7月18日

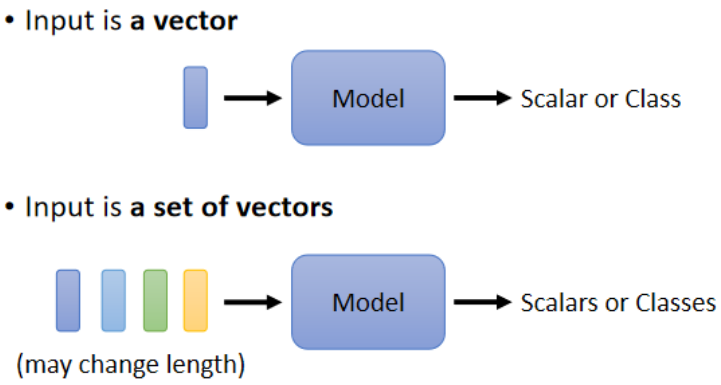
一、李宏毅2021春机器学习课程第4.1节：自注意力机制（一）

1 问题引入：为什么需要自注意力机制

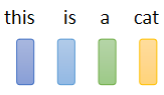
在探讨了CNN的架构以后，我们要探讨的另一个常见的Network架构，叫做**Self-Attention**，至于Self-Attention 是什么我们先按下不表，我们先来看看Self-Attention出现的背景。

1.1 更加复杂的输入

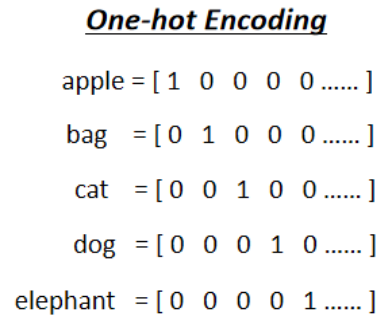
到目前为止，我们的Network的**输入都是一个向量**，而且这个向量的大小往往是固定的，就像我们在CNN中要求输入的图片大小需要统一。那**现在假设我们的输入是多个向量，并且输入向量的个数是不确定的**，那我们又该如何处理呢？



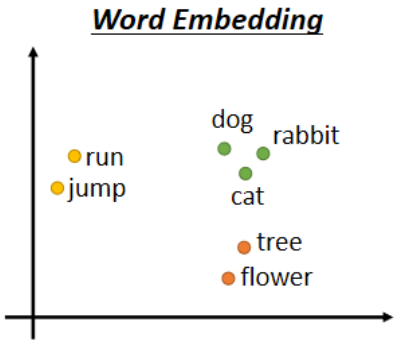
假设我们今天要Network处理的输入是一个句子，每一个句子的长度都不一样，并且每个句子里面词汇的数目也不一样。



如果我们把一个句子里面的每一个词汇，都描述成一个向量，那我们的输入就会是一个Vector Set，而且每次输入时句子的长度不一样，Vector Set的大小就不一样。至于**怎么把一个词汇表示成一个向量**，最简单的做法是One-Hot的Encoding



但是One-Hot的Encoding的表示方法有一个非常严重的问题，它**假设所有的词汇彼此之间都是没有关系的**，但事实上Cat跟Dog都是动物所以他们的表示应该比较接近，而Cat跟Apple一个动物一个植物，他们的表示应该比较疏远。能够实现这种表示的一个方法叫做Word Embedding。



> Word Embedding 给每个单词分配一个固定长度的向量表示，这个长度可以自行设定，比如300，实际上会远远小于字典长度（比如10000）。而且两个单词向量之间的夹角值可以作为他们之间关系的一个衡量

