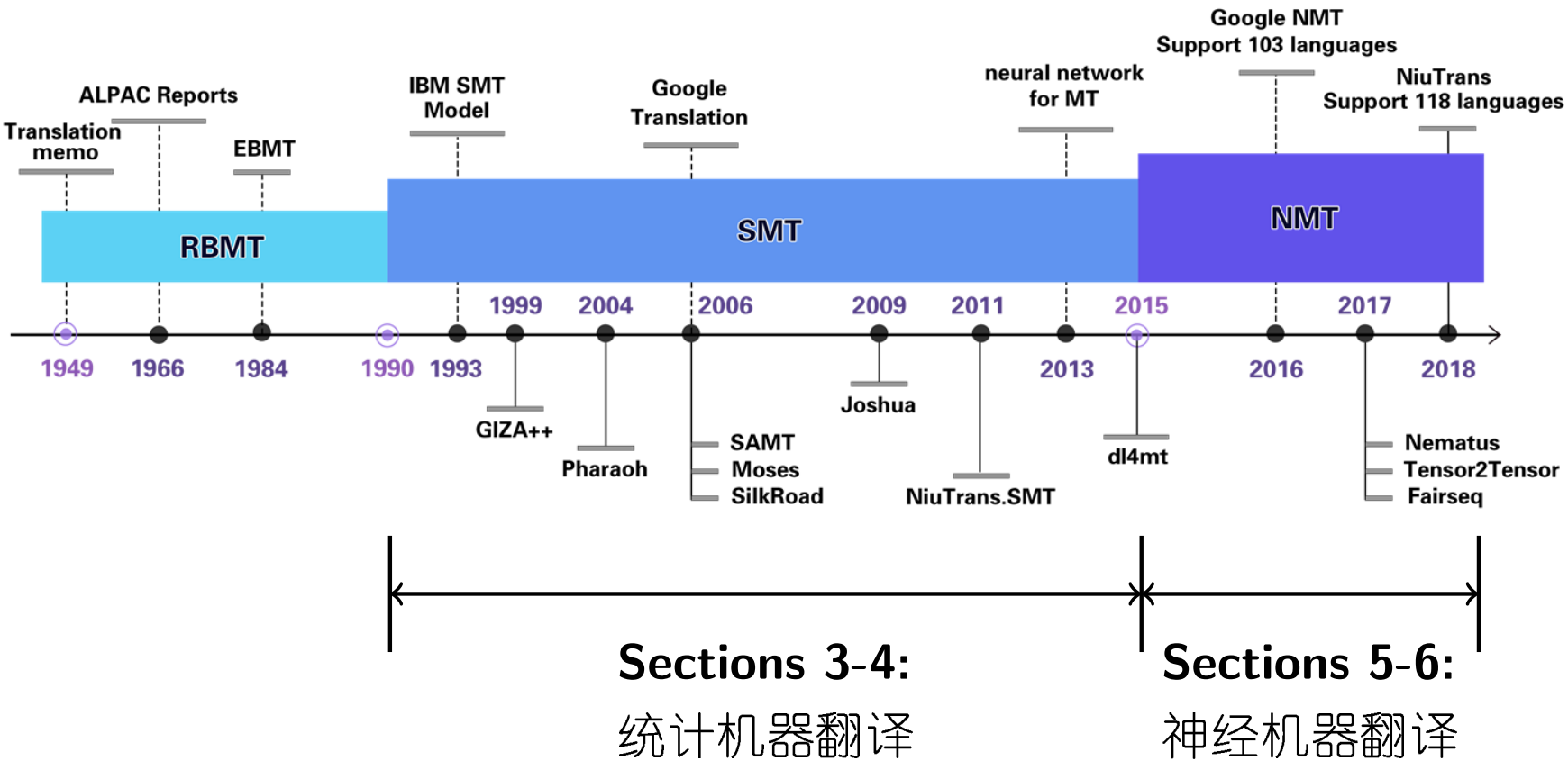
神经机器翻译

肖桐　朱靖波

xiaotong@mail.neu.edu.cn
zhujingbo@mail.neu.edu.cn

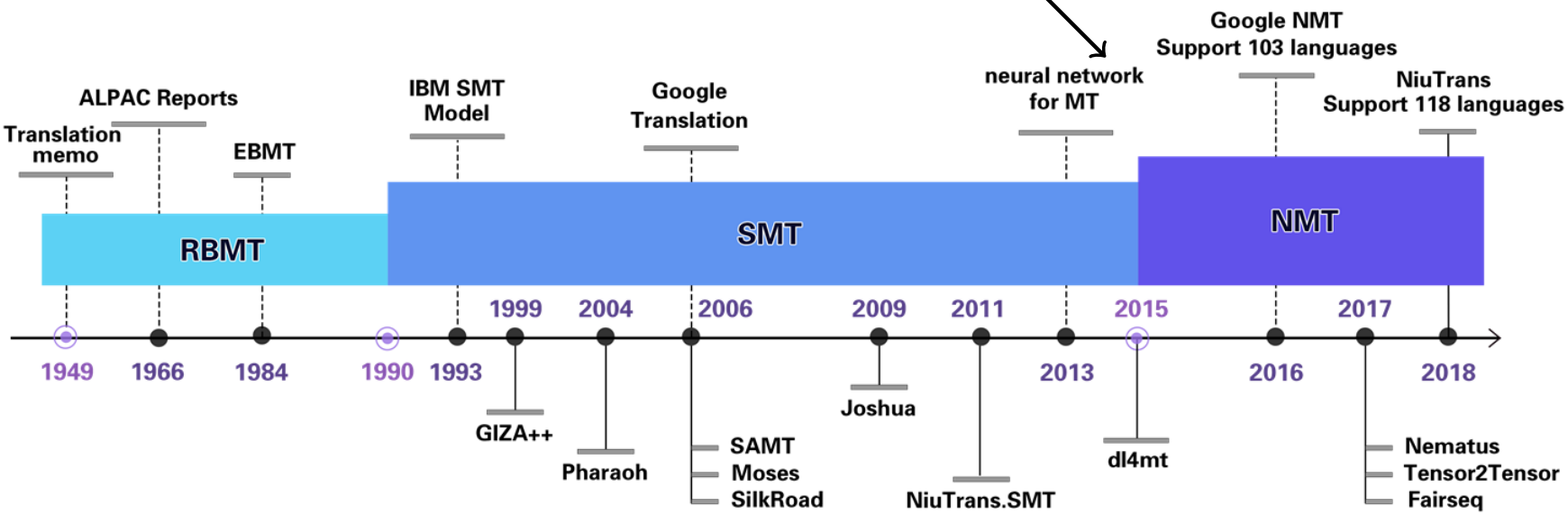
东北大学 自然语言处理实验室
http://www.nlplab.com

Landscape


Google NMT
Support 103 languages
NiuTrans
Support 118 languages
neural network
for MT
IBM SMT
Model
Google
Translation
ALPAC Reports
Translation
memo
EBMT
RBMT
SMT
NMT
1999
2004
2006
2009
2011
2015
2017
1949
1966
1984
1990
1993
2013
2016
2018
GIZA++
Pharaoh
SAMT
Moses
SilkRoad
Joshua
NiuTrans.SMT
dl4mt
Nematus
Tensor2Tensor
Fairseq
Sections 3-4:
统计机器翻译
Sections 5-6:
神经机器翻译

Landscape

本章内容（Section 6）：
神经机器翻译建模及实现



Sections 3-4:
统计机器翻译

Sections 5-6:
神经机器翻译

机器翻译今天的水平

原文(英语): During Soviet times, if a city's population topped one million, it would become eligible for its own metro. Planners wanted to brighten the lives of everyday Soviet citizens, and saw the metros, with their tens of thousands of daily passengers, as a singular opportunity to do so. In 1977, Tashkent, the capital of Uzbekistan, became the seventh Soviet city to have a metro built. Grand themes celebrating the history of Uzbekistan and the Soviet Union were brought to life, as art was commissioned and designers set to work. The stations reflected different themes, some with domed ceilings and painted tiles reminiscent of Uzbekistan's Silk Road mosques, while others ...

译文

在苏联时代，如果一个城市的人口突破一百万，这将成为合资格为自己的地铁。规划者想去照亮每天的苏联公民的生命，看到地铁，与他们的数十每天数千乘客，作为一个独特的机会来这样做。1977年，塔什干，乌兹别克斯坦的首都，成了苏联第七城市建有地铁。宏大主题，庆祝乌兹别克斯坦和苏联的历史被带到生活，因为艺术是委托和设计师开始工作。车站反映了不同的主题，有的圆顶天花板和绘瓷砖让人想起乌兹别克斯坦是丝绸之路的清真寺，而另一些则装饰着...

译文

在苏联时期，如果一个城市的人口超过一百万，它就有资格拥有自己的地铁。 规划者想要照亮日常苏联公民的生活，并把拥有数万名每日乘客的地铁看作是这样做的一个绝佳机会。 1977年，乌兹别克斯坦首都塔什干成为苏联第七个修建地铁的城市。 随着艺术的委托和设计师们的工作，乌兹别克斯坦和苏联历史的宏伟主题被赋予了生命力。 这些电台反映了不同的主题，有的有穹顶和彩砖，让人想起乌兹别克斯坦的丝绸之路清真寺，有的则用...

机器翻译今天的水平

原文(英语): During Soviet times, if a city's population topped one million, it would become eligible for its own metro. Planners wanted to brighten the lives of everyday Soviet citizens, and saw the metros, with their tens of thousands of daily passengers, as a singular opportunity to do so. In 1977, Tashkent, the capital of Uzbekistan, became the seventh Soviet city to have a metro built. Grand themes celebrating the history of Uzbekistan and the Soviet Union were brought to life, as art was commissioned and designers set to work. The stations reflected different themes, some with domed ceilings and painted tiles reminiscent of Uzbekistan's Silk Road mosques, while others ...

