

# Attention

## 参考文献:

1. Attn: Illustrated Attention : <https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3> (中文翻译版见文献2)
2. Attention 图解: <https://zhuanlan.zhihu.com/p/342235515>
3. Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) (待看)  
<https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>
4. Attention? Attention!: <https://lilianweng.github.io/posts/2018-06-24-attention/>
5. The Illustrated Transformer: <https://jalammar.github.io/illustrated-transformer>

This article will be based on the **seq2seq** framework and how attention can be built on it.

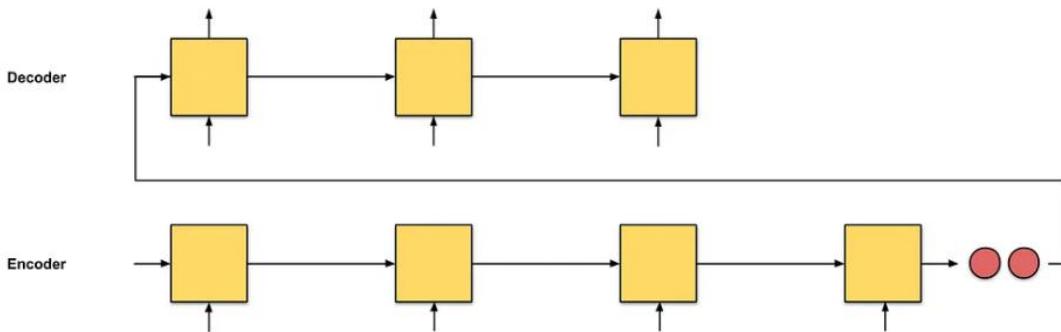


Fig. 0.1: seq2seq with an input sequence of length 4

In **seq2seq**, the idea is to have two recurrent neural networks (RNNs) with an **encoder-decoder architecture**: read the input words one by one to obtain a **vector representation of a fixed dimensionality** (encoder), and, conditioned on these inputs, extract the output words one by one using another RNN (decoder).

The trouble with seq2seq is that the only information that the decoder receives from the encoder is **the last encoder hidden state** (the 2 tiny red nodes in Fig. 0.1), a vector representation that is like a numerical summary of an input sequence. So, **for a long input text** (Fig. 0.2), we unreasonably expect the decoder to use just this one vector representation (hoping that it ‘sufficiently summarises the input sequence’) to output a translation. **This might lead to catastrophic forgetting.**

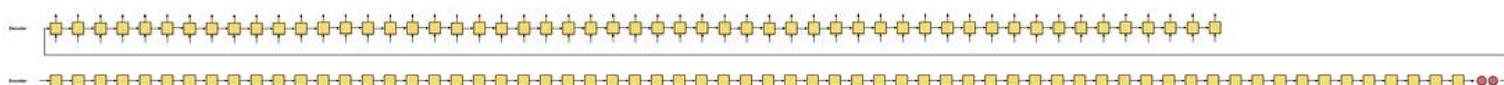


Fig. 0.2: seq2seq with an input sequence of length 64

Can you translate this paragraph to another language you know, right after this question mark?

If we can't, then we shouldn't be so cruel to the decoder. How about instead of just one vector representation, let's give the decoder a vector representation from every encoder time step so that it can make well-informed translations?

Enter **attention**.

**Attention** is an interface between the **encoder** and **decoder** that provides the decoder with **information from every encoder hidden state** (apart from the hidden state in red in Fig. 0.3). With this setting, the model can selectively focus on useful parts of the input sequence and hence, learn the **alignment** between them. This helps the model to cope effectively with long input sentences.

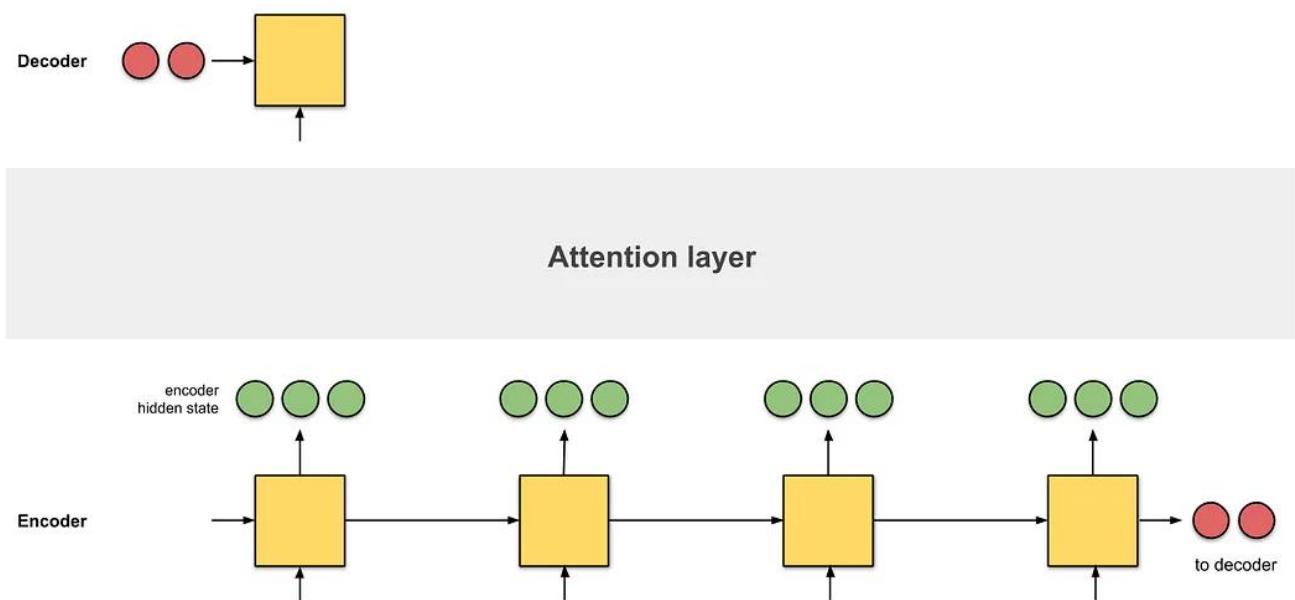


Fig 0.3: Adding an attention layer as an interface between encoder and decoder. Here, the first decoder time step is getting ready to receive information from the encoder before giving the first translated word.

**Alignment** means matching segments of an original text with their corresponding segments of the translation.

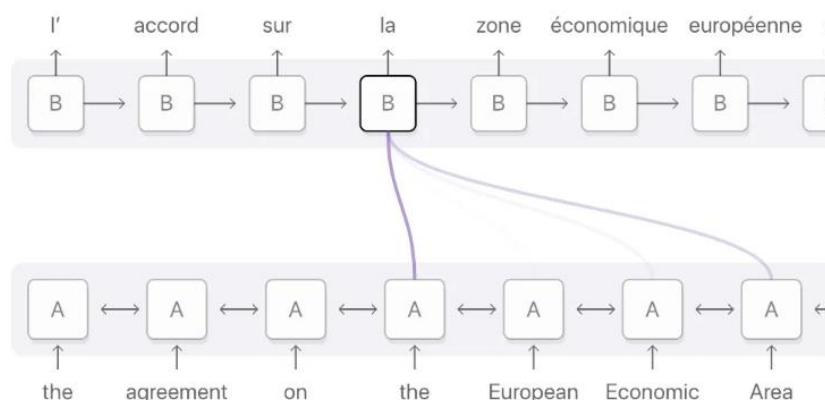


Fig. 0.3: Alignment for the French word 'la' is distributed across the input sequence but mainly on these 4 words: 'the', 'European', 'Economic' and 'Area'. Darker purple indicates better attention scores ([Image source](#))

There are **2 types of attention**:

- The type of attention that uses all the encoder hidden states is also known as **global attention**.
- In contrast, **local attention** uses only a subset of the encoder hidden states.

As the scope of this article is global attention, any references made to “attention” in this article are taken to mean “global attention.”

# 1 Attention: Overview

Before we look at how attention is used, allow me to share with you the intuition behind a translation task using the seq2seq model.

## Intuition: seq2seq

A translator reads the German text from start till the end. Once done, he starts translating to English word by word. It is possible that if the sentence is **extremely long**, he **might have forgotten** what he has read in the earlier parts of the text.

## Intuition: seq2seq + attention

A translator reads the German text while writing down the keywords from the start till the end, after which he starts translating to English. **While translating each German word, he makes use of the keywords he has written down.**

Attention places different focus on different words by assigning each word with a score. Then, using the softmaxed scores, we aggregate the encoder hidden states using a weighted sum of the encoder hidden states, to get the context vector.

The implementations of an attention layer can be broken down into 4 steps.

### Step 0: Prepare hidden states.

- Let's first prepare all the available **encoder hidden states (green)** and **the first decoder hidden state (red)**. In our example, we have 4 encoder hidden states and the current decoder hidden state.
- Note:** the last consolidated encoder hidden state is fed as input to the first time step of the decoder. The output of the first time step of the decoder is called the first decoder hidden state, as seen below.

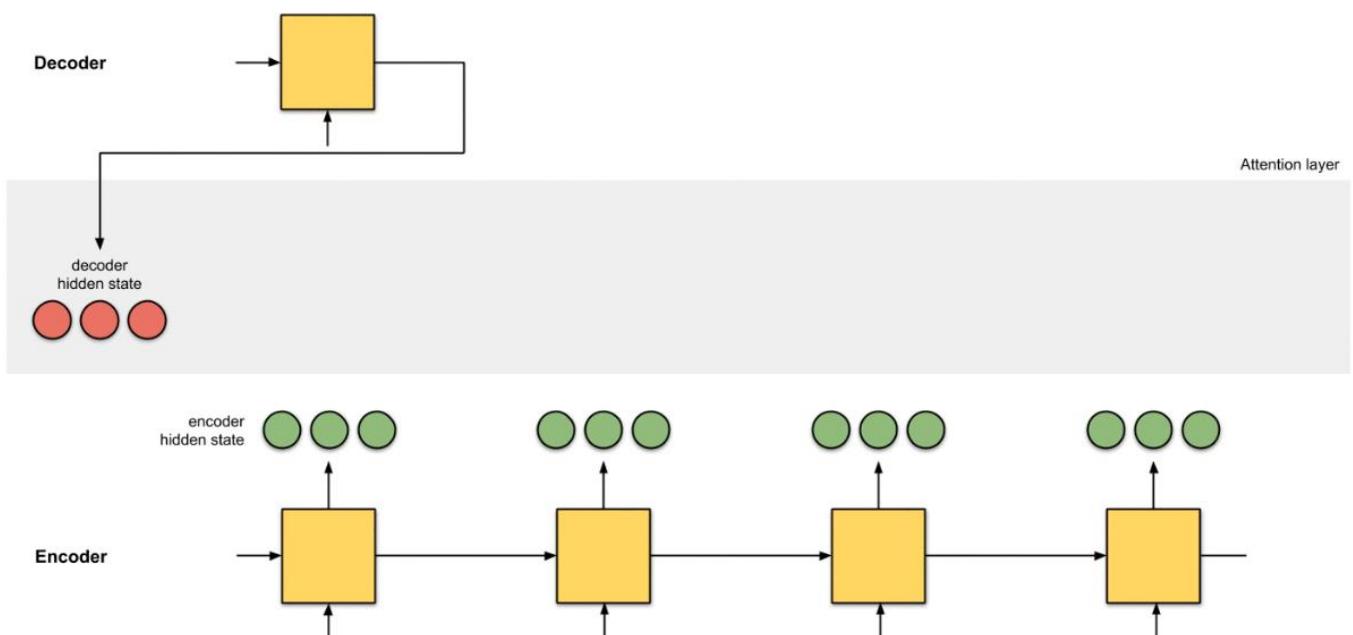


Fig. 1.0: Getting ready to pay attention

## Step 1: Obtain a score for every encoder hidden state.

A score (scalar) is obtained by a score function (also known as the **alignment score** function or **alignment model**). In this example, the score function is a **dot product** between the decoder and encoder hidden states.

See **Appendix A** for a variety of score functions.