如果你把Word Embedding画出来的话，你会发现所有的动物可能聚集成一团，所有的植物可能聚集成一团，所有的动词可能聚集成一团等等。

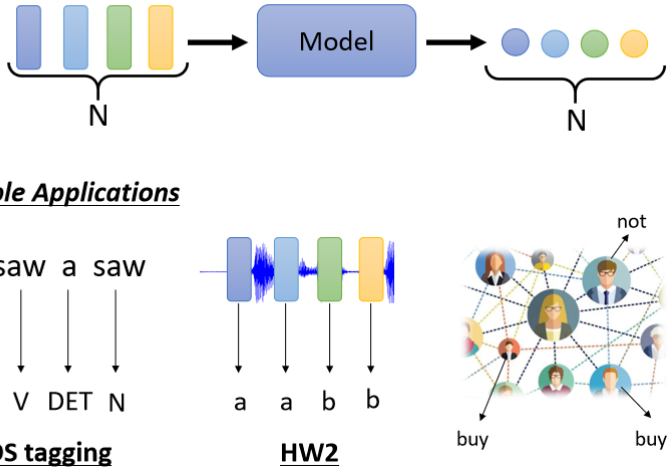
1.2 更多可能的输出

我们的输入是一堆向量，它可以是文字，可以是语音，可以是Graph，那这个时候，我们有可能有什么样的输出呢，有如下的三种可能性：

1.2.1 每一个向量都有一个对应的Label

当你的模型看到输入是四个向量的时候，它就要输出四个Label，而每一个Label如果是一个数值，那就是Regression的问题；如果是一个Class，那就是一个Classification的问题

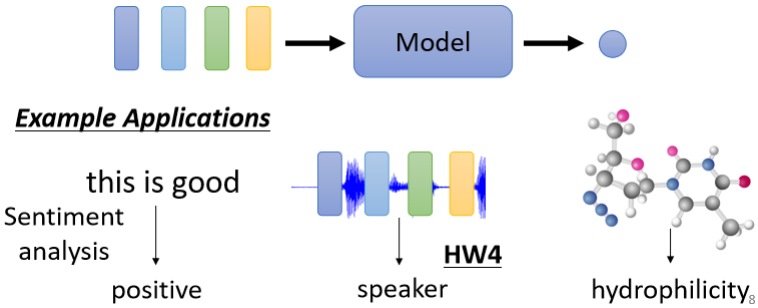
- Each vector has a label.



举例来说，在文字处理的问题上，假设你今天要做的是POS Tagging，POS Tagging就是**词性标注**，你要让机器自动识别句子中的每一个词汇是什么词性，它是名词、动词还是形容词等等。

这个任务其实并没有很容易，举例来说，你现在看到一个句子，“I saw a saw.” 这句话的意思是“我看到一个锯子”，第一个saw是动词，而第二个saw是名词。对于这种一词多义的情况，处理起来还是比较困难的。总之这个任务的输入跟输出的长度是一样的，这就属于我们所说的每一个向量都有一个对应的Label这种情况。

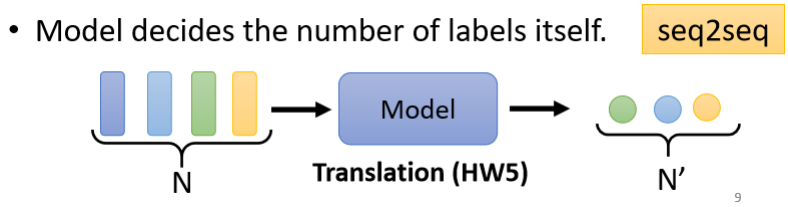
1.2.2 输入多个向量，只需要输出一个Label



举例来说，如果输入是句子的话，我们需要做Sentiment Analysis，就是给机器看一段话，它要**决定说这段话是正面的还是负面的**，这样一整个句子的输入只需要产生一个输出的Label，表示Positive或者Negative。

1.2.3 机器要自己决定应该输出多少个Label

有时候我们不知道应该最终需要输出多少个Label，机器要自己决定，这种任务又叫做sequence to sequence的任务。

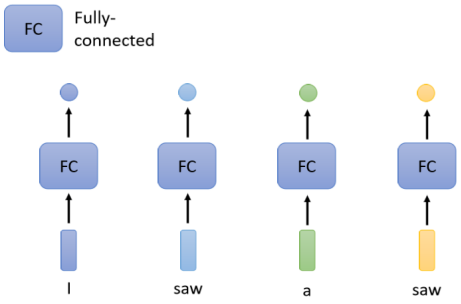


翻译就是sequence to sequence的任务，因为输入输出是不同的语言，它们的词汇的数目本来就不会一样多。此外语音辨识也是sequence to sequence的任务，输入一句话的语音信号，输出一段文字。

1.3 处理词性标注问题的困境

对于之前提到的三种可能的输出，第一种可能，也就是输入跟输出数目一样多的状况又叫做Sequence Labeling，我们需要给Sequence里的每一个向量一个对应的Label，那要怎么解决这个问题呢?