译文(统计机器翻译)

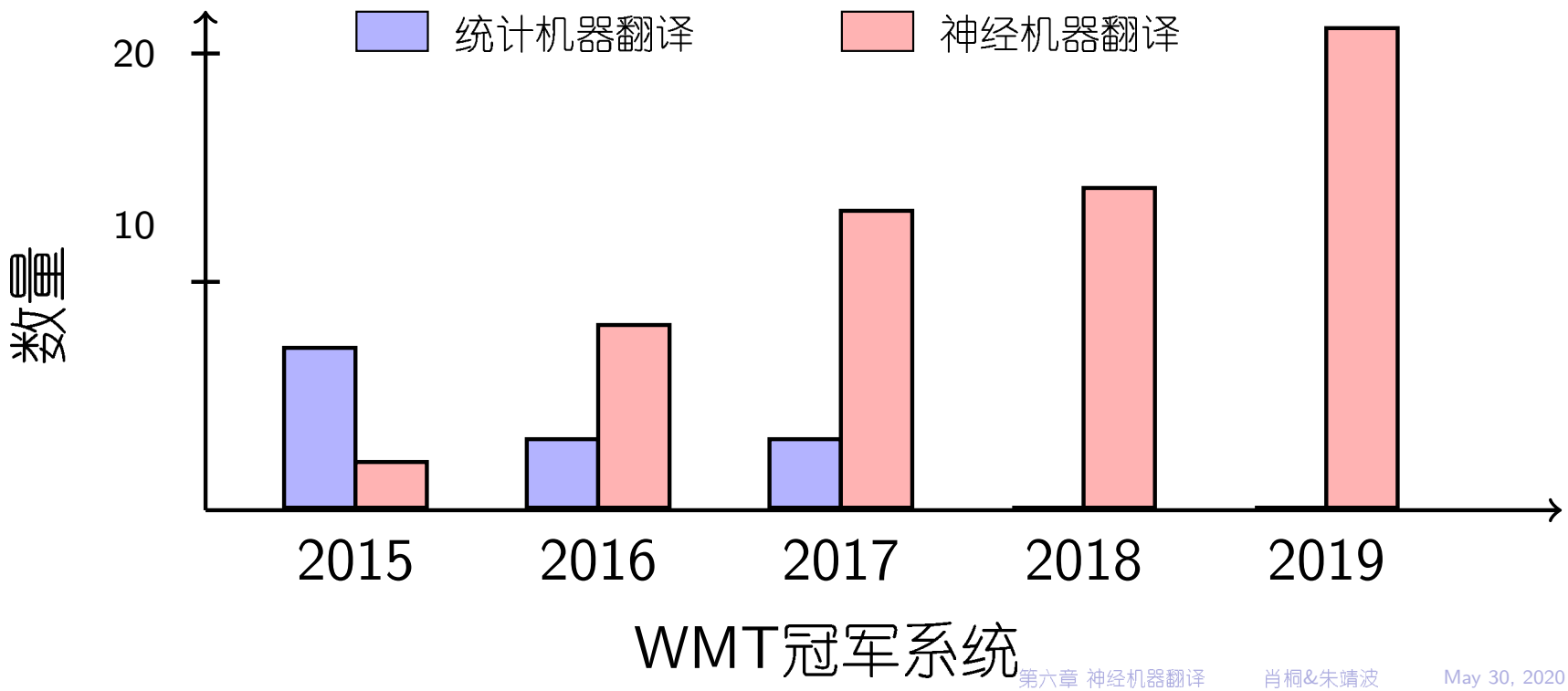
在苏联时代，如果一个城市的人口突破一百万，这将成为合资格为自己的地铁。规划者想去照亮每天的苏联公民的生命，看到地铁，与他们的数十每天数千乘客，作为一个独特的机会来这样做。1977年，塔什干，乌兹别克斯坦的首都，成了苏联第七城市建有地铁。宏大主题，庆祝乌兹别克斯坦和苏联的历史被带到生活，因为艺术是委托和设计师开始工作。车站反映了不同的主题，有的圆顶天花板和绘瓷砖让人想起乌兹别克斯坦是丝绸之路的清真寺，而另一些则装饰着...

译文(神经机器翻译 - 很流畅！)

在苏联时期，如果一个城市的人口超过一百万，它就有资格拥有自己的地铁。 规划者想要照亮日常苏联公民的生活，并把拥有数万名每日乘客的地铁看作是这样做的一个绝佳机会。 1977年，乌兹别克斯坦首都塔什干成为苏联第七个修建地铁的城市。 随着艺术的委托和设计师们的工作，乌兹别克斯坦和苏联历史的宏伟主题被赋予了生命力。 这些电台反映了不同的主题，有的有穹顶和彩砖，让人想起乌兹别克斯坦的丝绸之路清真寺，有的则用...

神经机器翻译的进展

- 想当年，08年NIST举办的汉英机器翻译评测，BLEU能突破30已经是巨猛无比的结果，而现在的神经机器翻译轻松突破45！
- 再比如，机器翻译的旗舰评测比赛WMT(Workshop of Machine Translation)已经被神经机器翻译刷榜
  - 现在冠军系统几乎没有纯统计机器翻译系统



神经机器翻译的进展(续)

- 神经机器翻译在很多场景下已经超越统计机器翻译

#	自动评价			系统
	BLEU	HTER	mTER	
统计机器翻译	25.3	28.0	21.8	PBSY
	24.6	29.9	23.4	HPB
	25.8	29.0	22.7	SPB
神经机器翻译	**31.1**	**21.1**	**16.2**	NMT

*Neural versus Phrase-Based Machine Translation Quality: a Case Study

- 微软的报道：在部分场景下机器翻译质量已经接近甚至超过人工翻译

#	人工评价	系统
机器翻译	**69.9**	COMBO-6
	69.8	COMBO-4
	69.9	COMBO-5
人工翻译	68.6	REFERENCE-HT
	67.6	REFERENCE-PE

*Achieving Human Parity on Automatic Chinese to English News Translation

神经机器翻译的优势

- 神经网络方法带来了新的思路，还是同样的表再看一遍

传统基于统计的方法 (统计机器翻译)	深度学习方法 (神经机器翻译)
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

神经机器翻译的优势

- 神经网络方法带来了新的思路，还是同样的表再看一遍

传统基于统计的方法 (统计机器翻译)	深度学习方法 (神经机器翻译)
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

- 在统计机器翻译时代，系统依赖很多模块，比如
  - **词对齐**：双语句子之间词和词的对应关系
  - **短语(规则)表**：原文和译文互译的片段
  - **特征**：人工设计的用于建模翻译的各种各样的子模型
  - **目标语语言模型**：大量单语数据训练的目标语建模模块
  - ...

神经机器翻译的优势

- 神经网络方法带来了新的思路，还是同样的表再看一遍

传统基于统计的方法 (统计机器翻译)	深度学习方法 (神经机器翻译)
基于离散空间的表示模型	基于连续空间的表示模型
NLP问题的隐含结构假设	无隐含结构假设，端到端学习
特征工程为主	无显性特征，但需要设计网络
特征、规则的存储耗资源	模型存储相对小，但计算慢

- ~~在统计机器翻译时代，系统依赖很多模块，比如~~
  - ~~词对齐：双语句子之间词和词的对应关系~~
  - ~~短语(规则)表：原文和译文互译的片段~~
  - ~~特征：人工设计的用于建模翻译的各种各样的子模型~~
  - ~~目标语语言模型：大量单语数据训练的目标语建模模块~~
  - ~~...~~
- 在神经机器翻译时代，以上均不是必要的，而仅仅需要一个神经网络进行端到端建模

