间解决。我们先来看看如何对束搜索算法进行误差分析。

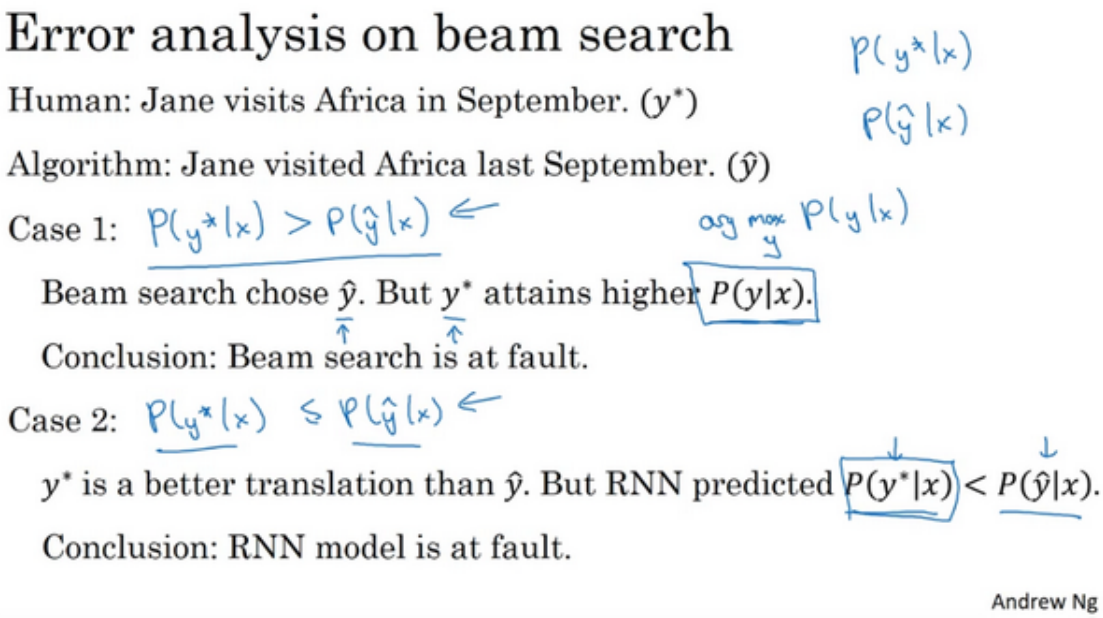


我们来用这个例子说明： **“Jane visite l'Afrique en septembre”**。假如说，在你的机器翻译的 **dev** 集中，也就是开发集（**development set**），人工是这样翻译的: **Jane visits Africa in September**,我会将这个标记为y^*。这是一个十分不错的人工翻译结果，不过假如说，当你在已经完成学习的 **RNN** 模型，也就是已完成学习的翻译模型中运行束搜索算法时，它输出了这个翻译结果：**Jane visited Africa last September**，我们将它标记为$\hat{y}$。这是一个十分糟糕的翻译，它实际上改变了句子的原意，因此这不是个好翻译。

你的模型有两个主要部分，一个是神经网络模型，或说是序列到序列模型（sequence to sequence model），我们将这个称作是 RNN 模型，它实际上是个编码器和解码器（ an encoder and a decoder）。另一部分是束搜索算法，以某个集束宽度 B 运行。如果你能够找出造成这个错误，这个不太好的翻译的原因，是两个部分中的哪一个，不是很好吗? **RNN (循环神经网络)**是更可能是出错的原因呢，还是束搜索算法更可能是出错的原因呢？你在第三门课中了

解到了大家很容易想到去收集更多的训练数据，这总归没什么坏处。所以同样的，大家也会觉得不行就增大束宽，也是不会错的，或者说是很大可能是没有危害的。但是就像单纯获取更多训练数据，可能并不能得到预期的表现结果。相同的，单纯增大束宽也可能得不到你想要的结果，不过你怎样才能知道是不是值得花时间去改进搜索算法呢？下面我们来分解这个问题弄清楚什么情况下该用什么解决办法。

RNN (循环神经网络)实际上是个编码器和解码器（**the encoder and the decoder**），它会计算$P(y|x)$。所以举个例子，对于这个句子：**Jane visits Africa in September**，你将 **Jane visits Africa** 填入这里（上图编号 1 所示），同样，我现在忽略了字母的大小写，后面也是一样，然后这个就会计算。$P(y|x)$结果表明，**你此时能做的最有效的事就是用这个模型来计算$P(y^*|x)$，同时也用你的 RNN 模型来计算$P(\hat{y}|x)$，然后比较一下这两个值哪个更大。**有可能是左边大于右边，也有可能是$P(y^*)$小于$P(\hat{y})$，其实应该是小于或等于，对吧。取决于实际是哪种情况，你就能够更清楚地将这个特定的错误归咎于 **RNN** 或是束搜索算法，或说是哪个负有更大的责任。我们来探究一下其中的逻辑。