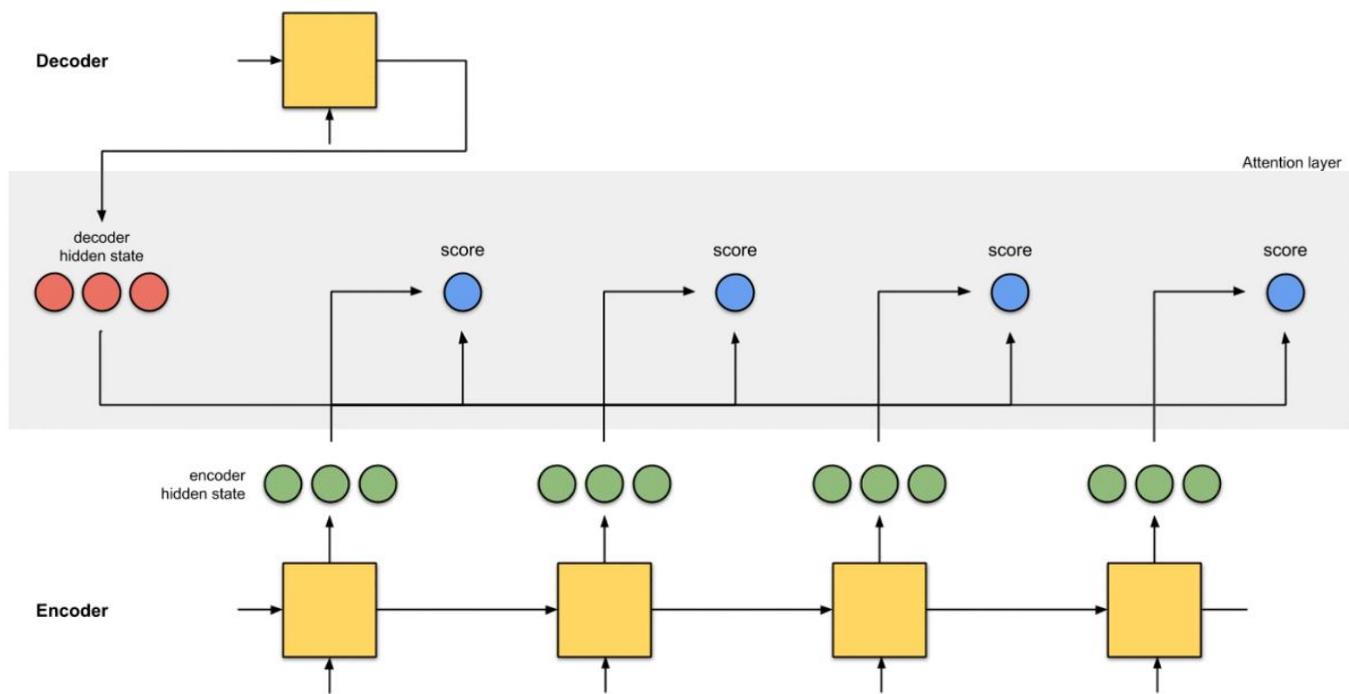


Fig. 1.1: Get the scores

```
decoder_hidden = [10, 5, 10]
encoder_hidden  score
-----
[0, 1, 1]      15 (= 10×0 + 5×1 + 10×1, the dot product)
[5, 0, 1]      60
[1, 1, 0]      15
[0, 5, 1]      35
```

In the above example, we obtain a high attention score of `60` for the encoder hidden state `[5, 0, 1]`. This means that the next word (next output by the decoder) is going to be heavily influenced by this encoder hidden state.

## Step 2: Run all the scores through a softmax layer.

We put the scores to a softmax layer so that the **softmaxed scores (scalar)** add up to 1. These softmaxed scores represent the **attention distribution**.

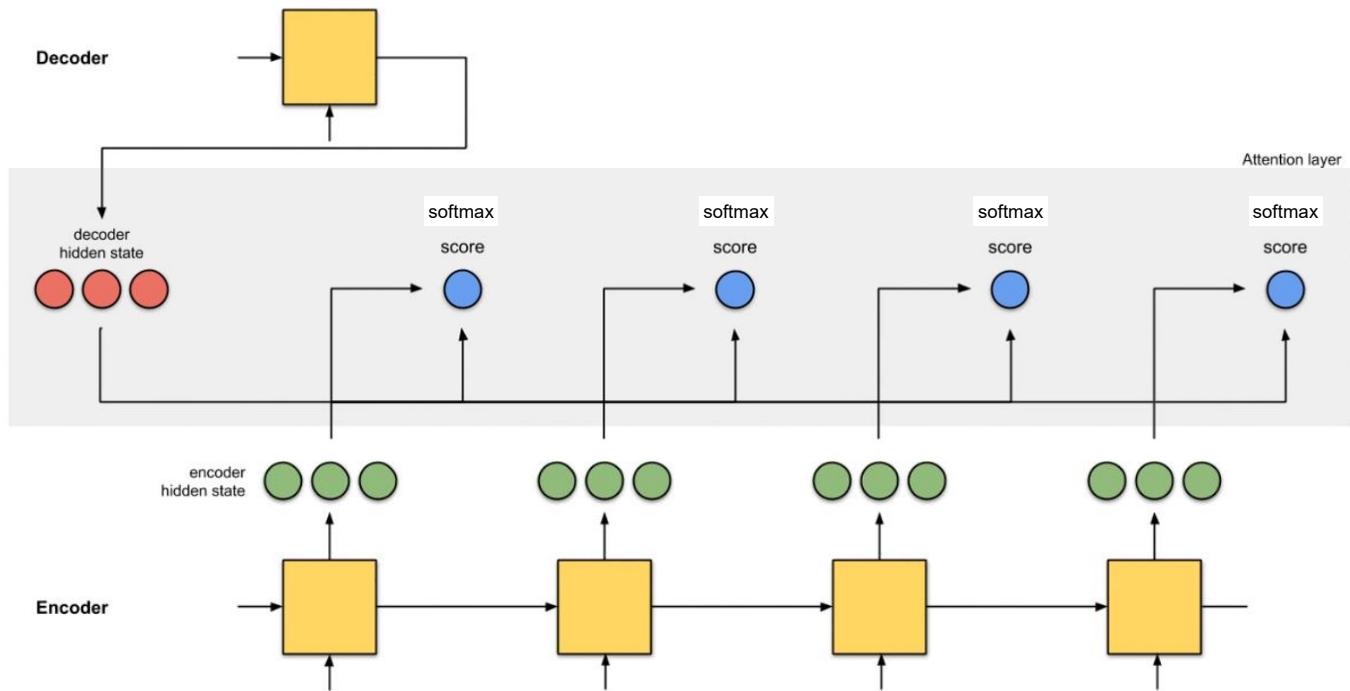


Fig. 1.2: Get the softmaxed scores

encoder_hidden	score	score^
[0, 1, 1]	15	0
[5, 0, 1]	60	1
[1, 1, 0]	15	0
[0, 5, 1]	35	0

Notice that based on the softmaxed score `score^`, the distribution of attention is only placed on `[5, 0, 1]` as expected. In reality, these numbers are not binary but a floating-point between 0 and 1.

### Step 3: Multiply each encoder hidden state by its softmaxed score.

By multiplying **each encoder hidden state** with its **softmaxed score (scalar)**, we obtain the **alignment vector** or the **annotation vector**. This is exactly the mechanism where alignment takes place.

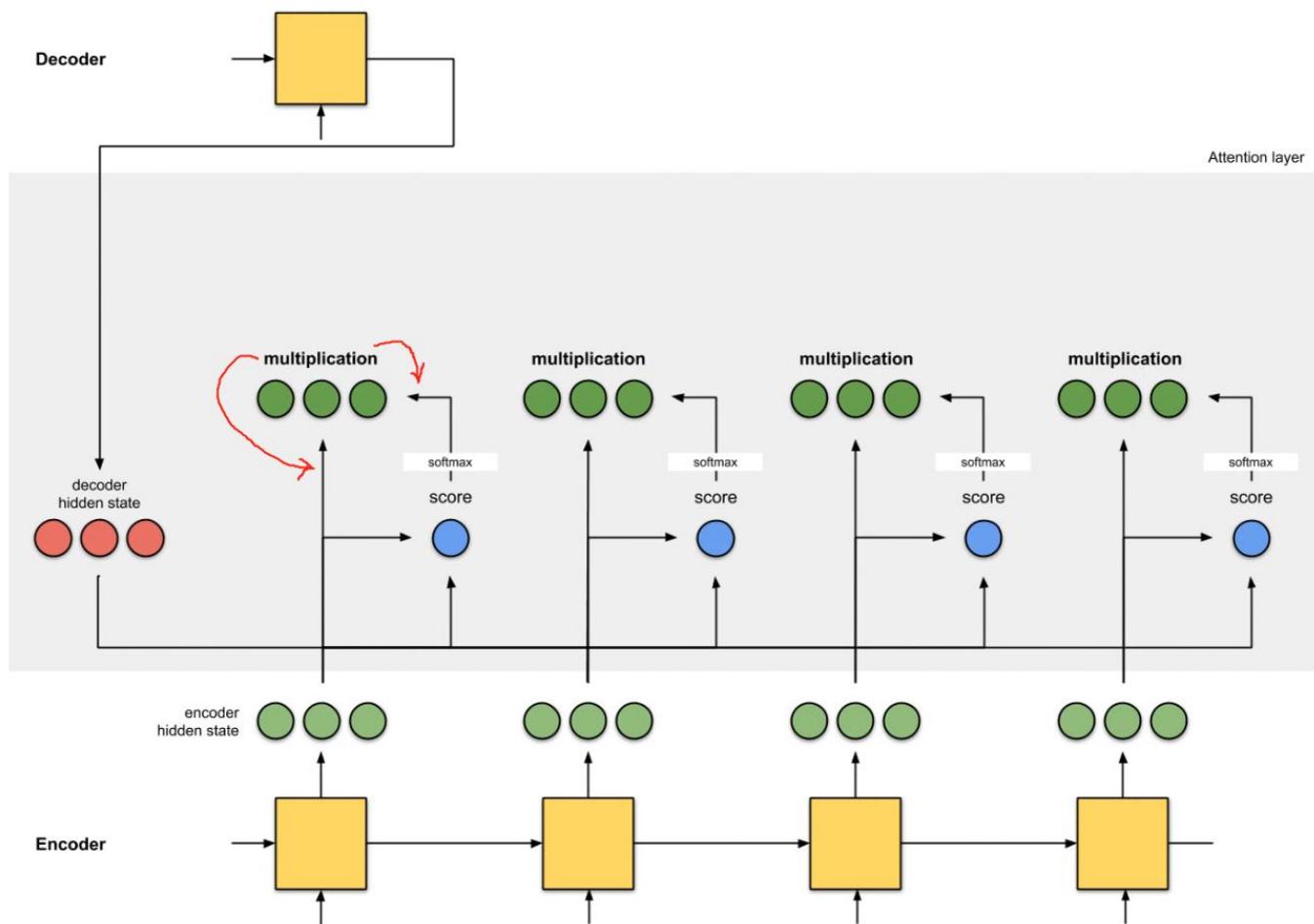


Fig. 1.3: Get the alignment vectors

encoder	score	score <sup>^</sup>	alignment
[0, 1, 1]	15	0	[0, 0, 0]
[5, 0, 1]	60	1	[5, 0, 1]
[1, 1, 0]	15	0	[0, 0, 0]
[0, 5, 1]	35	0	[0, 0, 0]

Here we see that the alignment for all encoder hidden states except [5, 0, 1] are reduced to 0 due to low attention scores. This means we can expect that the first translated word should match the input word with the [5, 0, 1] embedding.

## Step 4: Sum up the alignment vectors.

The alignment vectors are summed up to produce the **context vector** [1, 2]. A context vector is aggregated information of the alignment vectors from the previous step.