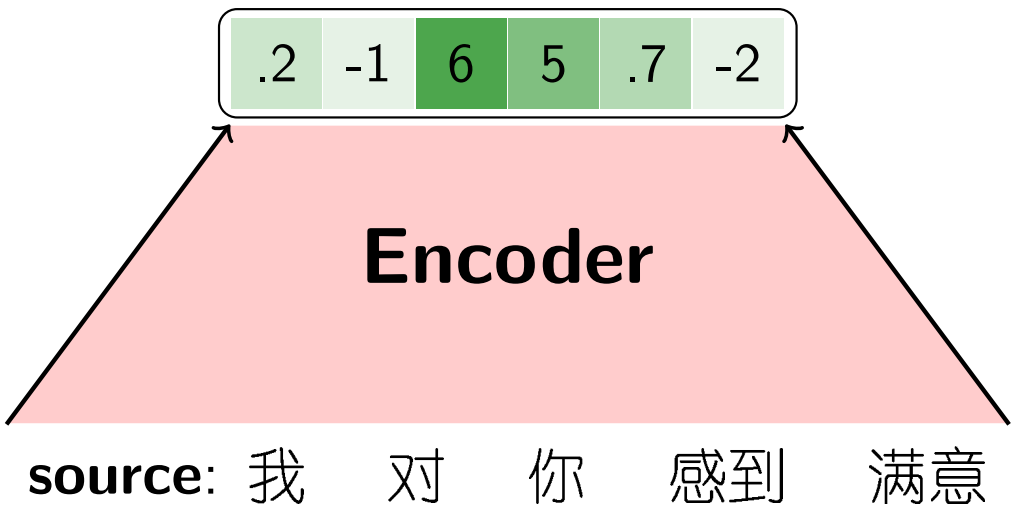
编码器-解码器框架

- 神经机器翻译遵循一种叫“编码器-解码器”的结构

source: 我　对　你　感到　满意

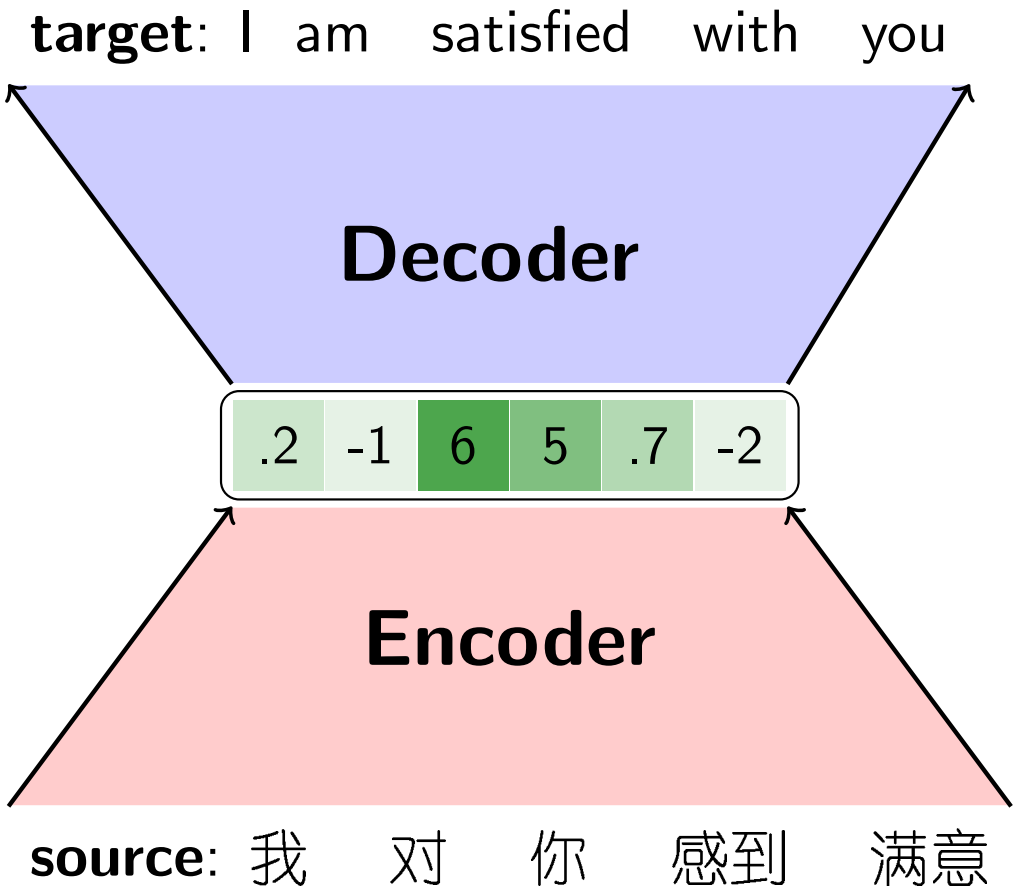
编码器-解码器框架

- 神经机器翻译遵循一种叫“编码器-解码器”的结构
  - **编码器**负责把源语言编码成一种句子表示形式(如：向量)



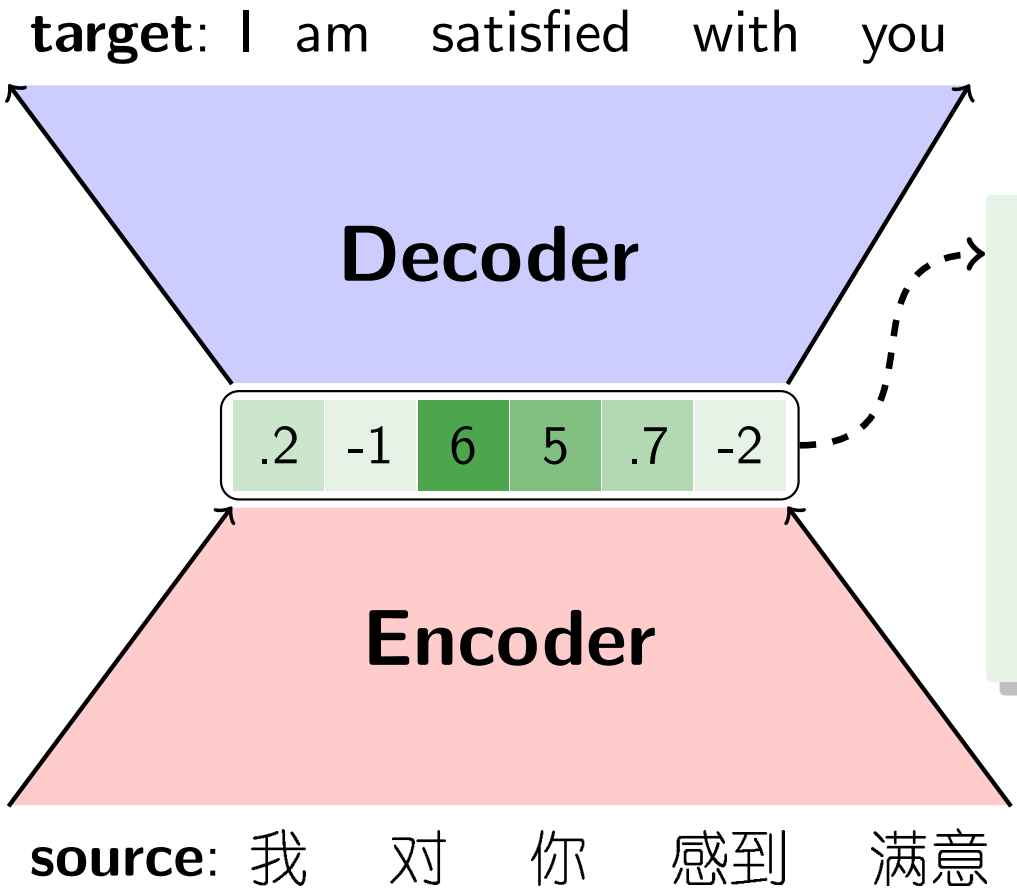
编码器-解码器框架

- 神经机器翻译遵循一种叫“编码器-解码器”的结构
  - 编码器负责把源语言编码成一种句子表示形式(如：向量)
  - 解码器负责利用这种表示逐词把目标语句子生成处理



编码器-解码器框架

- 神经机器翻译遵循一种叫“编码器-解码器”的结构
  - **编码器**负责把源语言编码成一种句子表示形式(如：向量)
  - **解码器**负责利用这种表示逐词把目标语句子生成处理
  - 编码器和解码器之间的向量就是连接这两部分的句子表示



句子的表示:

源语言句子被表示成一个实数向量$(0.2, -1, 6, 5, 0.7, -2)$

不要问0.2是什么意思，因为只有系统自己才知道 :)

基于连续空间表示模型的方法

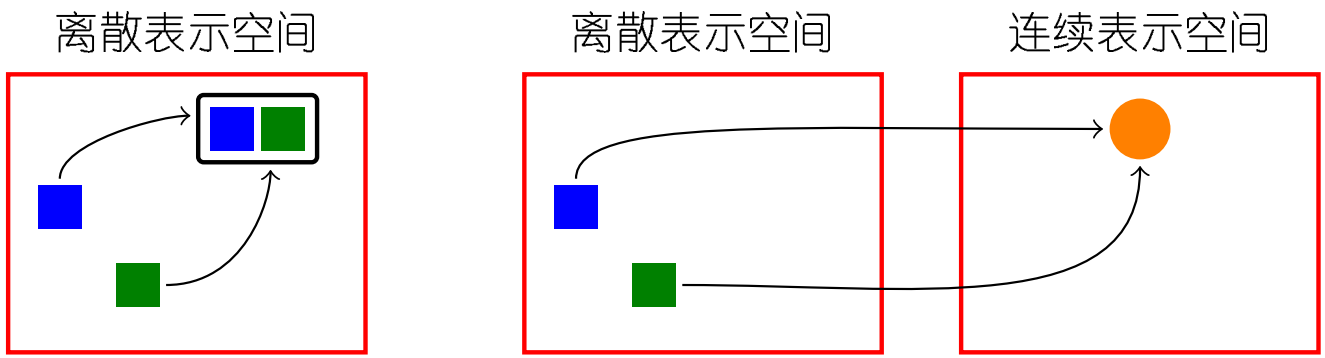
- 编码器-解码器框架的革命在于：它把传统基于符号的离散型知识转化为基于表示的连续型知识
  - 比如，对于一个短语，它可以对应一个文法规则的使用过程，而规则都是由离散的符号表示
  - 它也可以被表示为一种更加抽象的形式，通过向量记录短语的各个“属性”

基于连续空间表示模型的方法

- 编码器-解码器框架的革命在于：它把传统基于符号的离散型知识转化为基于表示的连续型知识
  - 比如，对于一个短语，它可以对应一个文法规则的使用过程，而规则都是由离散的符号表示
  - 它也可以被表示为一种更加抽象的形式，通过向量记录短语的各个“属性”
- 这种表示形式上的创新，让我们不再依赖离散符号系统的各种分解、组合，而直接在连续空间下学习这种表示
  - 所谓“端到端”只是这种表示模型下的一种自然的学习范式