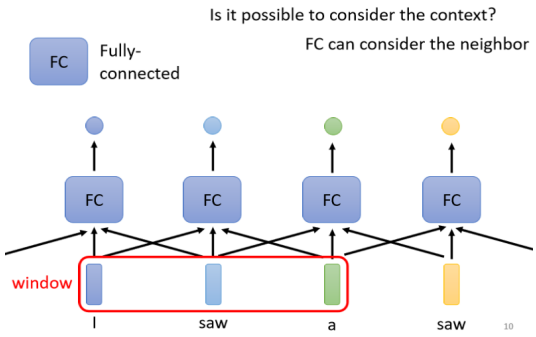
由于我们以前的Fully Connected的Network都是一个输入一个输出，那既然要多个输入多个输出，最直观的想法就是我们**用多个Fully Connected的Network来分别接收输入Sequence中的每一个向量，再对应输出就好。**



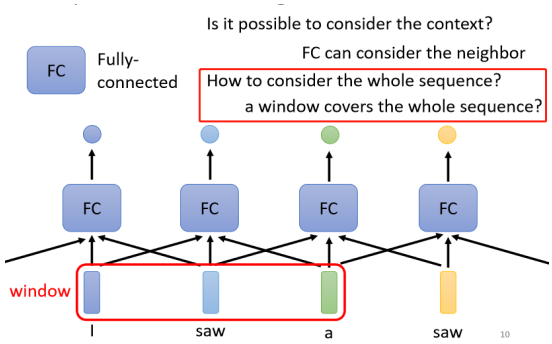
那这么做**有一个很明显的问题**，假设我们的Network是做词性标记的任务，你给机器一个句子 “I saw a saw”，对Fully Connected Network来说，**后面这一个saw跟前面这个saw完全一模一样**，因为它们是同一个词汇，所有分别处理这两个saw的Network**输入同一个词汇，它没有理由输出不同的东西。**

但实际上我们却期待机器在输入第一个saw后要输出动词，输入第二个saw后要输出名词，但现在看来用Fully Connected Network的方法是不可能做到的。那有没有可能**让Fully Connected的Network在看到一个输入的同时，还可以去考虑这个输入的上下文的资讯呢?**

这当然是有可能的，我们可以简单地**把前后几个向量都串起来，一起丢到Fully-Connected的Network就结束了。**假如我们把当前要处理的向量和它的前一个以及后一个向量一起丢进Network，这三个向量就组成了我们的一个**Window**，这时这个Network的输出就可以同时考虑这个Window内所有向量的资讯。



但如果今天我们有某一个任务，不是考虑一个Window就可以解决的，而是要**考虑整个Sequence才能够解决**的话，又要怎么办呢?

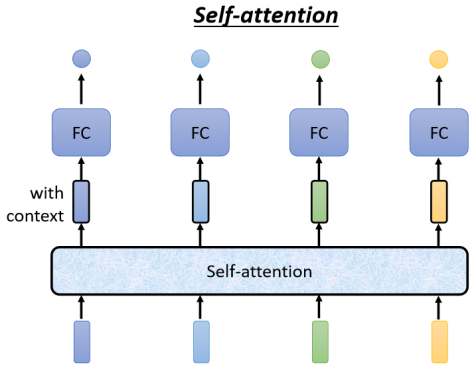


那我们可能会说这个很容易呀，**把Window开大一点，大到可以把整个Sequence盖住就可以了嘛。**但实际上我们的**Sequence的长度是有长有短的**，如果真的要用一个Window把全部Sequence盖住，那可能要**统计一下训练资料里面最长的Sequence有多长。**但是如果开一个这么大的Window，意味着Fully Connected Network需要非常多的参数，那可能不只**运算量很大，还容易Overfitting。**

所以有没有更好的方法来考虑整个Sequence的资讯呢，这就引出了接下来要介绍的Self-Attention这个技术。

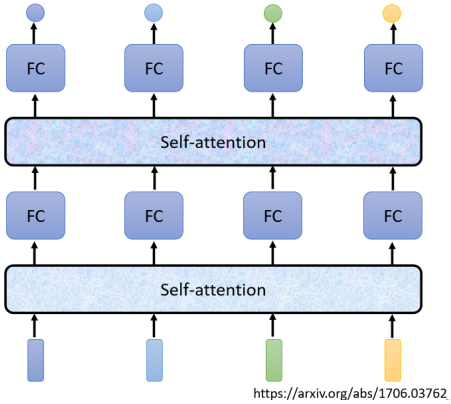
---

2 自注意力机制（Self-Attention）



Self-Attention会考虑一整个Sequence的资讯，输入几个Vector它就输出几个Vector，**输出的这4个Vector，他们都是考虑一整个Sequence以后才得到的**，至于它是如何做到考虑一整个Sequence的资讯的，等一会儿再解释。