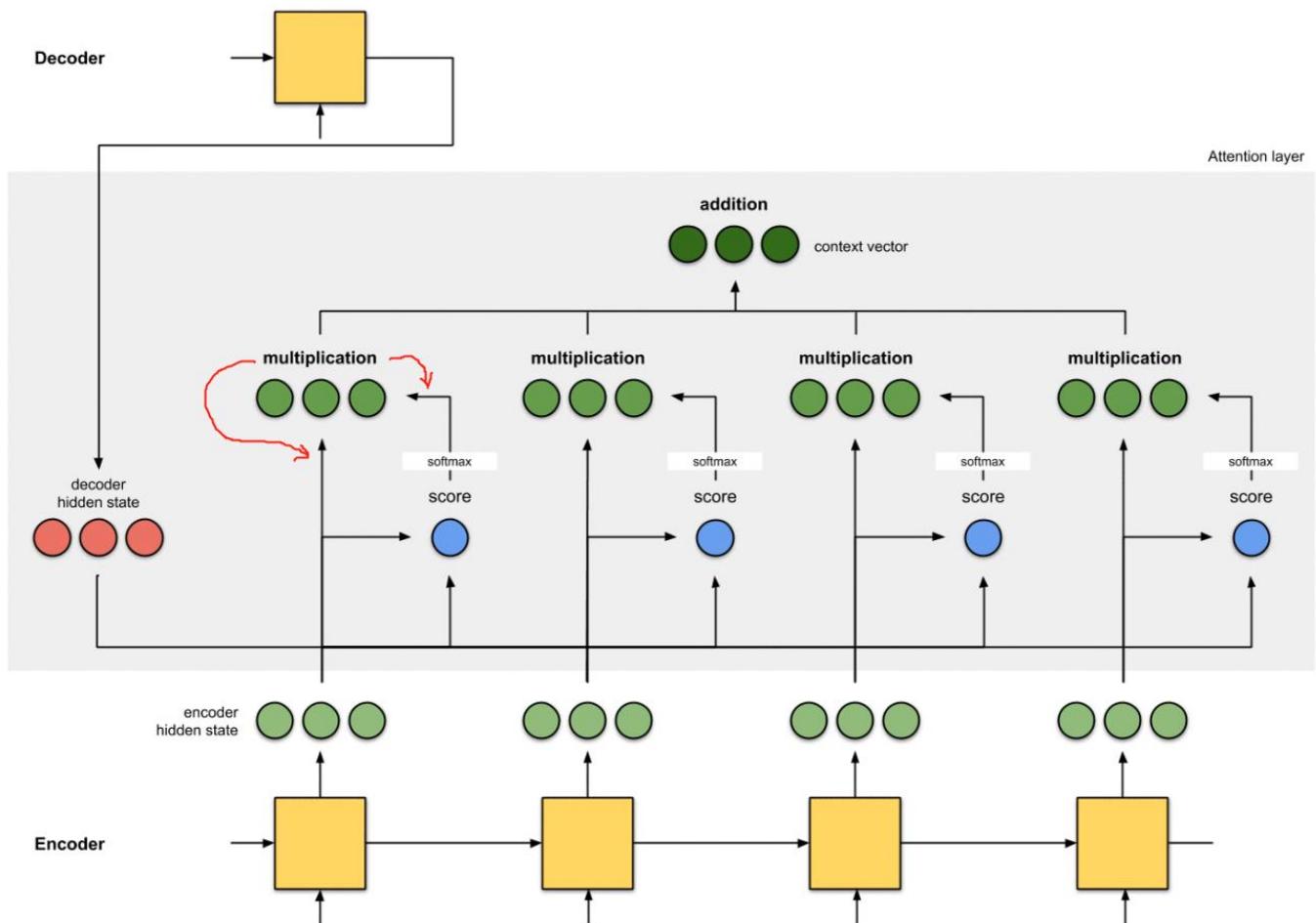


Fig. 1.4: Get the context vector

encoder	score	score <sup>^</sup>	alignment
[0, 1, 1]	15	0	[0, 0, 0]
[5, 0, 1]	60	1	[5, 0, 1]
[1, 1, 0]	15	0	[0, 0, 0]
[0, 5, 1]	35	0	[0, 0, 0]

**context** = [0+5+0+0, 0+0+0+0, 0+1+0+0] = [5, 0, 1]

## Step 5: Feed the context vector into the decoder.

The manner this is done depends on the architecture design. Later we will see in the examples in Sections 2a, 2b and 2c how the architectures make use of the context vector for the decoder.

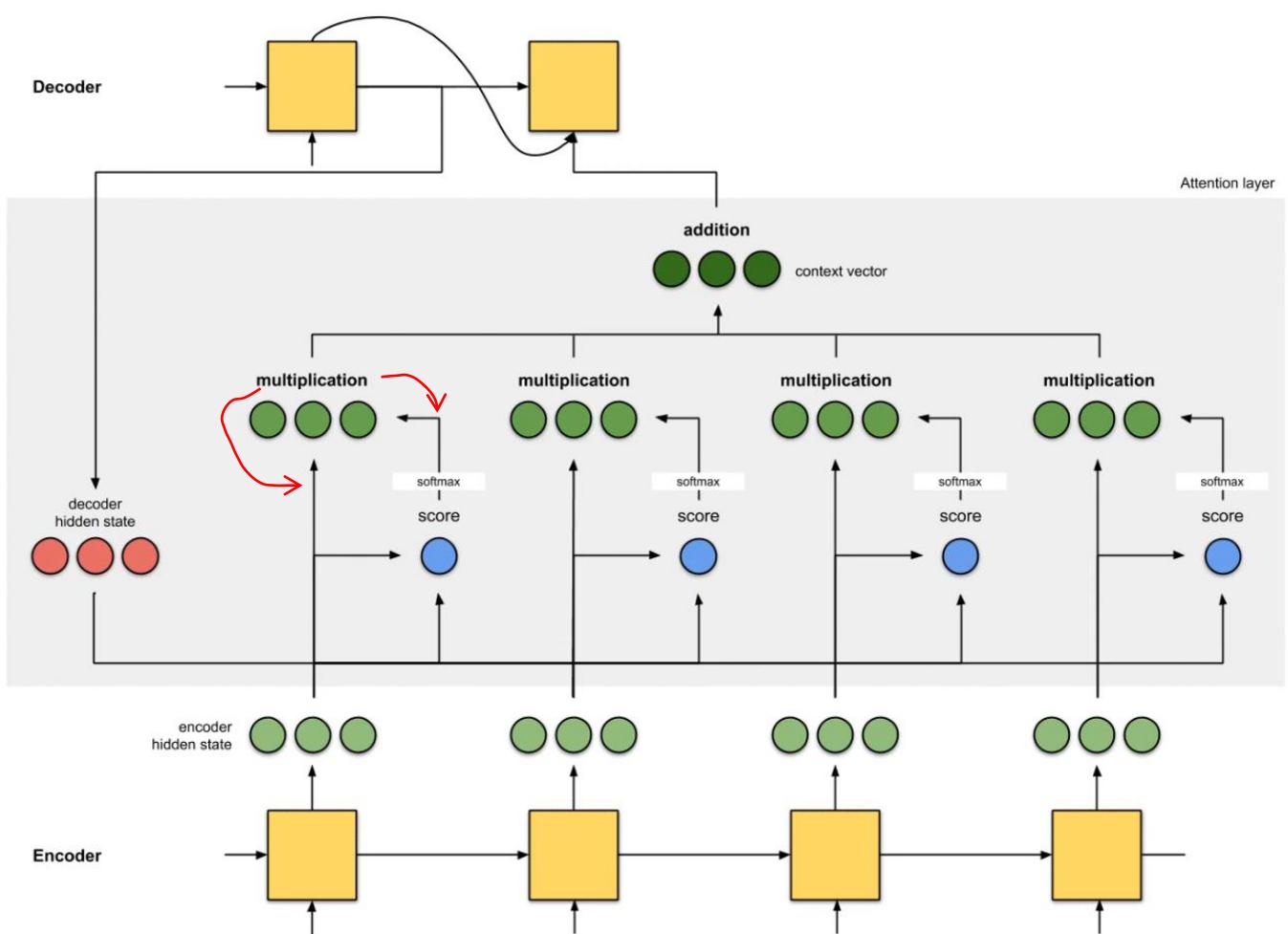


Fig. 1.5: Feed the context vector to the decoder

That's about it! Here's the entire animation:



Fig. 1.6: Attention

## Training and inference

- During **inference**, the input to each decoder time step  $t$  is the **predicted output** from decoder time step  $t-1$ .
- During **training**, the input to each decoder time step  $t$  is our **ground truth output** from decoder time step  $t-1$ .

## Intuition: How does attention actually work?

Answer: **Backpropagation**, surprise surprise. Backpropagation will do whatever it takes to ensure that the outputs will be close to the ground truth. This is done by altering the weights in the RNNs and the score function (if any). These weights will affect the encoder hidden states and decoder hidden states, which in turn affect the attention scores.

## 2. Attention: Examples

We have seen both the seq2seq and the seq2seq+attention architectures in the previous section. In the next sub-sections, let's examine 3 more seq2seq-based architectures for NMT that implement attention. For completeness, I have also appended their **Bilingual Evaluation Understudy (BLEU) scores** — a standard metric for evaluating a generated sentence to a reference sentence.

(此部分略，见原链接)

## Appendix: Score Functions

Below are some of the score functions as compiled by Lilian Weng. The score functions **additive/concat** and **dot product** have been mentioned in this article.

- The idea behind score functions involving the **dot product operation (dot product, cosine similarity etc.)**, is to **measure the similarity** between two vectors.
- For **feed-forward neural network score functions**, the idea is to let the model **learn the alignment weights** together with the translation.

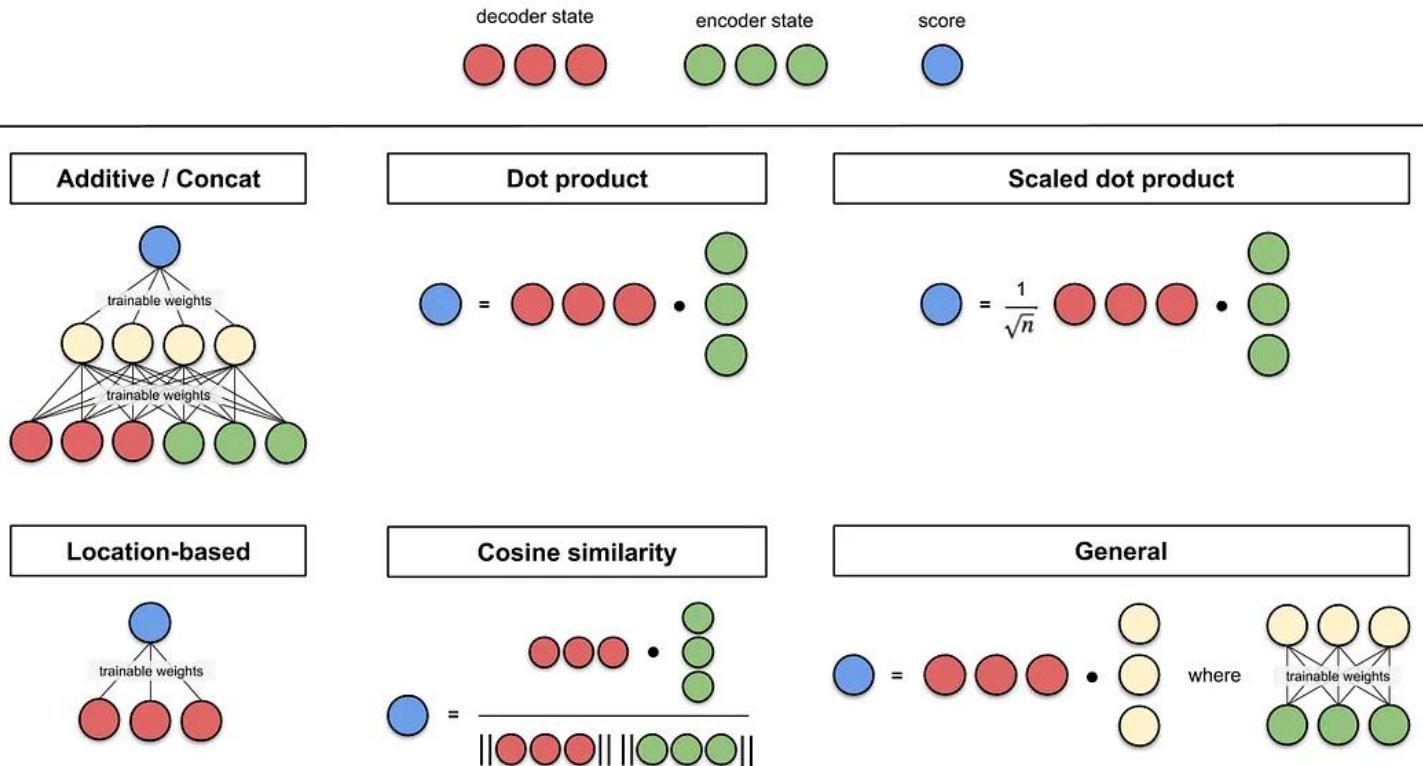


Fig. A0: Summary of score functions

Name	Alignment score function	Citation
Content-base attention	$\text{score}(s_t, h_i) = \text{cosine}[s_t, h_i]$	Graves2014
Additive(*)	$\text{score}(s_t, h_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[s_t; h_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a s_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(s_t, h_i) = s_t^\top \mathbf{W}_a h_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(s_t, h_i) = s_t^\top h_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(s_t, h_i) = \frac{s_t^\top h_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Fig. A1: Summary of score functions.  $h$  represents encoder hidden states while  $s$  represents decoder hidden states.