基于连续空间表示模型的方法

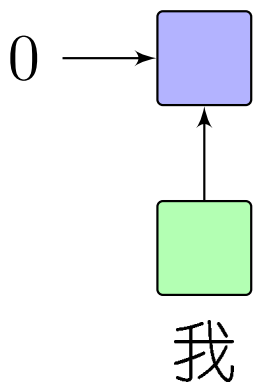
- 编码器-解码器框架的革命在于：它把传统基于符号的离散型知识转化为基于表示的连续型知识
  - 比如，对于一个短语，它可以对应一个文法规则的使用过程，而规则都是由离散的符号表示
  - 它也可以被表示为一种更加抽象的形式，通过向量记录短语的各个“属性”
- 这种表示形式上的创新，让我们不再依赖离散符号系统的各种分解、组合，而直接在连续空间下学习这种表示
  - 所谓“端到端”只是这种表示模型下的一种自然的学习范式



(a) 统计机器翻译　　(b) 神经机器翻译

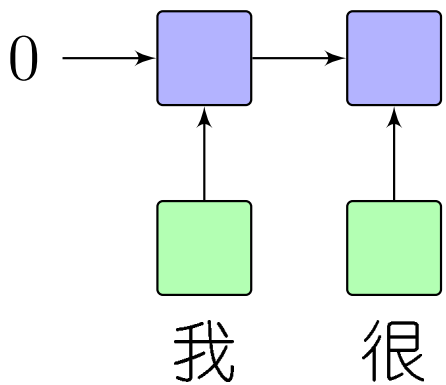
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词



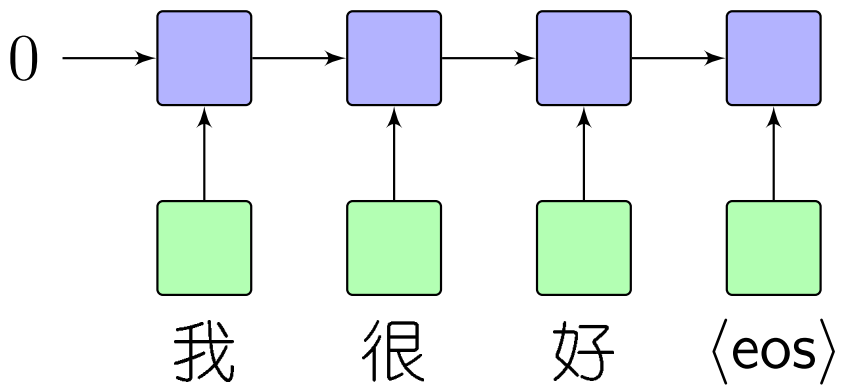
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词



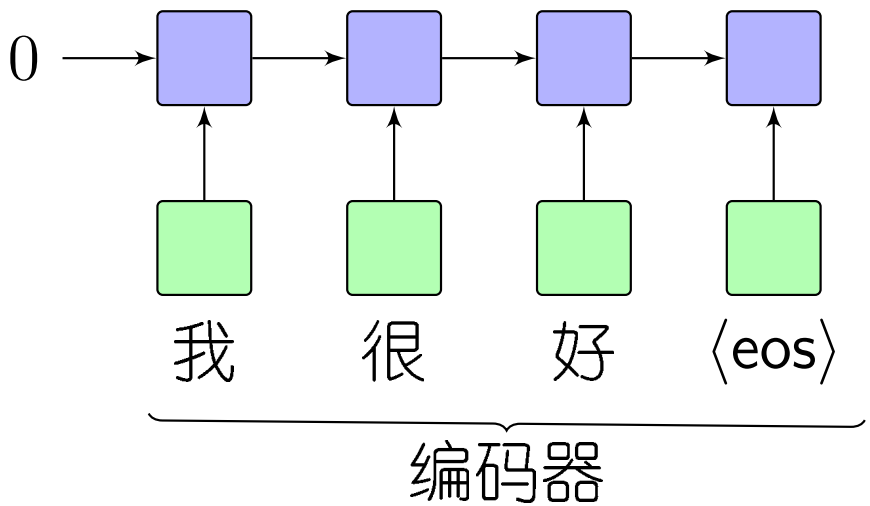
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词



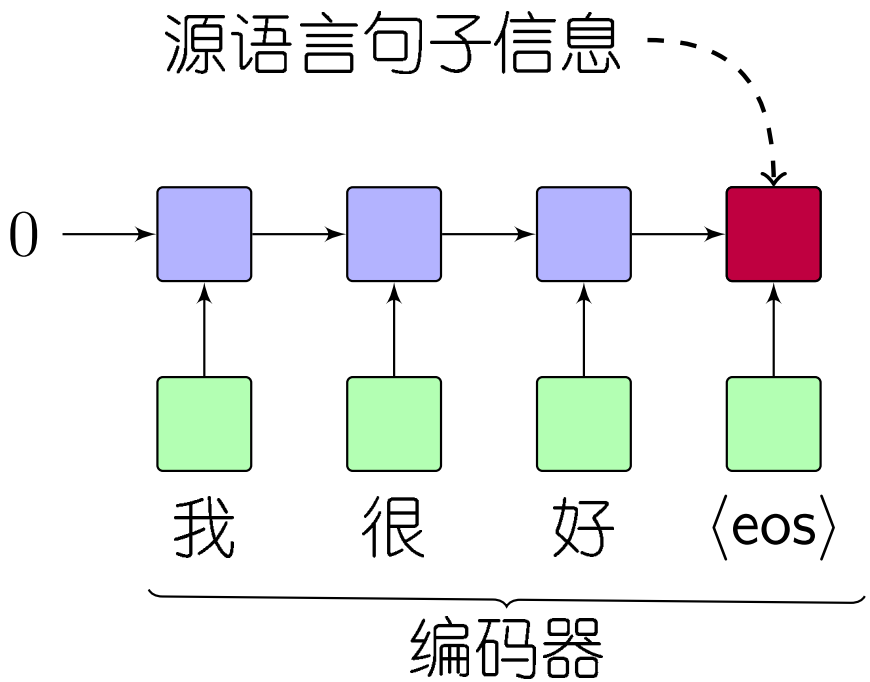
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词



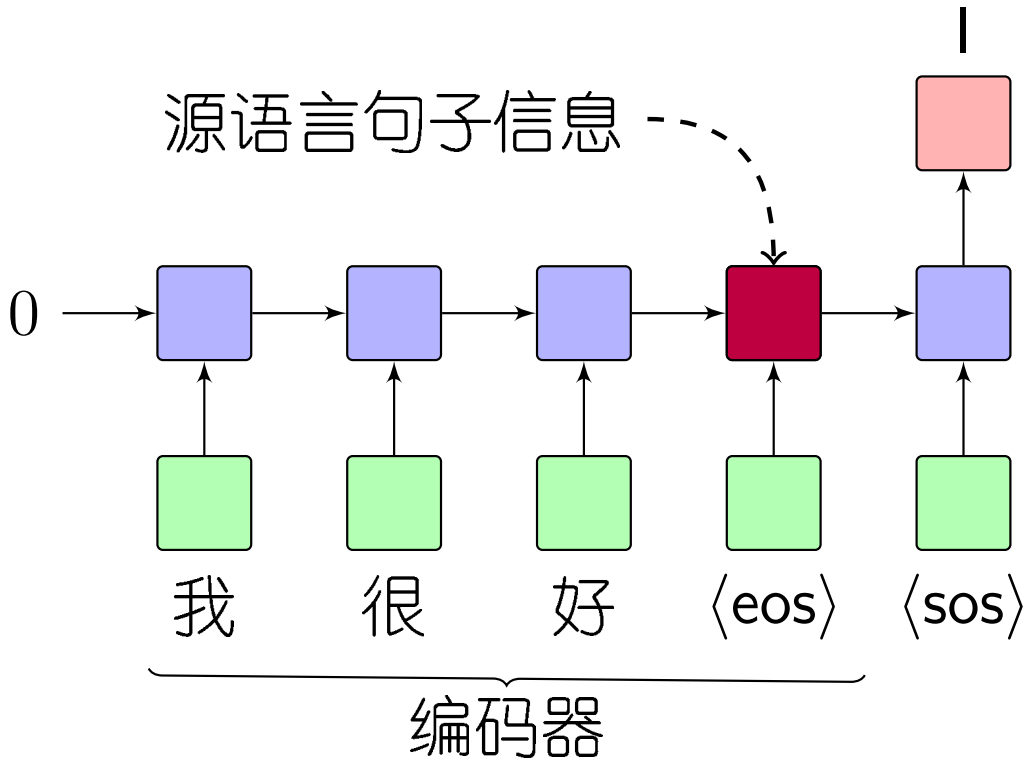
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词
  - 源语言句子信息被表示在最后一个循环单元的输出中