而且Self-Attention不是只能用一次，你可以叠加很多次，可以Self-Attention的输出，通过Fully Connected Network以后，再做一次Self-Attention，Fully-Connected的Network，如下图所示，最后再得到最终的结果。所以**可以把Fully Connected的Network跟Self-Attention交替使用。**



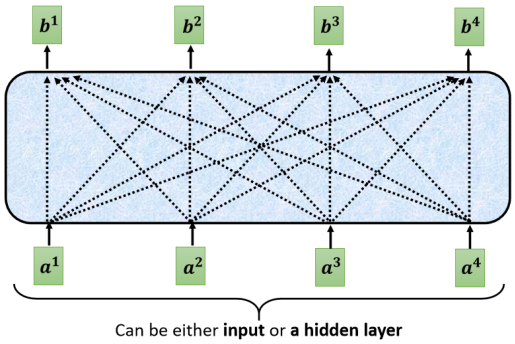
有关Self-Attention，最知名的一篇文章就是 Attention Is All You Need (arxiv.org)，在这篇Paper里面，Google提出了Transformer这样的Network架构，Transformer的架构之后会讲到，这里我们只需要知道Transformer里面一个最重要的Module就是Self-Attention，**它就是变形金刚的火种源。**

接下来我们来看一看Self-Attention具体是怎么运作的，它是如何做到考虑一整个Sequence的资讯的。

---

3 Self-Attention的具体运作过程

我们假设Self-Attention的Input就是一串的Vector，那**这个Vector可能是你整个Network的Input，也可能是某个Hidden Layer的Output**，所以我们这边不是用x来表示它，而是用a来表示它，代表它有可能是前面已经做过一些处理。

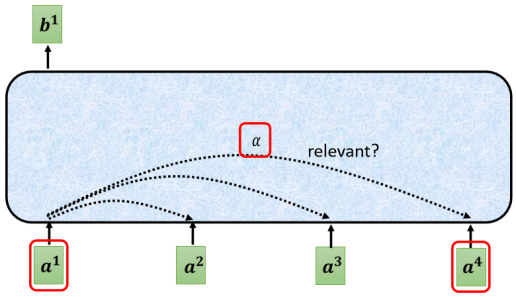


所以从上图可以看到我们的Self-Attention会输入一排向量a，最后输出另外一排向量b。

那这**每一个b都是考虑了所有的a以后才生成出来的**,所以这边刻意画了非常非常多的箭头,告诉你b^1考虑了a^1到a^4產生的,b^2考虑a^1到a^4產生的,b^3 b^4也是一样,考虑整个input的sequence,才產生出来的

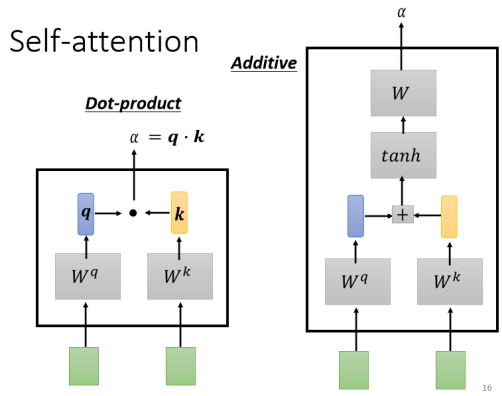
接下来我们来说明一些**怎么产生b^1这个向量**，剩下的b^2 b^3 b^4的产生过程和b^1是完全类似的。

产生b^1这个向量的过程中有一个特别的机制，这个机制是根据a^1这个向量，**找出整个很长的sequence里面哪些部分跟判断a^1是哪一个label是有关系的，哪些部分是我们要决定a^1的class，决定a^1的regression数值的时候，所需要用到的资讯。**每一个向量跟a^1的关联的程度,用一个数值叫α来表示