## 参考文献：

1. 深度学习中的注意力机制(2017版)：<https://blog.csdn.net/malefactor/article/details/78767781>
2. 自然语言处理中的Attention Model：是什么及为什么：<https://blog.csdn.net/malefactor/article/details/50550211>

**注意力模型 (Attention Model)** 被广泛使用在自然语言处理、图像识别及语音识别等各种不同类型的深度学习任务中，是深度学习技术中最值得关注与深入了解的核心技术之一。

本文以机器翻译为例，深入浅出地介绍了深度学习中注意力机制的原理及关键计算机制，同时也抽象出其本质思想，并介绍了注意力模型在图像及语音等领域的典型应用场景。

## 1 人类的视觉注意力

..... 我们首先简单介绍人类视觉的选择性注意力机制。

视觉注意力机制是人类视觉所特有的大脑信号处理机制。人类视觉通过快速扫描全局图像，获得需要重点关注的目标区域，也就是一般所说的注意力焦点，而后对这一区域投入更多注意力资源，以获取更多所需要关注目标的细节信息，而抑制其他无用信息。

这是人类利用有限的注意力资源从大量信息中快速筛选出高价值信息的手段，人类视觉注意力机制极大地提高了视觉信息处理的效率与准确性。

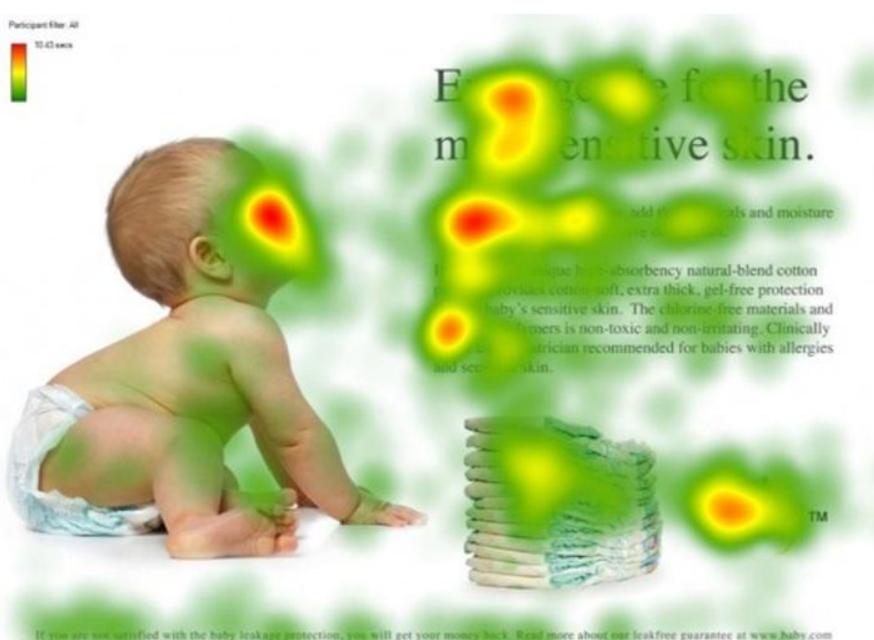


图1形象化展示了人类在看到一副图像时是如何高效分配有限的注意力资源的，其中红色区域表明视觉系统更关注的目标，很明显对于图1所示的场景，人们会把注意力更多投入到人的脸部，文本的标题以及文章首句等位置。

图1 人类的视觉注意力

**深度学习中的注意力机制**从本质上讲和人类的选择性视觉注意力机制类似，核心目标也是**从众多信息中选择出对当前任务目标更关键的信息**。

## 2 Encoder-Decoder 框架

要了解深度学习中的注意力模型，就不得不先谈 Encoder-Decoder 框架，因为目前大多数注意力模型附着在 Encoder-Decoder 框架下，当然，其实注意力模型可以看作一种通用的思想，本身并不依赖于特定框架，这点需要注意。

Encoder-Decoder 框架可以看作是一种深度学习领域的研究模式，应用场景异常广泛。图2是文本处理领域里常用的 Encoder-Decoder 框架最抽象的一种表示。

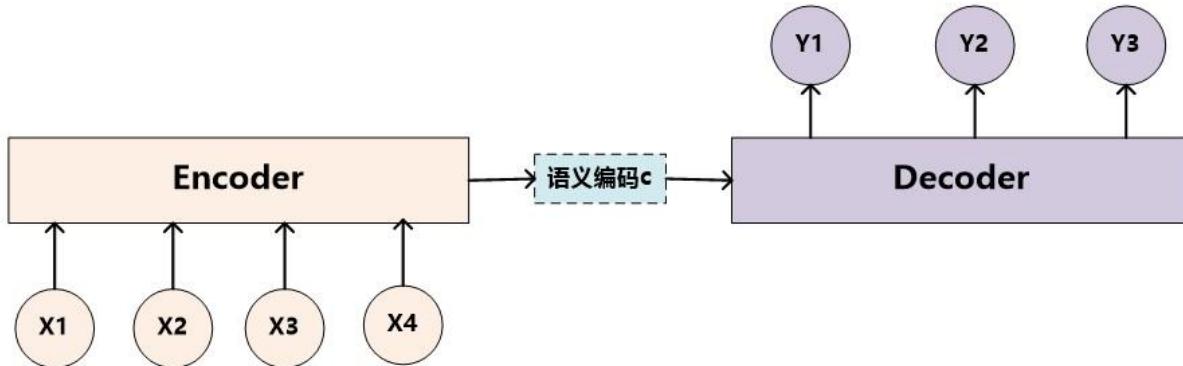


图2 抽象的文本处理领域的Encoder-Decoder框架

文本处理领域的Encoder-Decoder框架可以这么直观地去理解：可以把它看作适合处理由一个句子（或篇章）生成另外一个句子（或篇章）的通用处理模型。对于句子对 <Source, Target>，我们的目标是给定输入句子 Source，期待通过 Encoder-Decoder 框架来生成目标句子 Target。Source 和 Target 可以是同一种语言，也可以是两种不同的语言。而 Source 和 Target 分别由各自的单词序列构成：

$$\begin{aligned}\text{Source} &= \langle x_1, x_2 \dots x_m \rangle \\ \text{Target} &= \langle y_1, y_2 \dots y_n \rangle\end{aligned}$$

Encoder 顾名思义就是对输入句子 Source 进行编码，将输入句子通过非线性变换转化为中间语义表示 C：

$$C = \mathcal{F}(x_1, x_2 \dots x_m)$$

对于解码器 Decoder 来说，其任务是根据句子 Source 的中间语义表示 C 和之前已经生成的历史信息来生成 i 时刻要生成的单词  $y_i$ ：

$$y_i = \mathcal{G}(C, y_1, y_2 \dots y_{i-1})$$

每个  $y_i$  都依次这么产生，那么看起来就是整个系统根据输入句子 Source 生成了目标句子 Target。

- 如果 Source 是中文句子，Target 是英文句子，那么这就是解决机器翻译问题的 Encoder-Decoder 框架；
- 如果 Source 是一篇文章，Target 是概括性的几句描述语句，那么这是文本摘要的 Encoder-Decoder 框架；
- 如果 Source 是一句问句，Target 是一句回答，那么这是问答系统或者对话机器人的 Encoder-Decoder 框架。

由此可见，在文本处理领域，Encoder-Decoder 的应用领域相当广泛。

Encoder-Decoder 框架不仅仅在文本领域广泛使用，在语音识别、图像处理等领域也经常使用。

- 对于语音识别，图2所示的框架完全适用，区别无非是 Encoder 部分的输入是语音流，输出是对应的文本信息；
- 对于“图像描述”任务来说，Encoder 部分的输入是一副图片，Decoder 的输出是能够描述图片语义内容的一句描述语

一般而言，文本处理和语音识别的 Encoder 部分通常采用 RNN 模型，图像处理的 Encoder 一般采用 CNN 模型。

## 3 Attention 模型

### 3.1 Soft Attention模型

图2中展示的 Encoder-Decoder框架是没有体现出“注意力模型”的，所以可以把它看作是注意力不集中的分心模型。为什么说它注意力不集中呢？请观察下目标句子 Target 中每个单词的生成过程如下：

$$\begin{aligned}y_1 &= f(C) \\y_2 &= f(C, y_1) \\y_3 &= f(C, y_1, y_2)\end{aligned}$$

其中  $f$  是 Decoder 的非线性变换函数。从这里可以看出，在生成目标句子的单词时，不论生成哪个单词，它们使用的输入句子 Source 的语义编码  $C$  都是一样的，没有任何区别。而语义编码  $C$  是由句子 Source 的每个单词经过 Encoder 编码产生的，这意味着不论是生成哪个单词， $y_1$ 、 $y_2$  还是  $y_3$ ，其实句子 Source 中任意单词对生成某个目标单词  $y_i$  来说影响力都是相同的，这是为何说这个模型没有体现出注意力的缘由。这类似于人类看到眼前的画面，但是眼中却没有注意焦点一样。（此处省略一个将“Tom chase Jerry”翻译成中文的例子）

没有引入注意力的模型在输入句子比较短的时候问题不大，但是如果输入句子比较长，此时所有语义完全通过一个中间语义向量来表示，单词自身的信息已经消失，可想而知会丢失很多细节信息，这也是为何要引入注意力模型的重要原因。

同理，目标句子中的每个单词都应该学会其对应的源语句子中单词的注意力分配概率信息。这意味着在生成每个单词的时候，原先都是相同的中间语义表示  $C$  会被替换成根据当前生成单词而不断变化的。理解Attention模型的关键就是这里，即由固定的中间语义表示  $C$  换成了根据当前输出单词来调整成加入注意力模型的变化的。增加了注意力模型的 Encoder-Decoder 框架理解起来如图3所示。