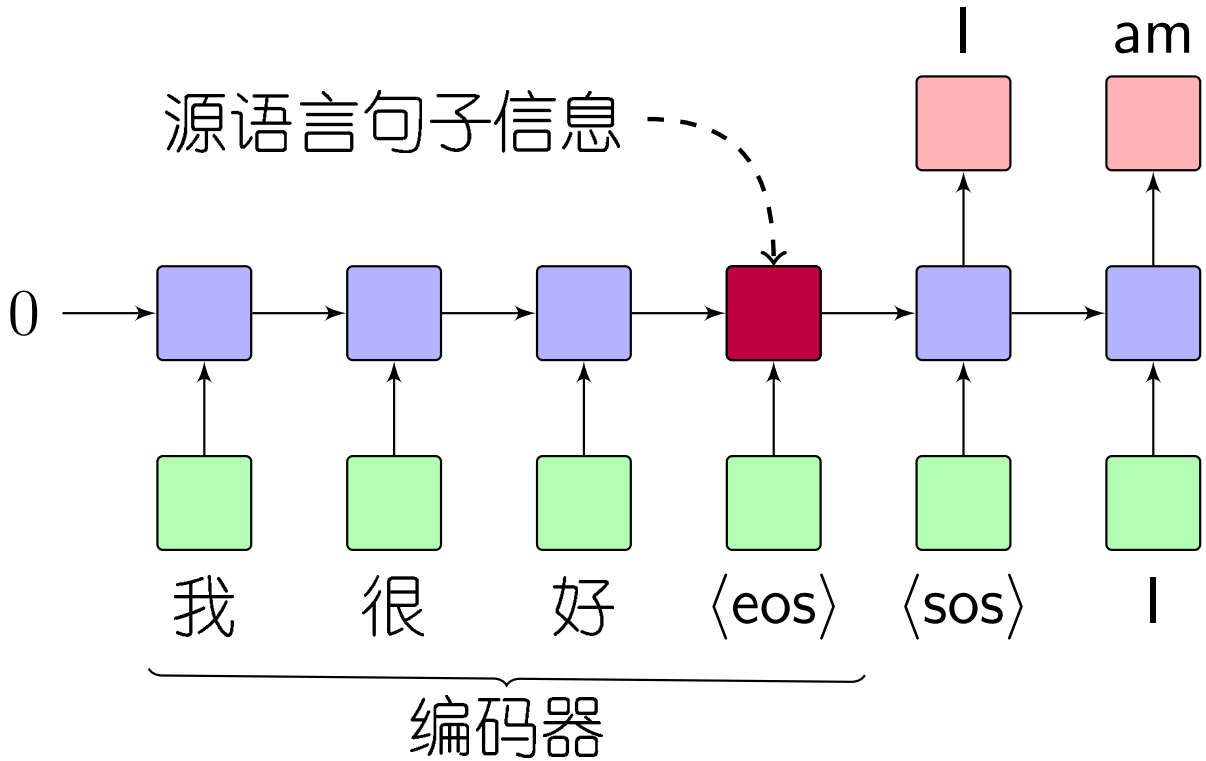
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词
  - 源语言句子信息被表示在最后一个循环单元的输出中
  - **解码器**利用源语言句子信息逐词生成目标语译文



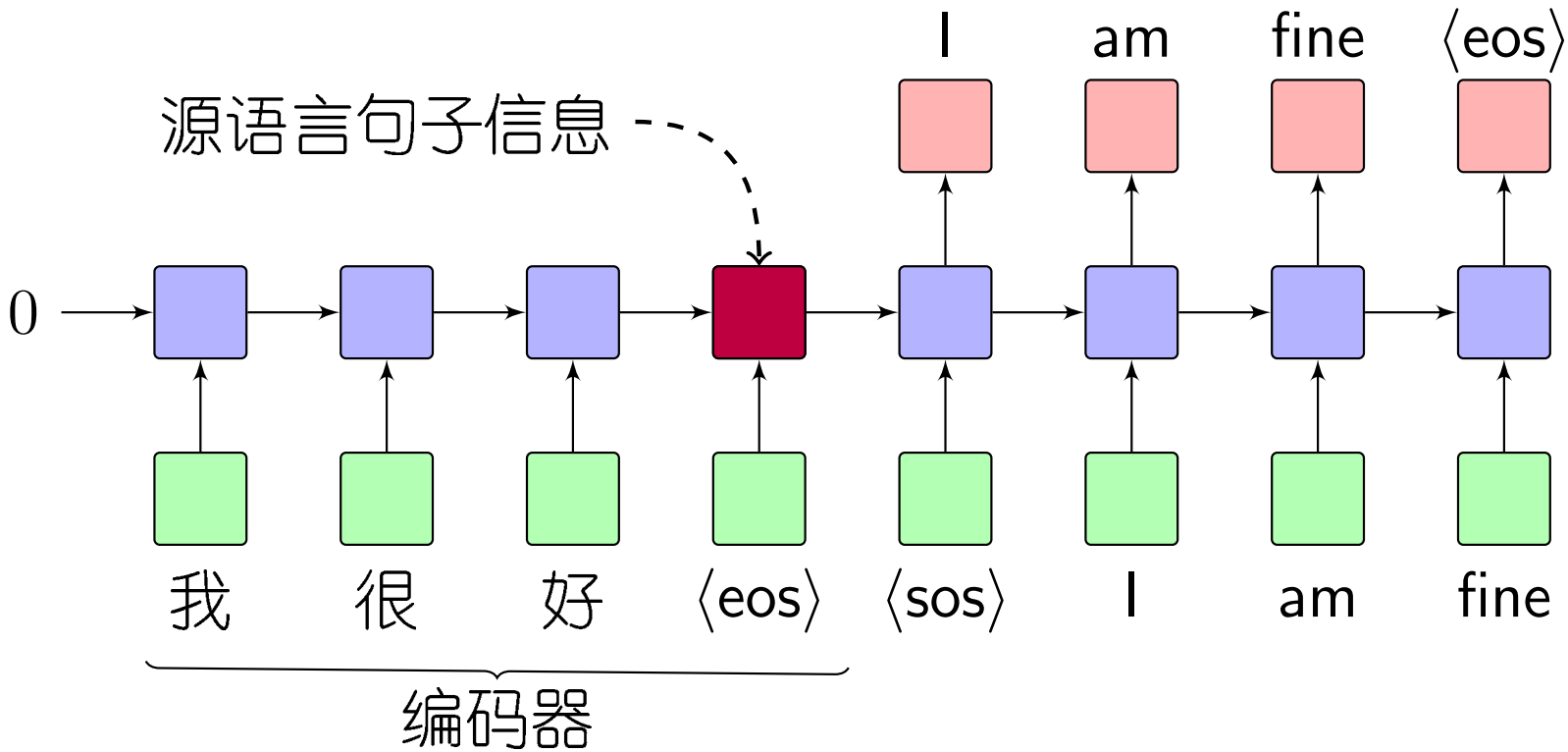
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词
  - 源语言句子信息被表示在最后一个循环单元的输出中
  - **解码器**利用源语言句子信息逐词生成目标语译文



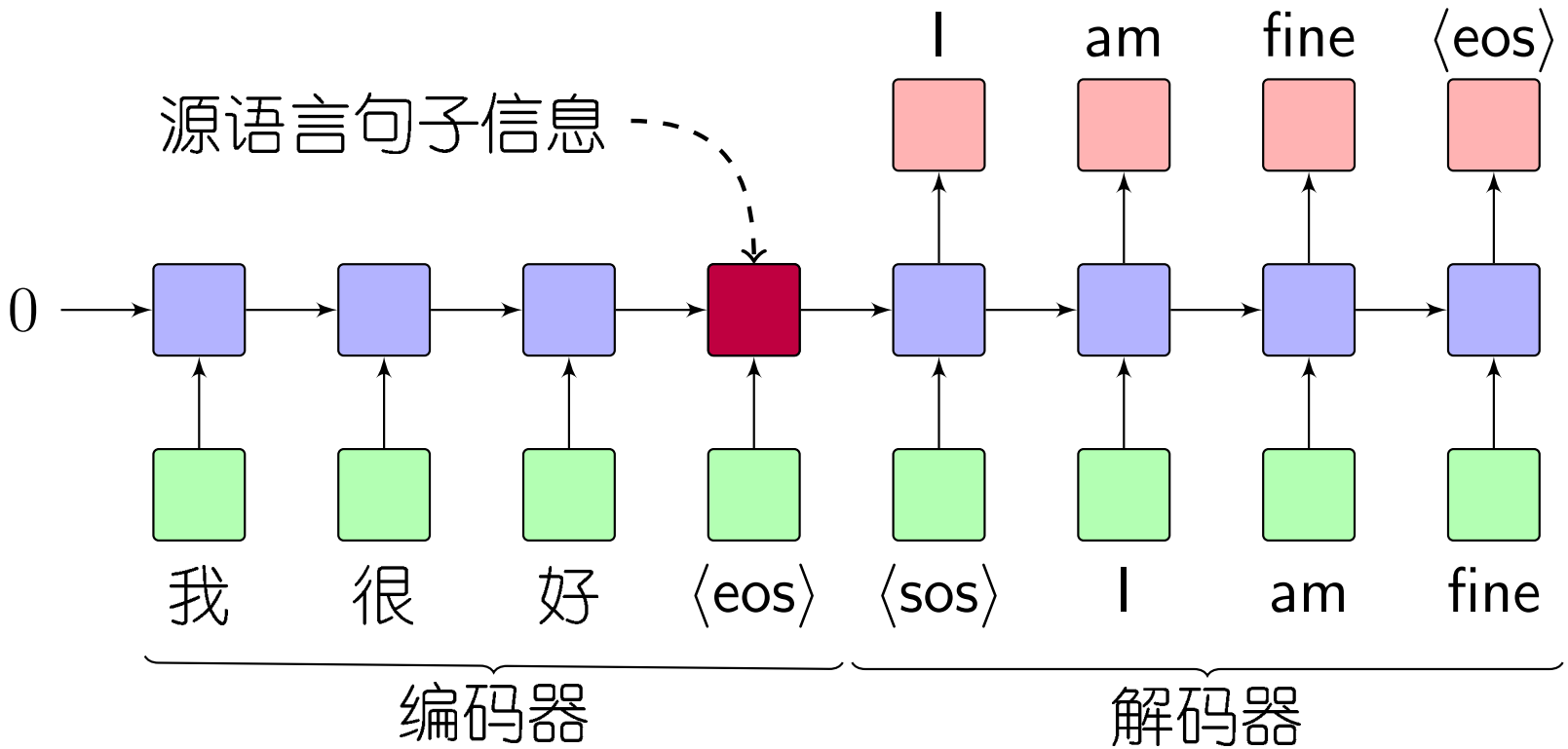
简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词
  - 源语言句子信息被表示在最后一个循环单元的输出中
  - **解码器**利用源语言句子信息逐词生成目标语译文