Find the relevant vectors in a sequence

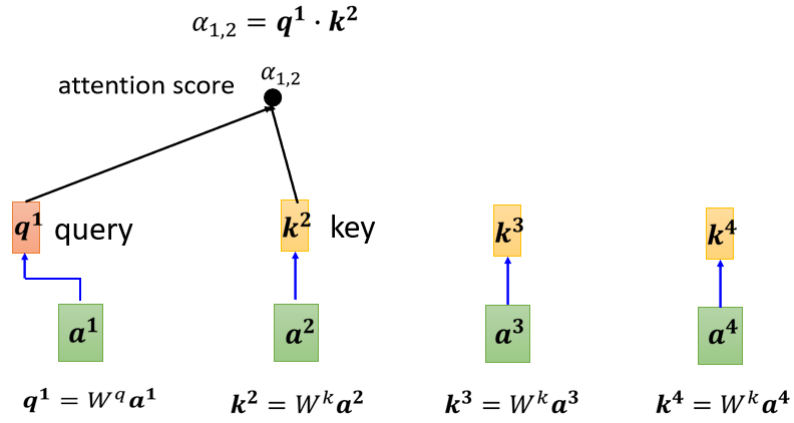
那这个Self-Attention的模块怎么表达两个向量之间的关联性呢，就比如两个向量a^1跟a^4，它怎么决定a^1跟a^4有多相关，然后给它一个数值α呢。这就需要在这个Self-Attention的模块的里面有一个计算attention的模组：



这个计算attention的模组用**两个向量作为输入**，然后直接输出表示两者相关性的α这个数值。而计算这个α的数值有两种比较常见的做法：

- 第一个做法呢叫做dot product，**输入的这两个向量分别乘上两个不同的矩阵**，左边的向量乘上W^q这个矩阵得到矩阵q，右边的向量乘上W^k这个矩阵得到矩阵k，再把q跟k做element-wise 的相乘（对应位置的元素逐个相乘），再全部加起来以后就得到一个数值α。
- 另外一种做法叫做Additive，它同样把输入的两个向量通过W^q W^k得到q跟k，但之后不是把它们逐元素相乘，而是直接把q跟k串起来，然后通过一个Activation Function，最后再通过一个变换得到α。

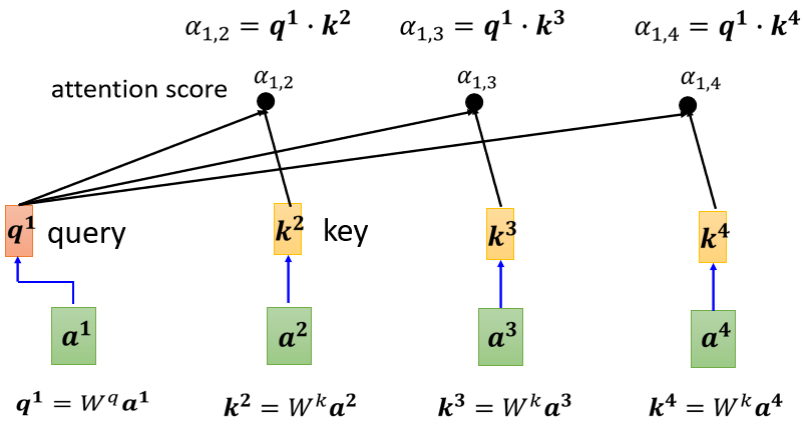
但是在接下来的讨论里，我们都**只用左边这个方法**，这也是今日最常用的方法，也**是用在Transformer里面的方法。**回到刚刚的问题上来，我们就需要把a^1去跟另外的a^2 a^3 a^4，分别去计算他们之间的关联性，也就是计算他们之间的α



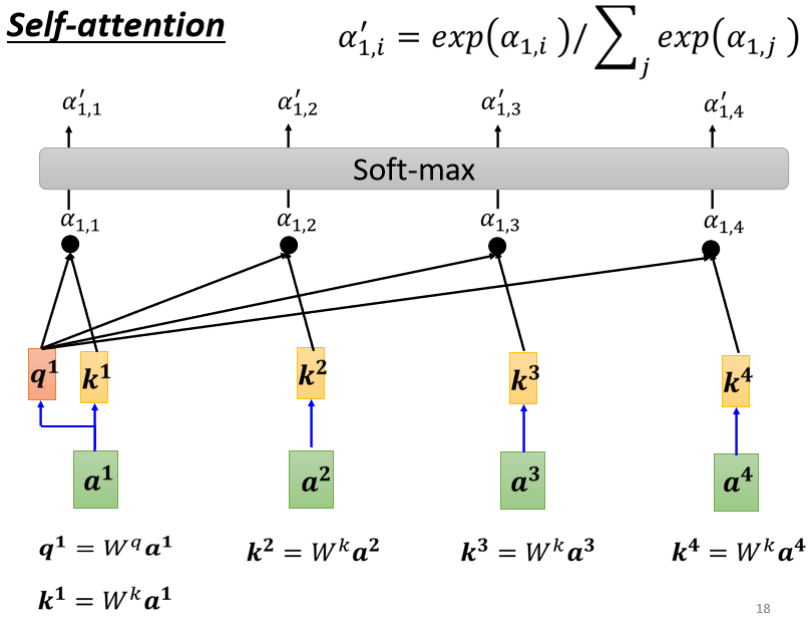
把a^1乘上W^q得到q^1，那这个q有一个名字，我们叫做Query，它就像是你在操作搜索引擎时候搜索的关键字，所叫做Query。