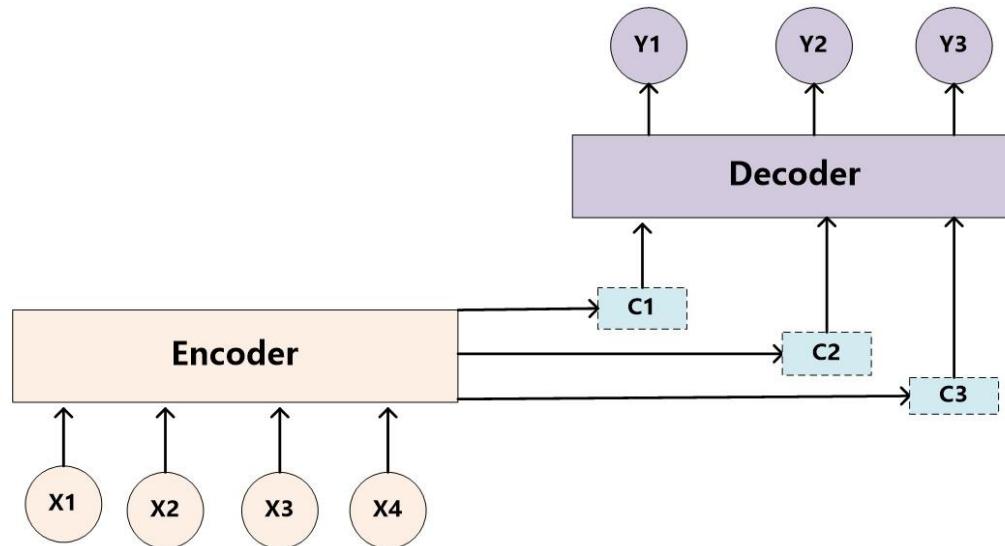


图3 引入注意力模型的Encoder-Decoder框架

即生成目标句子单词的过程成了下面的形式：

$$\begin{aligned}y_1 &= f_1(C_1) \\y_2 &= f_1(C_2, y_1) \\y_3 &= f_1(C_3, y_1, y_2)\end{aligned}$$

这里还有一个问题：生成目标句子某个单词，如何知道Attention模型所需要的输入句子单词注意力分配概率分布值呢？比如生成“汤姆”的时候，对应的输入句子Source中各个单词的概率分布：(Tom,0.6) (Chase,0.2) (Jerry,0.2) 是如何得到的呢？

为了便于说明，我们假设对图2的非Attention模型的 Encoder-Decoder框架 进行细化，Encoder 采用 RNN模型，Decoder 也采用 RNN模型，这是比较常见的一种模型配置，则图2的框架转换为图5。

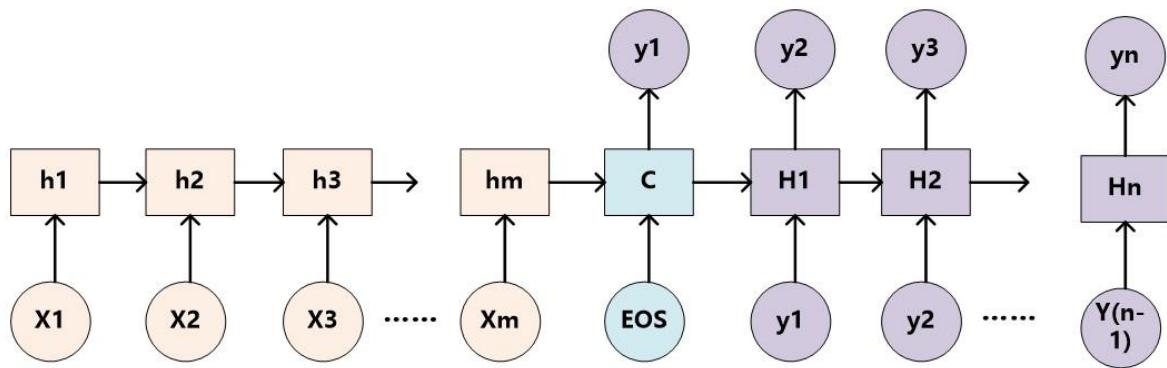


图5 RNN作为具体模型的Encoder-Decoder框架

那么用图6可以较为便捷地说明注意力分配概率分布值的通用计算过程。

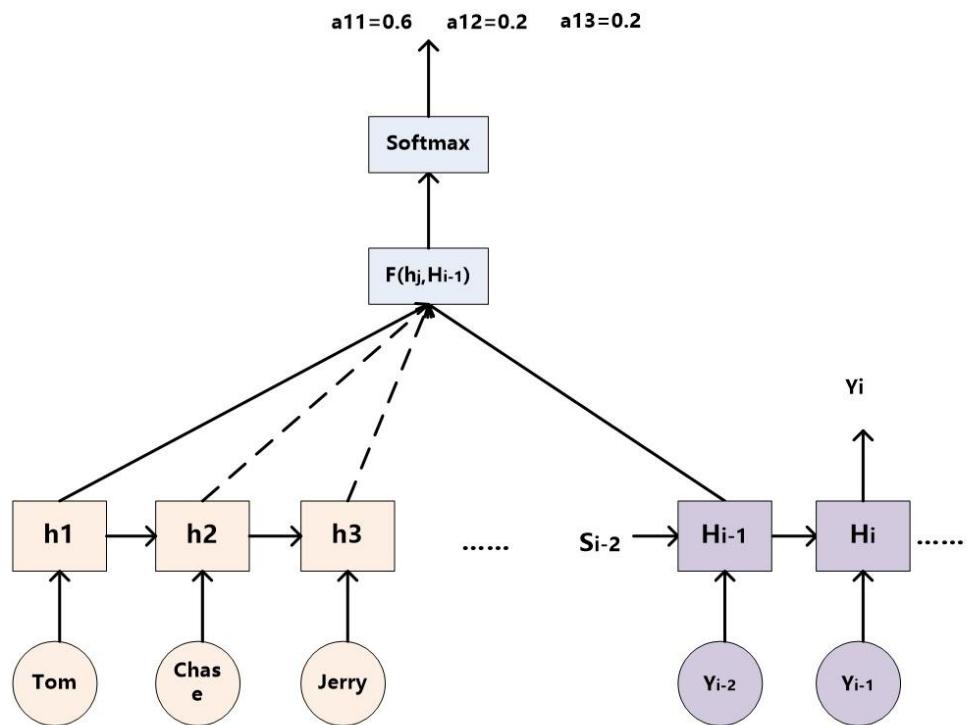


图6 注意力分配概率计算

对于采用 RNN 的 Decoder 来说，在时刻  $i$ ，如果要生成  $y_i$  单词，我们是可以知道 Target 在生成之前的时刻  $i-1$  时，隐层节点  $i-1$  时刻的输出值  $H_{i-1}$  的，而我们的目的是要计算生成时输入句子中的单词“Tom”、“Chase”、“Jerry”对来说的注意力分配概率分布，那么可以用 Target 输出句子  $i-1$  时刻的隐层节点状态  $H_{i-1}$  去——和输入句子Source中每个单词对应的 RNN 隐层节点状态  $h_j$  进行对比，即通过函数F( $h_j, H_{i-1}$ )来获得目标单词和每个输入单词对应的对齐可能性，这个 F 函数在不同论文里可能会采取不同的方法，然后函数 F 的输出经过 Softmax 进行归一化就得到了符合概率分布取值区间的注意力分配概率分布数值。

绝大多数 Attention 模型都是采取上述计算框架来计算注意力分配概率分布信息，区别只是在 F 的定义上可能有所不同。

上述内容就是经典的 Soft Attention 模型的基本思想，那么怎么理解Attention模型的物理含义呢？

一般在自然语言处理应用里会把 Attention 模型看作是输出 Target 句子中某个单词和输入 Source 句子每个单词的对齐模型，这是非常有道理的。

目标句子生成的每个单词对应输入句子单词的概率分布可以理解为输入句子单词和这个目标生成单词的对齐概率，这在机器翻译语境下是非常直观的：传统的统计机器翻译一般在做的过程中会专门有一个短语对齐的步骤，而注意力模型其实起的是相同的作用。

### 3.2 Attention 机制的本质思想

如果把 Attention 机制从上文讲述例子中的 Encoder-Decoder 框架中剥离，并进一步做抽象，可以更容易看懂 Attention 机制的本质思想。

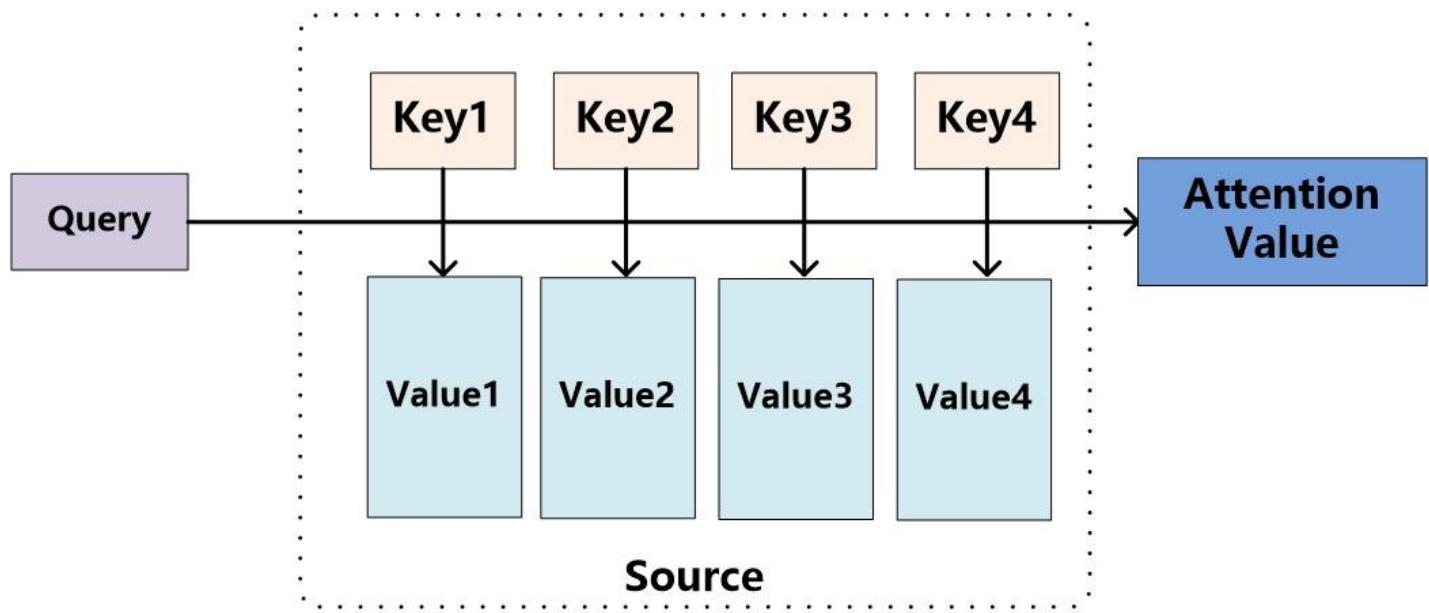


图9 Attention机制的本质思想

我们可以这样来看待 Attention 机制（参考图9）：将 Source 中的构成元素想象成是由一系列的  $\langle \text{Key}, \text{Value} \rangle$  数据对构成，此时给定 Target 中的某个元素 Query，通过计算 Query 和各个 Key 的相似性或者相关性，得到每个 Key 对应 Value 的权重系数，然后对 Value 进行加权求和，即得到了最终的 Attention 数值。所以本质上 Attention 机制是对 Source 中元素的 Value 值进行加权求和，而 Query 和 Key 用来计算对应 Value 的权重系数。

即可以将其本质思想改写为如下公式：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} \text{Similarity}(\text{Query}, \text{Key}_i) * \text{Value}_i$$

其中， $L_x = ||\text{Source}||$  代表 Source 的长度，公式含义即如上所述。上文所举的机器翻译的例子，因为在计算 Attention 的过程中，Source 中的 Key 和 Value 合二为一，指向的是同一个东西，也即输入句子中每个单词对应的语义编码，所以可能不容易看出这种能够体现本质思想的结构。

当然，从概念上理解，把 Attention 仍然理解为从大量信息中有选择地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，这种思路仍然成立。聚焦的过程体现在权重系数的计算上，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。