简单的运行实例

- 一个简单的例子：基于循环神经网络的翻译过程
  - **编码器**顺序处理源语言单词
  - 源语言句子信息被表示在最后一个循环单元的输出中
  - **解码器**利用源语言句子信息逐词生成目标语译文



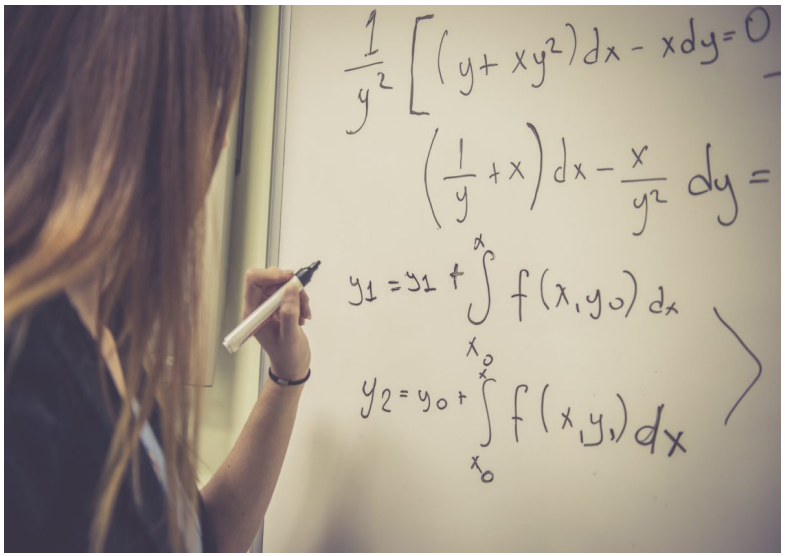
神经机器翻译会“魔法”？

统计机器翻译仍然依赖人工定义特征和翻译单元

- 相对符合人类的理解
- 具有一定可解释性

神经机器翻译把所有工作都交给神经网络

- 不需要先验假设
- 语言的“表示”给机器看的



- 神经机器翻译在使用魔法？
  - No! No! No! 变化的只有人类使用知识的形式

神经机器翻译所带来的范式更迭

- 不同机器翻译时代，人类知识的使用方式

机器翻译方法	人类参与方式
基于规则的方法	设计翻译规则
传统统计方法	设计翻译特征
神经网络方法	设计网络架构

神经机器翻译所带来的范式更迭

- 不同机器翻译时代，人类知识的使用方式

机器翻译方法	人类参与方式
基于规则的方法	设计翻译规则
传统统计方法	设计翻译特征
神经网络方法	设计网络架构

人类在机器翻译中将扮演什么角色？

神经机器翻译所带来的范式更迭

- 不同机器翻译时代，人类知识的使用方式

机器翻译方法	人类参与方式
基于规则的方法	设计翻译规则
传统统计方法	设计翻译特征
神经网络方法	设计网络架构

人类在机器翻译中将扮演什么角色？

- 机器翻译会逐渐替代人？
- 还需要人的语言学知识吗？
- 就连神经网络也可以通过结构搜索自动学习，人类是不是就什么也不用干了？

神经机器翻译所带来的范式更迭

- 不同机器翻译时代，人类知识的使用方式

机器翻译方法	人类参与方式
基于规则的方法	设计翻译规则
传统统计方法	设计翻译特征
神经网络方法	设计网络架构

人类在机器翻译中将扮演什么角色？

- 机器翻译会逐渐替代人？- 不会，爹是不能被儿子替代的
- 还需要人的语言学知识吗？- 会，但是需要新的思路
- 就连神经网络也可以通过结构搜索自动学习，人类是不是就什么也不用干了？- 结构搜索又是谁设计的，人至少需要敲下键盘吧，哈哈哈哈

Outline

入门：循环网络翻译模型及注意力机制

1. 起源
2. 模型结构
3. 注意力机制
4. 训练和推断

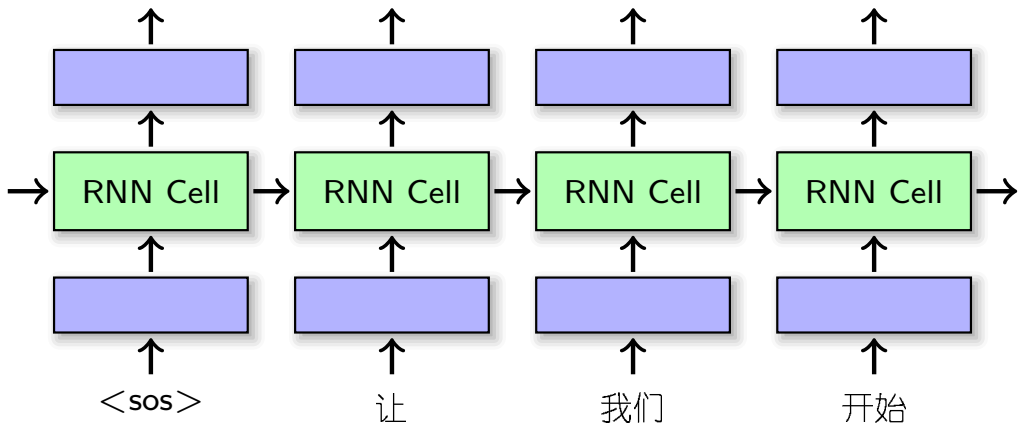
热门：**Transformer**

1. 自注意力模型
2. 多头注意力和层正则化
3. 更深、更宽的模型

其它：一些有趣的神经机器翻译应用

首先

循环网络翻译模型及注意力机制



最初的神经机器翻译

- 神经网络的在机器翻译中并不新鲜，在很多模块中早有实现，比如，翻译候选打分、语言模型等
  - 但是，整个框架仍然是统计机器翻译

最初的神经机器翻译

- 神经网络的在机器翻译中并不新鲜，在很多模块中早有实现，比如，翻译候选打分、语言模型等
  - 但是，整个框架仍然是统计机器翻译
- 基于神经元网络的端到端建模出现在2013-2015，被称为Neural Machine Translation (NMT)，一些代表性工作：

时间	作者	论文
2013	Kalchbrenner和 Blunsom	Recurrent Continuous Translation Models
2014	Sutskever等	Sequence to Sequence Learning with neural networks
2014	Bahdanau等	Neural Machine Translation by Jointly Learning to Align and Translate
2014	Cho等	On the Properties of Neural Machine Translation
2015	Jean等	On Using Very Large Target Vocabulary for Neural Machine Translation
2015	Luong等	Effective Approaches to Attention-based Neural Machine Translation

崛起

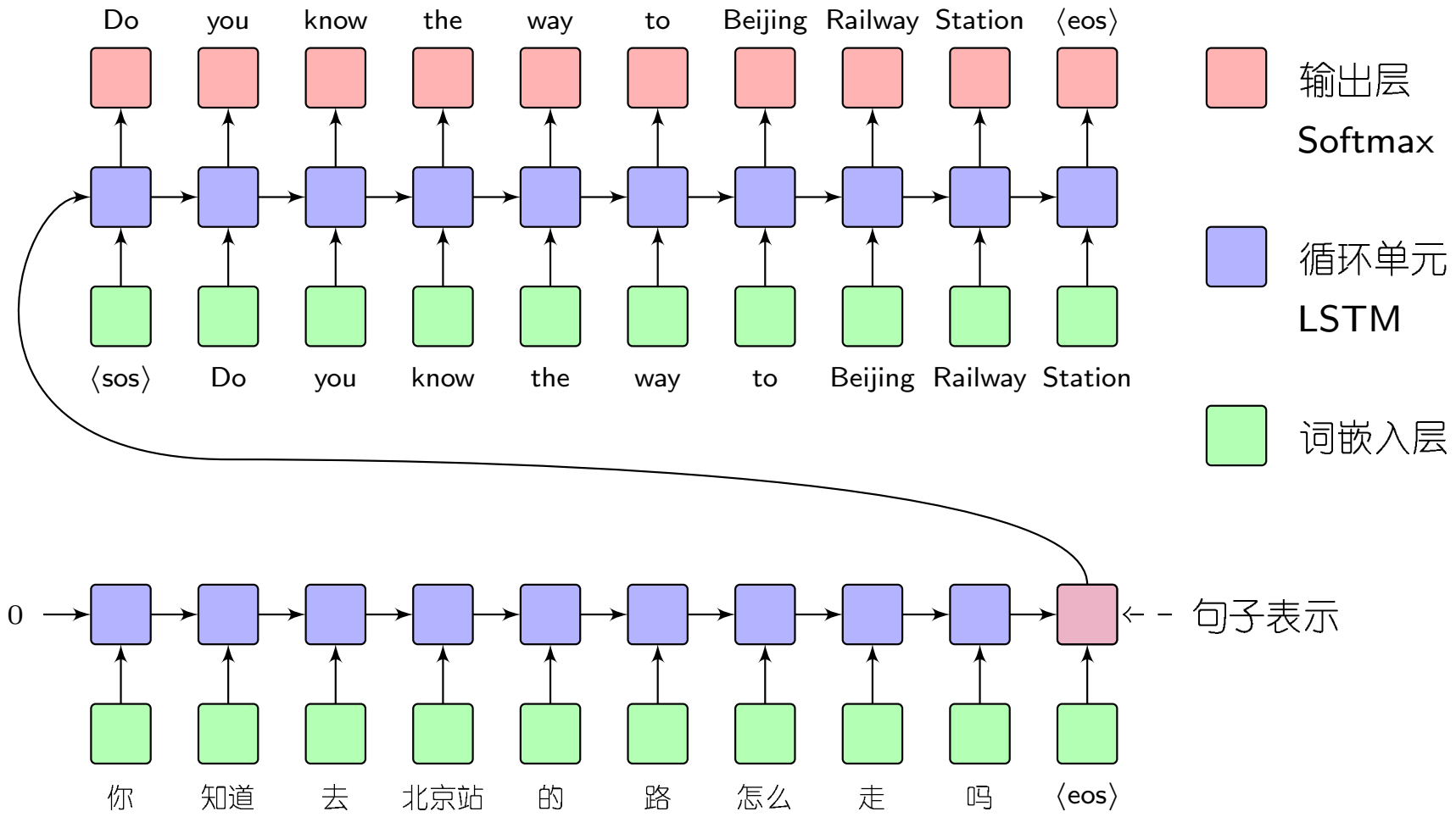
- 2015年前统计机器翻译(SMT)在NLP是具有统治力的
  - 当时的NMT系统还很初级，被SMT碾压
  - 大多数的认知还没有进化到NMT时代，甚至Kalchbrenner等人早期的报告也被人质疑