接下来呢，$a^2 a^3 a^4$都要分别乘上W^k，得到k这个Vector，这个k呢我们叫做Key，然后你**把Query q^1跟这个Key k_2逐元素相乘，最后相加就得到α。**我们这里得到的$\alpha_{1,2}$，下标的1表示Query是a^1提供的，下标中的2表示Key是a^2提供的时候，这个$\alpha_{1,2}$也叫做Attention的Score，表示了$a^1 a^2$之间的相关性。

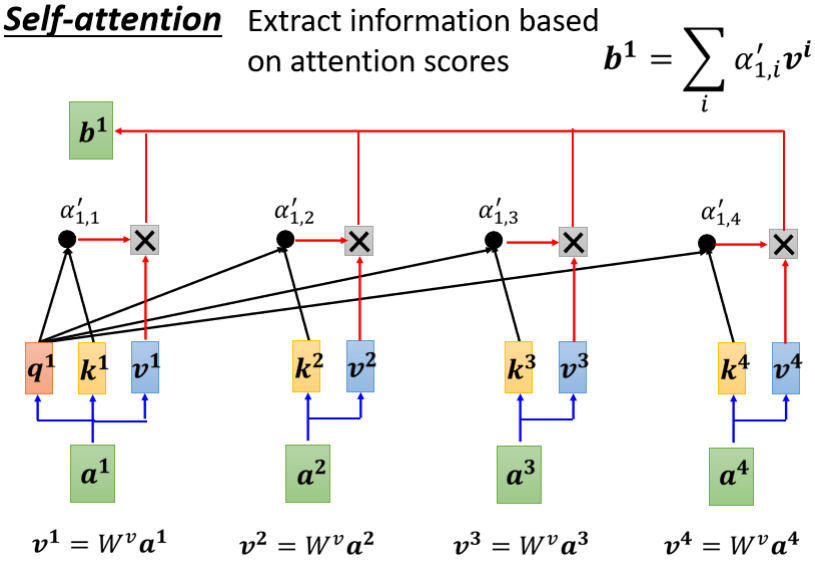
计算完$\alpha_{1,2}$后，a^1接下来也要跟$a^3 a^4$来分别计算Attention Score，分别是$\alpha_{1,3}$，$\alpha_{1,4}$。



一般在实践中，a^1**也会自己跟自己算关联性，这件事情其实是很重要的，不计算的话可能会导致最终的结果有很大的偏差。**在计算完a^1跟每一个向量的关联性以后，最后这些输出会**通过一个SoftMax层。**



这个SoftMax跟分类的时候那个SoftMax是一模一样的，当然这里用ReLU作为激活函数也可以，α通过Soft-Max后就得到了α'，接下来得到这个α'以后，我们就要根据这个α'去抽取出这个Sequence里面重要的资讯，具体要怎么做呢?



- 首先把a^1到a^4的每一个向量都乘上W^v得到新的向量，分别就是用$v^1 v^2 v^3 v^4$来表示。
- 接下来把v^1到v^4的每一个向量都去乘上对应Attention的分数，也就是乘上对应的α'。
- 最后再这些乘出来的结果加起来，就得到b^1

$$b^1 = \sum_i \alpha'_{1,i} v^i$$

事实上谁的Attention的分数最大，谁的v就会主导最终生成的b^1这个结果。因为我们生成的b^1最主要的还是要看a^1，这就是为什么之前提到a^1要和自己算一个Attention的分数，自己跟自己的关联度肯定是最大的，这就保证了a^1的主导地位，再接着考虑其他变量Attention的分数，实质上就是考虑了a^1的上下文。