至于 Attention 机制的具体计算过程，如果对目前大多数方法进行抽象的话，可以将其归纳为两个过程：

- 第一个过程是根据 Query 和 Key 计算权重系数。
- 第二个过程根据权重系数对 Value 进行加权求和。

而第一个过程又可以细分为两个阶段：

- 第一个阶段根据 Query 和 Key 计算两者的相似性或者相关性。
- 第二个阶段对第一阶段的原始分值进行归一化处理。

这样，可以将 Attention 的计算过程抽象为如图10展示的三个阶段。

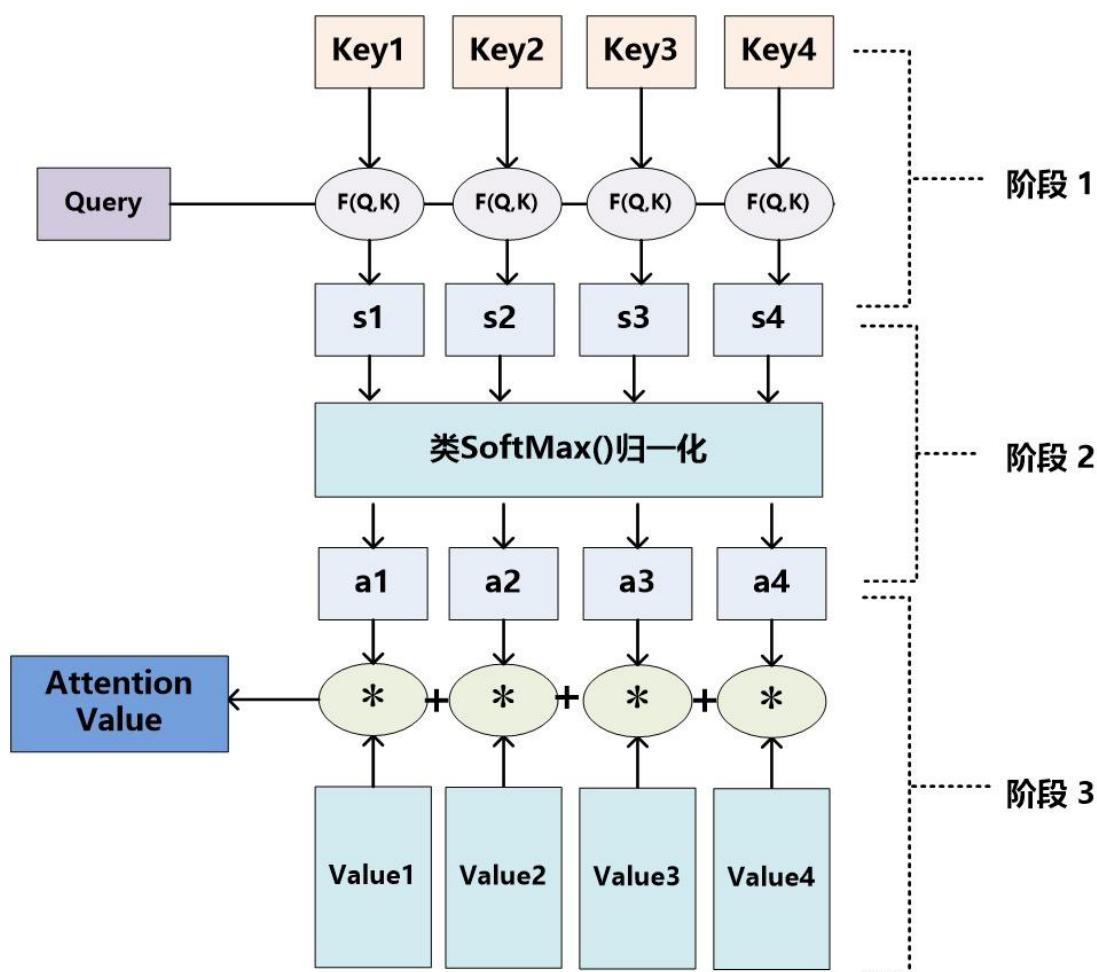


图10 三阶段计算Attention过程

在第一个阶段，可以引入不同的函数和计算机制，根据Query和某个 $Key_i$ ，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，即如下方式：

$$\text{点积: } \mathbf{Similarity}(\mathbf{Query}, \mathbf{Key}_i) = \mathbf{Query} \cdot \mathbf{Key}_i$$

$$\text{Cosine 相似性: } \mathbf{Similarity}(\mathbf{Query}, \mathbf{Key}_i) = \frac{\mathbf{Query} \cdot \mathbf{Key}_i}{\|\mathbf{Query}\| \cdot \|\mathbf{Key}_i\|}$$

$$\text{MLP 网络: } \mathbf{Similarity}(\mathbf{Query}, \mathbf{Key}_i) = \mathbf{MLP}(\mathbf{Query}, \mathbf{Key}_i)$$

第二阶段引入类似 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。

即一般采用如下公式计算：

$$a_i = \text{Softmax}(Sim_i) = \frac{e^{Sim_i}}{\sum_{j=1}^{L_x} e^{Sim_j}}$$

第三阶段，上一步的计算结果  $a_i$  即为  $\text{Value}_i$  对应的权重系数，然后进行加权求和即可得到 Attention 数值：

$$\text{Attention}(\text{Query}, \text{Source}) = \sum_{i=1}^{L_x} a_i \cdot \text{Value}_i$$

通过如上三个阶段的计算，即可求出针对 Query 的 Attention 数值，目前绝大多数具体的注意力机制计算方法都符合上述的三阶段抽象计算过程。

### 3.3 Self Attention模型

Self Attention 也经常被称为 intra Attention (内部Attention) 。

- 在一般任务的 Encoder-Decoder 框架中，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。
- 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。其具体计算过程是一样的，只是计算对象发生了变化而已。

如果是常规的 Target 不等于 Source 情形下的注意力计算，其物理含义正如上文所讲，比如对于机器翻译来说，本质上是目标语单词和源语单词之间的一种单词对齐机制。

那么如果是 Self Attention 机制，一个很自然的问题是：通过 Self Attention 到底学到了哪些规律或者抽取出了哪些特征呢？或者说引入 Self Attention 有什么增益或者好处呢？

我们仍然以机器翻译中的 Self Attention 来说明，图11和图12是可视化地表示 Self Attention 在同一个英语句子内单词间产生的联系。

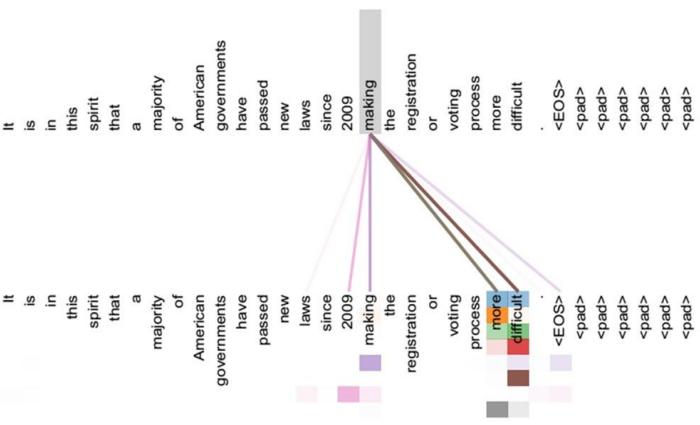


图11 可视化Self Attention实例

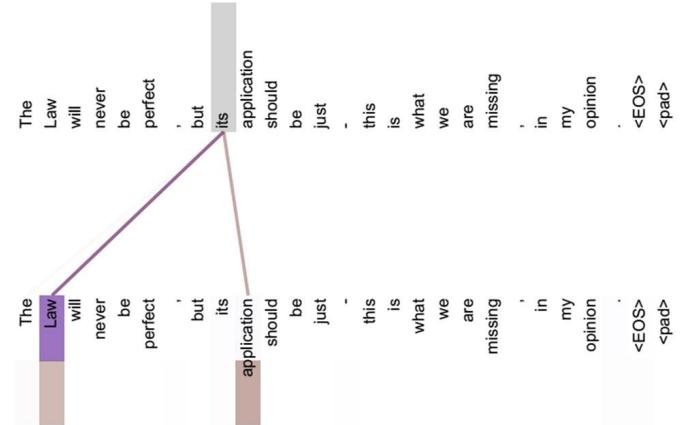


图12 可视化Self Attention实例

从两张图（图11、图12）可以看出，**Self Attention** 可以捕获同一个句子中单词之间的一些**句法特征**（比如图11展示的有一定距离的短语结构）或者**语义特征**（比如图12展示的 its 的指代对象 Law）。

很明显，引入**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。

但是 **Self Attention** 在计算过程中会直接将句子中任意两个单词的联系通过一个计算步骤直接联系起来，所以远距离依赖特征之间的距离被极大缩短，有利于有效地利用这些特征。除此外，**Self Attention** 对于增加计算的并行性也有直接帮助作用。这是为何 **Self Attention** 逐渐被广泛使用的主要原因。

### 3.4 Attention机制的应用

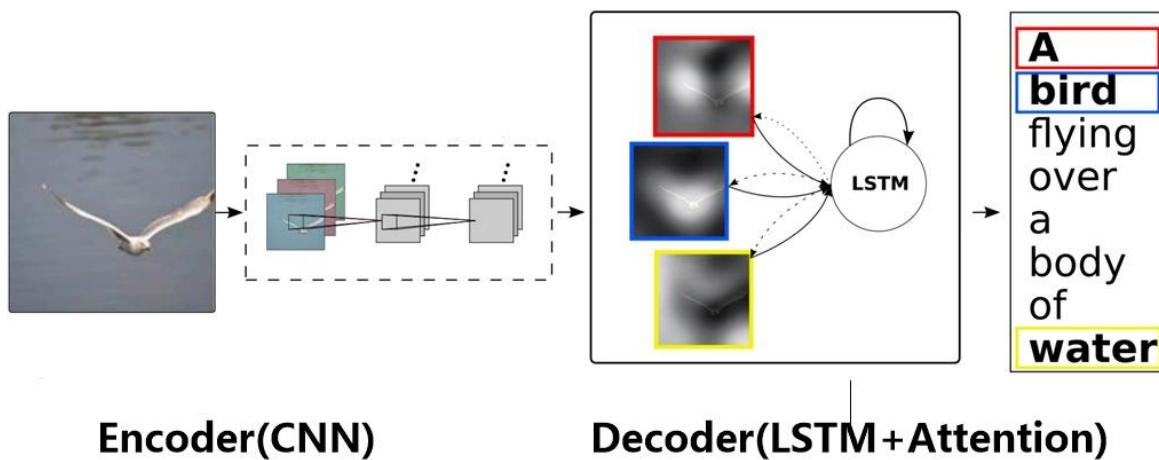


图13 图片描述任务的Encoder-Decoder框架

图片描述 (Image-Caption) 是一种典型的图文结合的深度学习应用，输入一张图片，人工智能系统输出一句描述句子，语义等价地描述图片所示内容。这种应用场景也可以使用Encoder-Decoder框架来解决任务目标，此时Encoder输入部分是一张图片，一般会用CNN来对图片进行特征抽取，Decoder部分使用RNN或者LSTM来输出自然语言句子（图13）。

(详见原文章)