崛起

- 2015年前统计机器翻译(SMT)在NLP是具有统治力的
  - 当时的NMT系统还很初级，被SMT碾压
  - 大多数的认知还没有进化到NMT时代，甚至Kalchbrenner等人早期的报告也被人质疑
- 2016年情况大有改变，当时非常受关注的一项工作是Google上线了神经机器翻译系统GNMT
  - 在GNMT前后，百度、微软、小牛翻译等也分别推出了自己的神经机器翻译系统，出现了百花齐放的局面

基于循环神经网络的翻译模型

- 一种简单的模型：用循环神经网络进行编码和解码
  - 编码端是一个RNN，最后一个隐层状态被看做句子表示
  - 解码端也是一个RNN，利用编码结果逐词解码出译文



数学建模

- 对于源语言序列$\mathbf{x} = \{x_1, x_2, ..., x_m\}$，生成目标语序列$\mathbf{y} = \{y_1, y_2, ..., y_n\}$的概率可以被描述为

$$\log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

根据源于句子$\mathbf{x}$和已生成的译文$\mathbf{y}_{<j} = \{y_1, y_2, ..., y_{j-1}\}$生成第$j$个译文$y_j$

数学建模

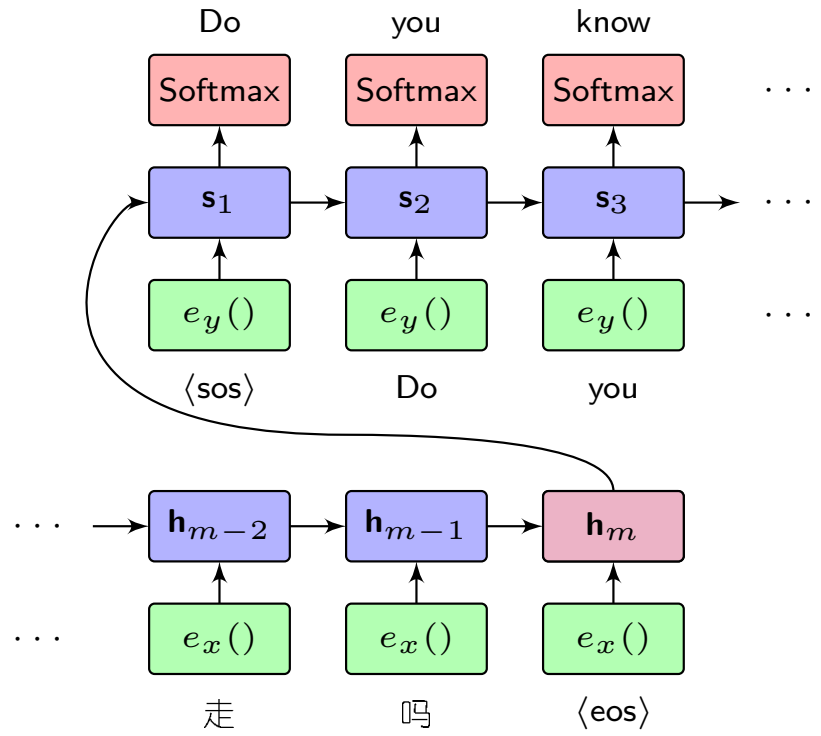
- 对于源语言序列$\mathbf{x} = \{x_1, x_2, ..., x_m\}$，生成目标语序列$\mathbf{y} = \{y_1, y_2, ..., y_n\}$的概率可以被描述为

$$\log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

  根据源于句子$\mathbf{x}$和已生成的译文$\mathbf{y}_{<j} = \{y_1, y_2, ..., y_{j-1}\}$生成第$j$个译文$y_j$

- **核心**：如何求解$\mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$。在这个循环神经网络模型中，有三个步骤
  1. 输入的单词用分布式表示，如$\mathbf{x}$被表示为词向量序列$e_x(\mathbf{x})$，同理$\mathbf{y}_{<j}$被表示为$e_y(\mathbf{y}_{<j})$
  2. 源语言句子被一个RNN编码为一个表示$\mathbf{C}$，如前面的例子中是一个实数向量
  3. 目标端解码用另一个RNN，因此生成y_j时只考虑前一个状态$\mathbf{s}_{j-1}$（这里，$\mathbf{s}_{j-1}$表示RNN第$j-1$步骤的隐层状态）

数学建模(续)


Do
you
know
Softmax
Softmax
Softmax
$\mathbf{s}_1$
$\mathbf{s}_2$
$\mathbf{s}_3$
$e_y()$
$e_y()$
$e_y()$
⟨sos⟩
Do
you
$\mathbf{h}_{m-2}$
$\mathbf{h}_{m-1}$
$\mathbf{h}_m$
$e_x()$
$e_x()$
$e_x()$
走
吗
⟨eos⟩

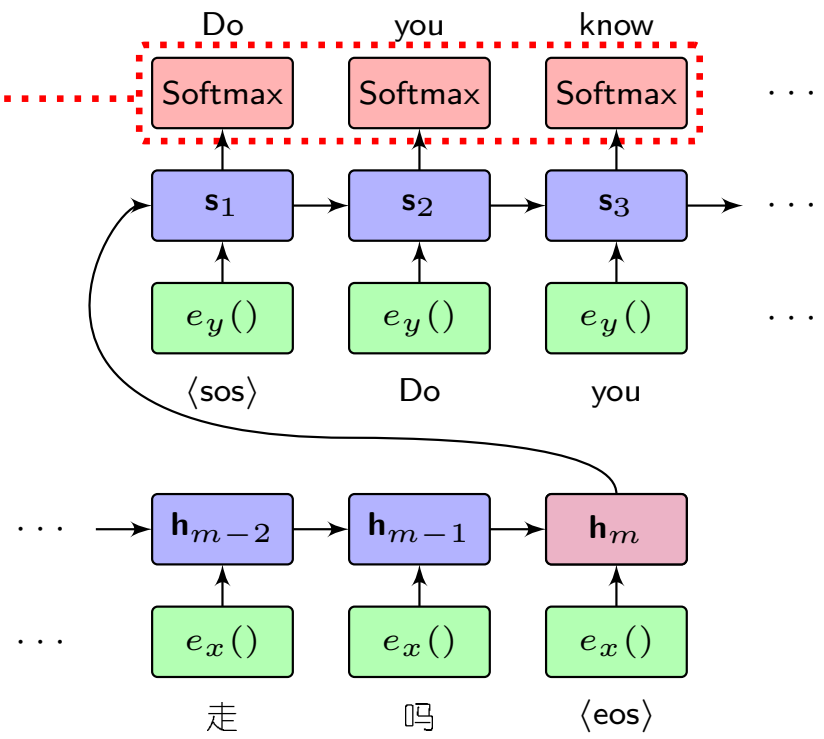
数学建模(续)

基于RNN的隐层状态$\mathbf{s}_j$

预测目标词的概率

通常，用Softmax函数

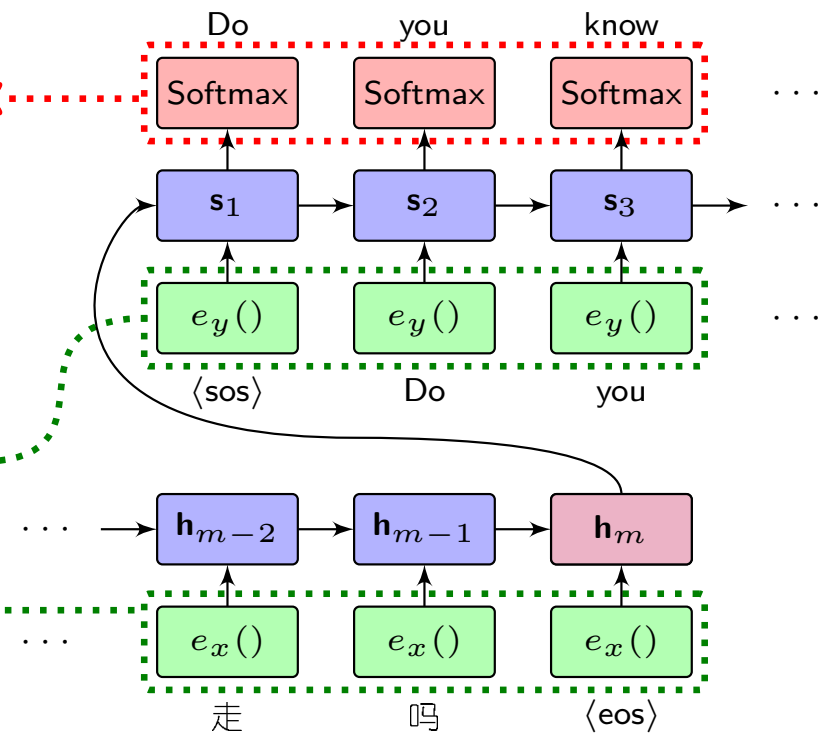
实现 $\mathrm{P}(y_j|...)$



数学建模(续)