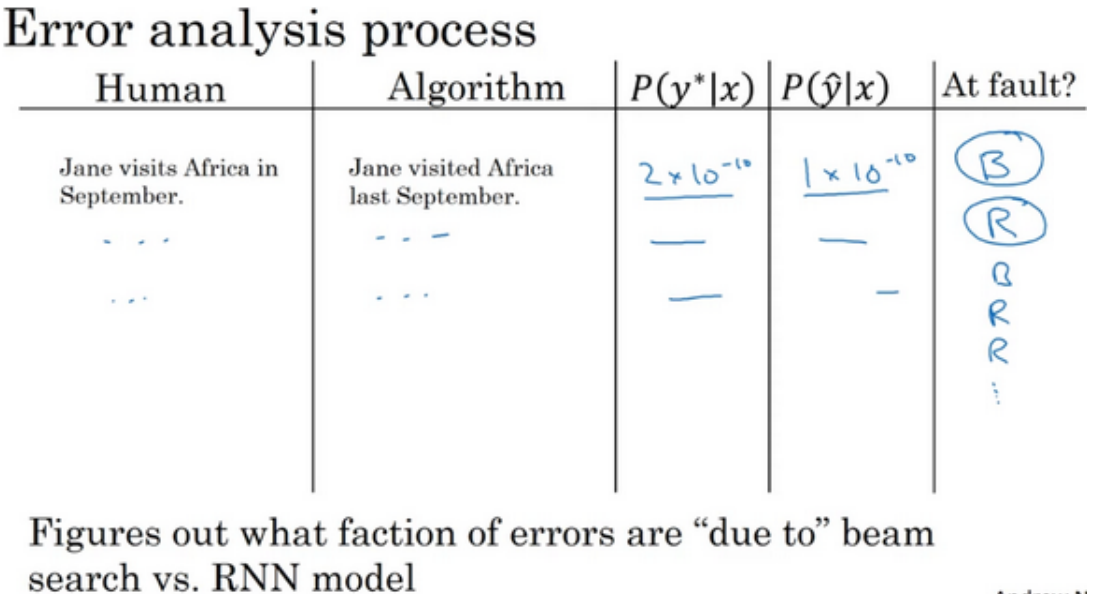


这是之前幻灯片里的两个句子。记住，我们是要计算$P(y^*|x)$和$P(\hat{y}|x)$，然后比较这两个哪个更大，所以就会有两种情况。

第一种情况，**RNN** 模型的输出结果$P(y^*|x)$ 大于$P(\hat{y}|x)$，这意味着什么呢？束搜索算法选择了$\hat{y}$ ，对吧？你得到$\hat{y}$的方式是，你用一个 **RNN** 模型来计算$P(y|x)$，然后束搜索算法做的就是尝试寻找使$P(y|x)$最大的y，不过在这种情况下，相比于$\hat{y}$，y^*的值更$P(y|x)$大，因此你能够得出束搜索算法实际上不能够给你一个能使$P(y|x)$最大化的y值，因为束搜索算法的任务就是寻找一个y的值来使这项更大，但是它却选择了$\hat{y}$，而y^*实际上能得到更大的值。因

此这种情况下你能够得出是束搜索算法出错了。那另一种情况是怎样的呢?

第二种情况是$P(y^*|x)$小于或等于$P(\hat{y}|x)$对吧？这两者之中总有一个是真的。情况 1 或是情况 2 总有一个为真。情况 2 你能够总结出什么呢? 在我们的例子中，y^* 是比 $\hat{y}$更好的翻译结果，不过根据 RNN 模型的结果，$P(y^*)$ 是小于$P(\hat{y})$的，也就是说，相比于$\hat{y}$，y^*成为输出的可能更小。因此在这种情况下，看来是 **RNN** 模型出了问题。同时可能值得在 **RNN** 模型上花更多时间。这里我少讲了一些有关长度归一化（**length normalizations**）的细节。这里我略过了有关长度归一化的细节，如果你用了某种长度归一化，那么你要做的就不是比较这两种可能性大小，而是比较长度归一化后的最优化目标函数值。不过现在先忽略这种复杂的情况。第二种情况表明虽然y^*是一个更好的翻译结果，**RNN** 模型却赋予它更低的可能性，是 **RNN** 模型出现了问题。



所以误差分析过程看起来就像下面这样。你先遍历开发集，然后在其中找出算法产生的错误，这个例子中，假如说$P(y^*|x)$的值为 2 x 10-10，而$P(\hat{y}|x)$的值为 1 x10-10，根据上页幻灯片中的逻辑关系，这种情况下我们得知束搜索算法实际上选择了比y^*可能性更低的$\hat{y}$，因此我会说束搜索算法出错了。我将它缩写为 **B**。接着你继续遍历第二个错误，再来看这些可能性。也许对于第二个例子来说，你认为是 **RNN** 模型出现了问题，我会用缩写 **R** 来代表 **RNN**。再接着你遍历了更多的例子，有时是束搜索算法出现了问题，有时是模型出现了问题，等等。通过这个过程，你就能够执行误差分析，得出束搜索算法和 **RNN** 模型出错的比例是多少。有了这样的误差分析过程，你就可以对开发集中每一个错误例子，即算法输出了比人工翻译更差的结果的情况，尝试确定这些错误，是搜索算法出了问题，还是生成目标函数(束搜索算法使之最大化)的 **RNN** 模型出了问题。并且通过这个过程，你能够发现这两个部分中

哪个是产生更多错误的原因，并且只有当你发现是束搜索算法造成了大部分错误时，才值得花费努力增大集束宽度。相反地，如果你发现是 **RNN** 模型出了更多错，那么你可以进行更深层次的分析，来决定是需要增加正则化还是获取更多的训练数据，抑或是尝试一个不同的网络结构，或是其他方案。你在第三门课中，了解到各种技巧都能够应用在这里。

这就是束搜索算法中的误差分析，我认为这个特定的误差分析过程是十分有用的，它可以用于分析近似最佳算法(如束搜索算法)，这些算法被用来优化学习算法(例如序列到序列模型/**RNN**)输出的目标函数。也就是我们这些课中一直讨论的。学会了这个方法，我希望你能够在你的应用里更有效地运用好这些类型的模型。