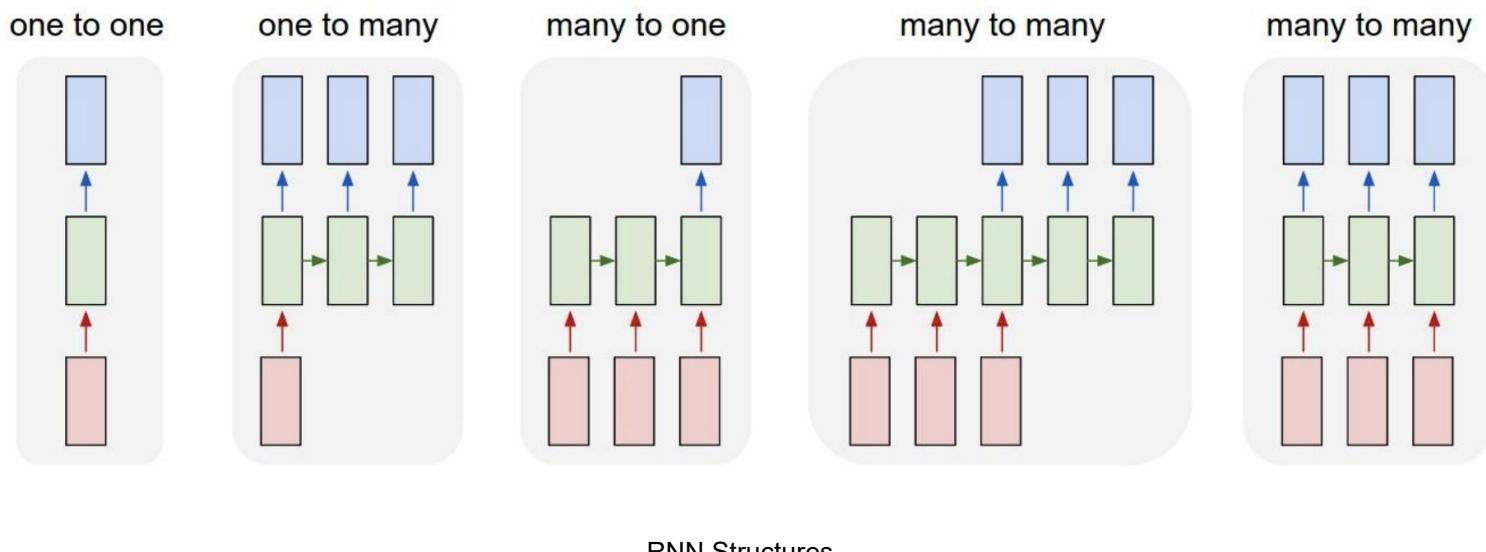
## 参考文献:

1. Seq2Seq模型和Attention机制: <https://luweikxy.gitbook.io/machine-learning-notes/seq2seq-and-attention-mechanism>
2. 论文: [NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE](https://arxiv.org/pdf/1409.0473.pdf)  
<https://arxiv.org/pdf/1409.0473.pdf>
3. 从seq2seq到谷歌BERT, 浅谈对Attention Mechanism的理解: [https://blog.csdn.net/weixin\\_39671140/article/details/88240612](https://blog.csdn.net/weixin_39671140/article/details/88240612)

## 1 RNN的多种结构

先从RNN的结构说起, 根据输出和输入序列不同数量RNN可以有多种不同的结构, 不同结构有不同的应用场合。

- **one to one** 结构, 仅仅只是简单的给一个输入得到一个输出, 此处并未体现序列的特征, 例如**图像分类**场景。
- **one to many** 结构, 给一个输入得到一系列输出, 这种结构可用于**生产图片描述**的场景。
- **many to one** 结构, 给一系列输入得到一个输出, 这种结构可用于**文本情感分析**, 对一些列的文本输入进行分类, 看是消极还是积极情感。
- **many to many** 结构, 给一些列输入得到一系列输出, 这种结构可用于**翻译或聊天对话**场景, 对输入的文本转换成另外一些列文本。
- **同步 many to many** 结构, 它是经典的RNN结构, 前一输入的状态会带到下一个状态中, 而且每个输入都会对应一个输出, 我们最熟悉的就是用于**字符预测**了, 同样也可以用于**视频分类**, 对视频的帧打标签。



## 2 Seq2Seq模型

Seq2Seq 是 2014 年 Google 提出的一个模型：[Sequence to Sequence Learning with Neural Networks](#)。论文中提出的 Seq2Seq 模型可简单理解为由三部分组成：

- Encoder
- Decoder
- 连接两者的 State Vector (中间状态向量) C

在上图 many to many 的两种模型中，可以看到第四和第五种是有差异的：

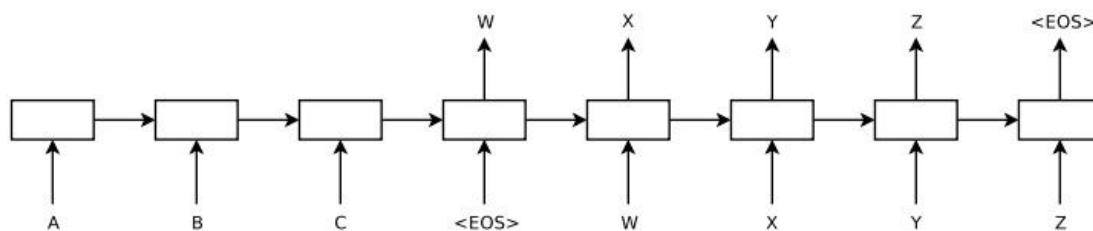
- 经典的RNN结构 (平：即第五种) 的输入和输出序列必须要是等长，它的应用场景也比较有限。
- 而第四种它可以是输入和输出序列不等长，这种模型便是Seq2Seq模型，即Sequence to Sequence。

Seq2Seq实现了从一个序列到另外一个序列的转换，比如google曾用Seq2Seq模型加attention模型来实现了翻译功能，类似的还可以实现聊天机器人对话模型。经典的RNN模型固定了输入序列和输出序列的大小，而Seq2Seq模型则突破了该限制。

这种结构最重要的地方在于输入序列和输出序列的长度是可变的。

Seq2Seq模型有两种常见结构。

- 1) 简单结构：该结构是最简单的结构，Decoder的第一个时刻只用到了Encoder最后输出的中间状态变量。



上图来自谷歌2014年的论文：[Sequence to Sequence Learning with Neural Networks](#)  
<https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>

- 2) 复杂结构

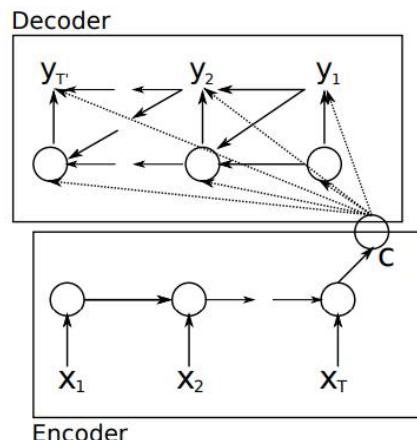


Figure 1: An illustration of the proposed RNN Encoder–Decoder.

上图来自蒙特利尔大学2014年的论文：[Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation](#)  
<https://www.aclweb.org/anthology/D14-1179.pdf>

Seq2Seq的应用有：

- 在英文翻译中，将英文输入到Encoder中，Decoder输出中文。蒙特利尔大学2014年的论文：[Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation](#)
- 在图像标注中，将图像特征输入到Encoder中，Decoder输出一段文字对图像的描述。Google2015年的论文：[Show and Tell: A Neural Image Caption Generator](#)
- 在 QA 系统中，将提出的问题输入Encoder中，Decoder输出对于问题的回答。

# 3 Encoder-Decoder 结构

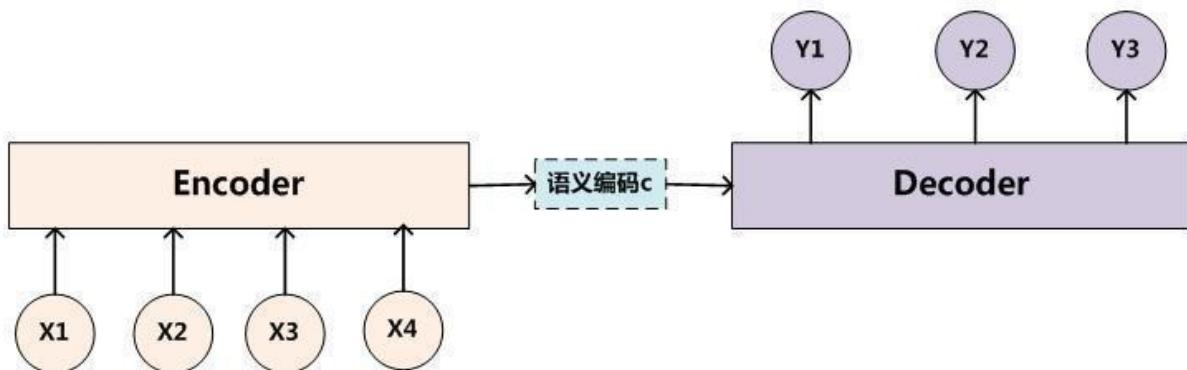
## 3.1 Encoder-Decoder 介绍

所谓的 Sequence2Sequence 任务主要是泛指一些 Sequence 到 Sequence 的映射问题，Sequence在这里可以理解为一个字符串序列，当我们在给定一个字符串序列后，希望得到与之对应的另一个字符串序列（如 翻译后的、如语义上对应的）时，这个任务就可以称为 Sequence2Sequence 了。

在现在的深度学习领域当中，通常的做法是将输入的源 Sequence 编码到一个中间的 context 当中，这个 context 是一个特定长度的编码（可以理解为一个向量），然后再通过这个 context 还原成一个输出的目标 Sequence。

- 如果用人的思维来看，就是我们先看到 源Sequence，将其读一遍，然后在我们大脑当中就记住了这个 源Sequence，并且存在大脑的某一个位置上，形成我们自己的记忆（对应Context），然后我们再经过思考，将这个大脑里的东西转变成输出，写下来。
- 那么我们大脑读入的过程叫做 Encoder，即将输入的东西变成我们自己的记忆，放在大脑当中，而这个记忆可以叫做 Context，然后我们再根据这个Context，转化成答案写下来，这个写的过程叫做Decoder。其实就是**编码-存储-解码**的过程。
- 而对应的，大脑怎么读入（Encoder怎么工作）有一个特定的方式，怎么记忆（Context）有一种特定的形式，怎么转变成答案（Decoder怎么工作）又有一种特定的工作方式。

现在我们大体了解了一个工作的流程 Encoder-Decoder 后，我们来介绍一个深度学习当中，最经典的 Encoder-Decoder 实现方式，即用RNN来实现。



在 RNN Encoder-Decoder 的工作当中，我们用一个 RNN 去模拟大脑的读入动作，用一个特定长度的特征向量去模拟记忆，再用另外一个 RNN 去模拟大脑思考得到答案的动作，将三者组织起来利用就成了一个可以实现 Seq2Seq 工作的“模拟大脑”了。

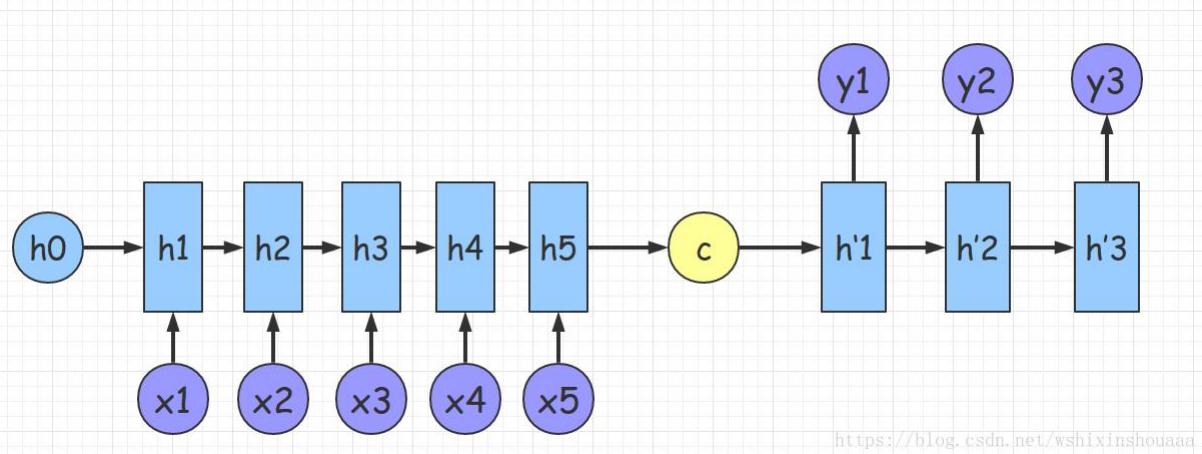
而我们剩下的工作也就是如何正确的利用RNN去实现，以及如何正确且合理的组织这三个部分了。

### 3.2 Encoder-Decoder 分析

- 1) **Encoder:** 给定句子对
- 2) **语义向量 C:**
  - 获取语义向量 C 最简单的方式就是直接将最后一个输入的隐状态作为语义向量 C。
  - 也可以对最后一个隐含状态做一个变换得到语义向量，
  - 还可以将输入序列的所有隐含状态做一个变换得到语义变量。
- 3) 得到中间语义向量  $C$  后，使用 Decoder 进行解码。Decoder 根据中间状态向量  $C$  和已经生成的历史信息  $y_1, y_2, \dots, y_{i-1}$  去生成  $t$  时刻的单词  $y_i$ ：

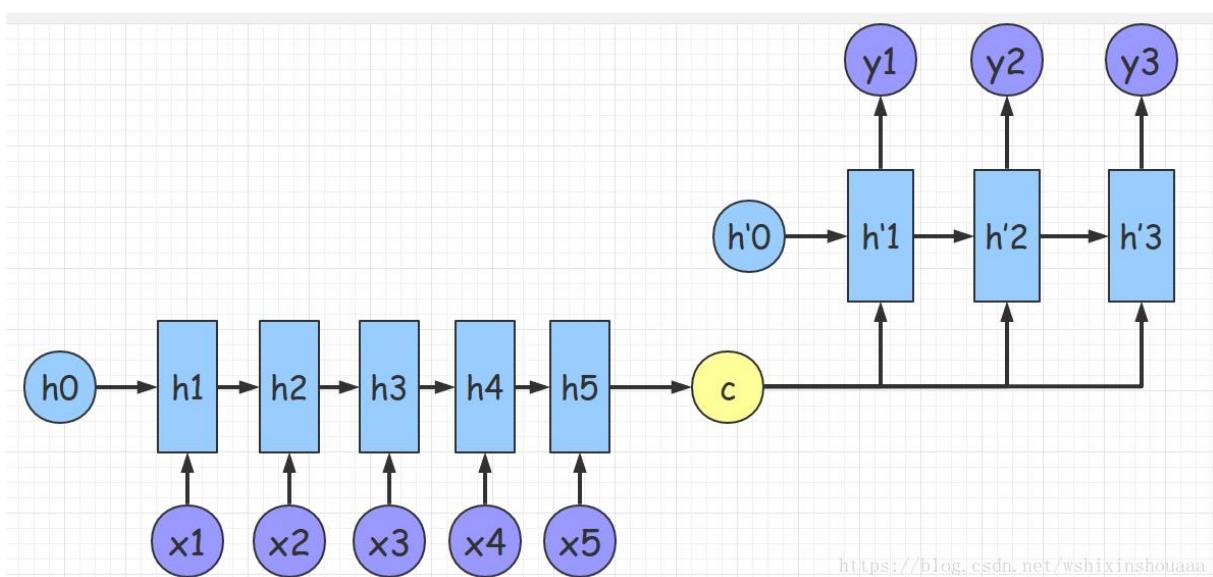
$$y_i = g(C, y_1, y_2, \dots, y_{i-1})$$

如果直接将  $C$  输入到 Decoder 中，则是 Seq2Seq 模型的第一种模型：



<https://blog.csdn.net/wshixinshouaaa>

如果将  $C$  当作 Decoder 的每一时刻输入，则是 Seq2Seq 模型的第二种模型：



<https://blog.csdn.net/wshixinshouaaa>

## 4 Attention 机制

### 4.1 Encoder-Decoder 结构的局限性

#### 1) Encoder 和 Decoder 的唯一联系只有语义编码 C

即将整个输入序列的信息编码成一个固定大小的状态向量再解码，相当于将信息“有损压缩”。很明显这样做有两个缺点：

- 中间语义向量无法完全表达整个输入序列的信息。
- 随着输入信息长度的增加，由于向量长度固定，先前编码好的信息会被后来的信息覆盖，丢失很多信息。

#### 2) 不同位置的单词的贡献都是一样的

Decoder过程，其输出的产生如下：

$$\begin{aligned}y_1 &= g(C, h'_0) \\y_2 &= g(C, y_1) \\y_3 &= g(C, y_1, y_2)\end{aligned}$$

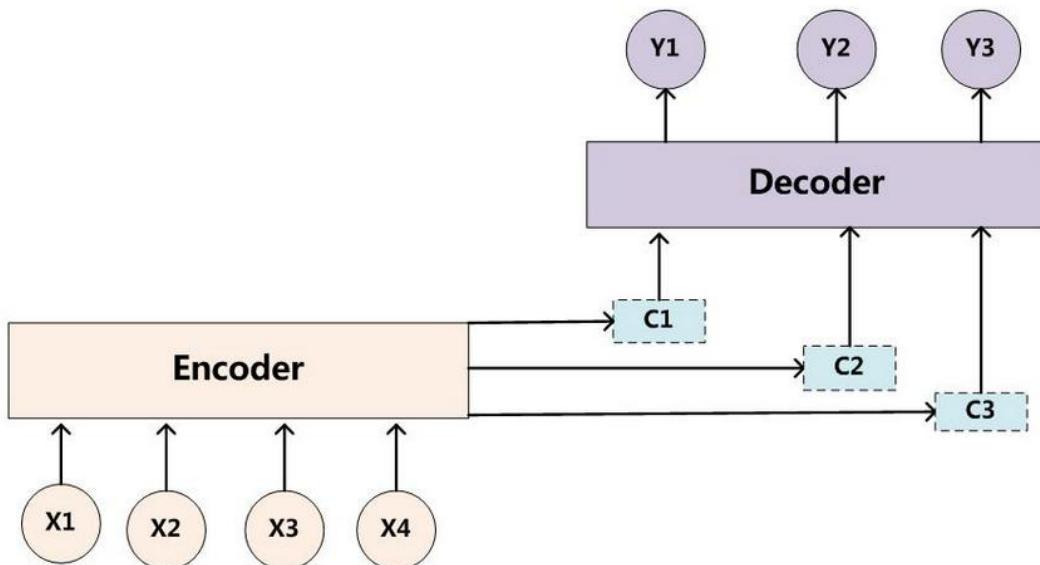
明显可以发现在生成 $y_1$ 、 $y_2$ 、 $y_3$ 时，语义编码 $C$ 对它们所产生的贡献都是一样的。例如翻译：Cat chase mouse，Encoder-Decoder模型逐字生成：“猫”、“捉”、“老鼠”。在翻译mouse单词时，每一个英语单词对“老鼠”的贡献都是相同的。如果引入了Attention模型，那么mouse对于它的影响应该是最大的。

### 4.2 Attention 机制原理

为了解决上面两个问题，于是引入了Attention模型。

Attention模型的特点是Decoder不再将整个输入序列编码为固定长度的中间语义向量 $C$ ，而是根据当前生成的新单词计算新的 $C_i$ ，使得每个时刻输入不同的 $C_i$ ，这样就解决了单词信息丢失的问题。

引入了Attention的Encoder-Decoder模型如下图：



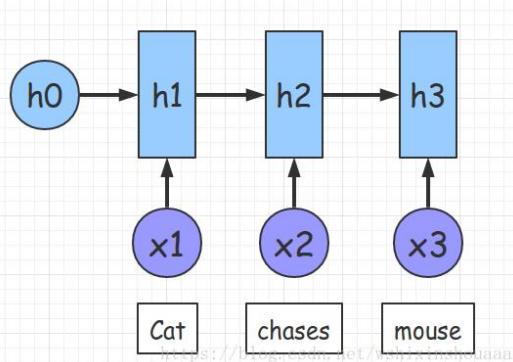
对于刚才提到的那个“猫捉老鼠”的翻译过程变成了如下：

$$y_1 = g(C_1, h'_0)$$

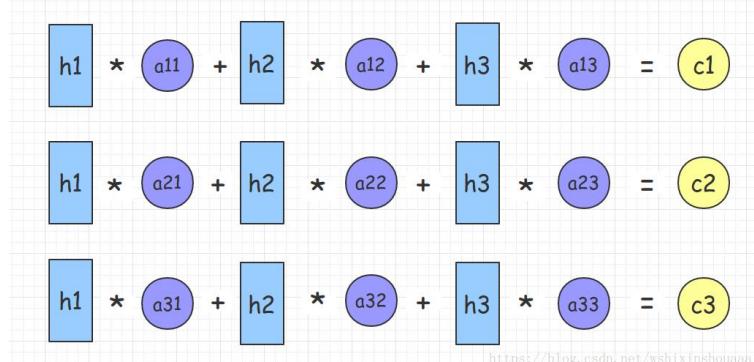
$$y_2 = g(C_2, y_1)$$

$$y_3 = g(C_3, y_1, y_2)$$

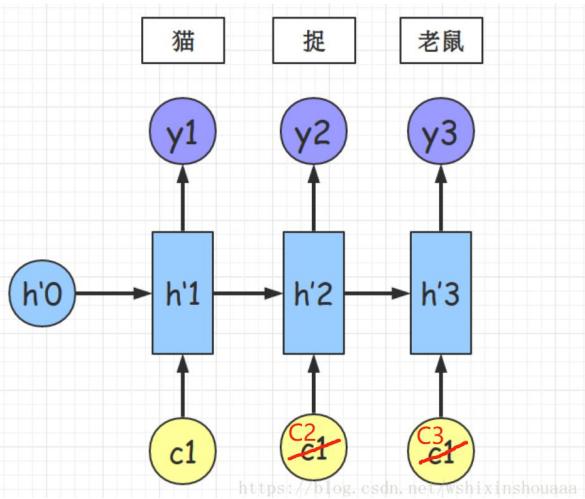
整个翻译流程如下：



图中输入是 Cat chase mouse , Encoder中隐层  $h_1$ 、 $h_2$ 、 $h_3$  可看作经过计算 Cat、chase、mouse 这些词的信息。



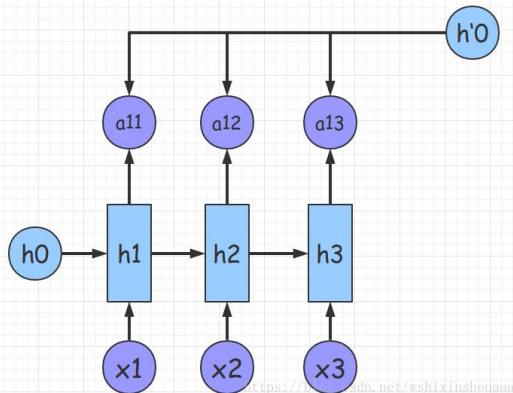
使用  $a_{ij}$  表示 Encoder 中第  $j$  阶段的  $h_j$  和解码时第  $i$  阶段的相关性，计算出解码需要的中间语义向量  $C_i$ 。  
 $C_1$  和“猫”关系最近，相对应的  $a_{11}$  要比  $a_{12}$ 、 $a_{13}$  大；而  $C_2$  和“捉”关系最近，相对应的  $a_{22}$  要比  $a_{21}$ 、 $a_{23}$  大；同理  $C_3$  和“老鼠”关系最近，相对应的  $a_{33}$  要比  $a_{31}$ 、 $a_{32}$  大。



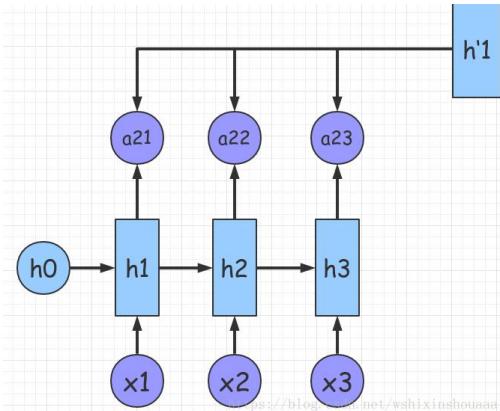
那么参数  $a_{ij}$  是如何得到呢？

Encoder 中第  $j$  个隐层单元  $h_j$  和 Decoder 第  $i - 1$  个隐层单元  $h'_{i-1}$  经过运算得到  $a_{ij}$ 。

例如  $a_{1j}$  的计算过程：



例如  $a_{2j}$  的计算过程：



通过训练，注意力机制可以对关系较大的输入输出的赋以较大权重（两者在转换中对齐的概率更大），对位置信息进行了建模，而因此减少了信息损失，能专注于更重要的信息进行序列预测。