基于RNN的隐层状态$\mathbf{s}_j$
预测目标词的概率
通常，用Softmax函数
实现 $\mathrm{P}(y_j|...)$

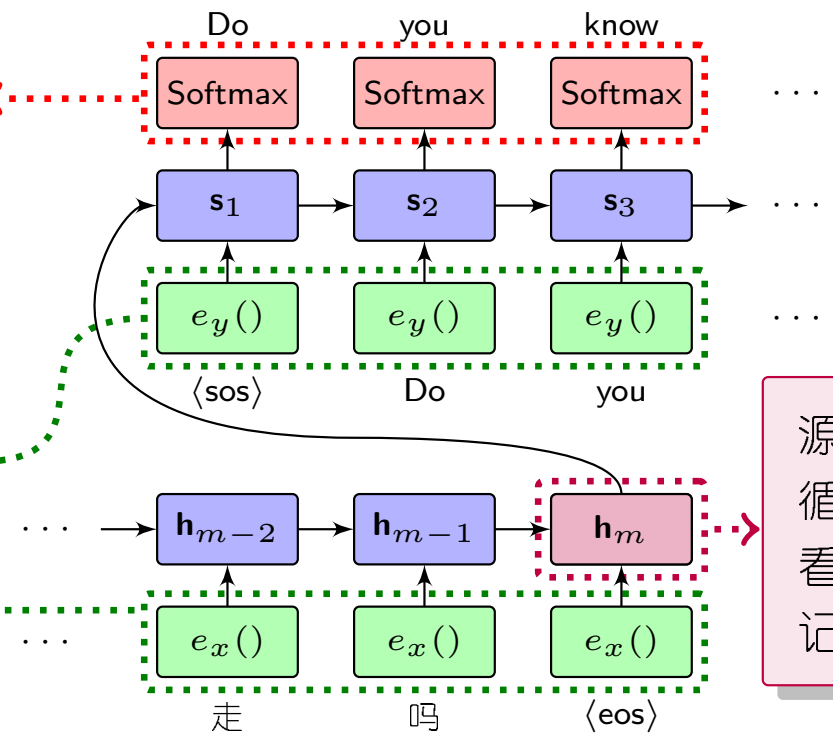
每个词的one-hot
离散化表示都被转化为
实数向量，即词嵌入
($e_x()$和$e_y()$函数)



数学建模(续)

基于RNN的隐层状态$\mathbf{s}_j$预测目标词的概率
通常，用Softmax函数实现 $\mathrm{P}(y_j|...)$

每个词的one-hot离散化表示都被转化为实数向量，即词嵌入
($e_x()$和$e_y()$函数)

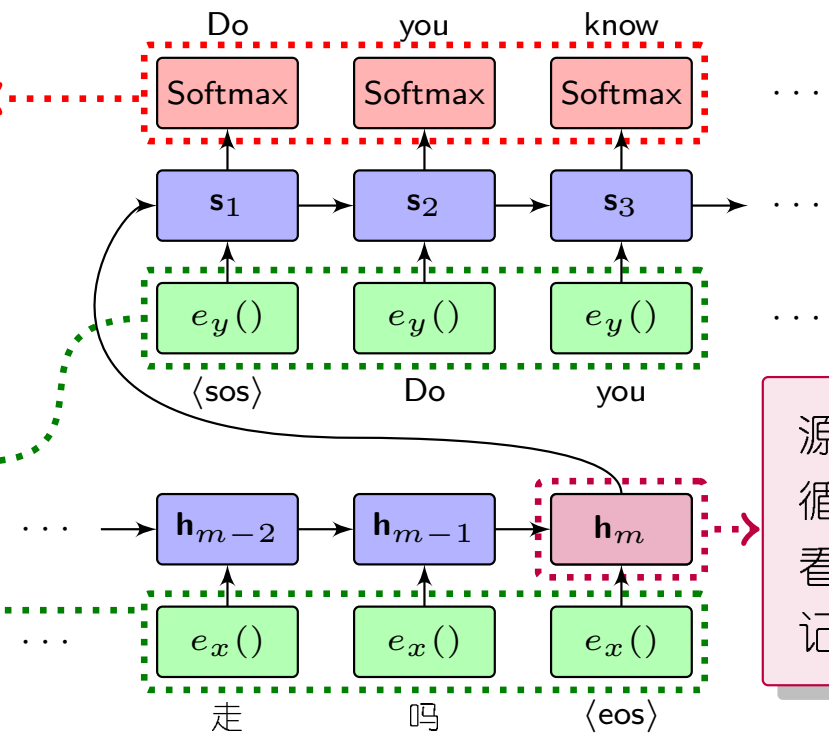


源语编码器最后一个循环单元的输出被看作是句子的表示，记为**C**

数学建模(续)

基于RNN的隐层状态$\mathbf{s}_j$
预测目标词的概率
通常，用Softmax函数
实现 $\mathrm{P}(y_j|...)$

每个词的one-hot
离散化表示都被转化为
实数向量，即词嵌入
($e_x()$和$e_y()$函数)



源语编码器最后一个
循环单元的输出被
看作是句子的表示，
记为$\mathbf{C}$

- 可以重新定义

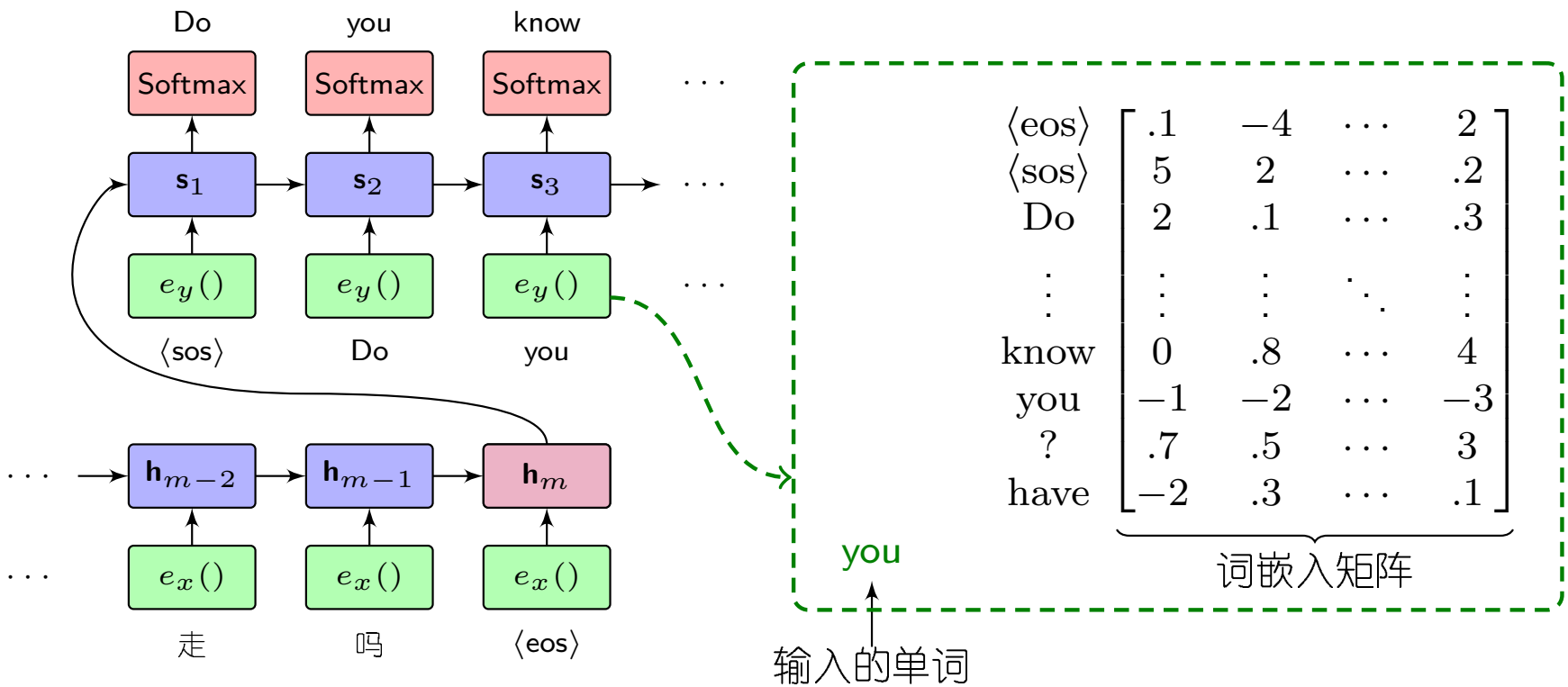
$$\mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x}) \triangleq \mathrm{P}(y_j|\mathbf{s}_{j-1}, y_{j-1}, \mathbf{C})$$

对于上图中的模型，进一步化简为：

$$\mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x}) \triangleq \begin{cases} \mathrm{P}(y_j|\mathbf{C}, y_{j-1}) & j = 1 \\ \mathrm{P}(y_j|\mathbf{s}_{j-1}, y_{j-1}) & j > 1 \end{cases}$$

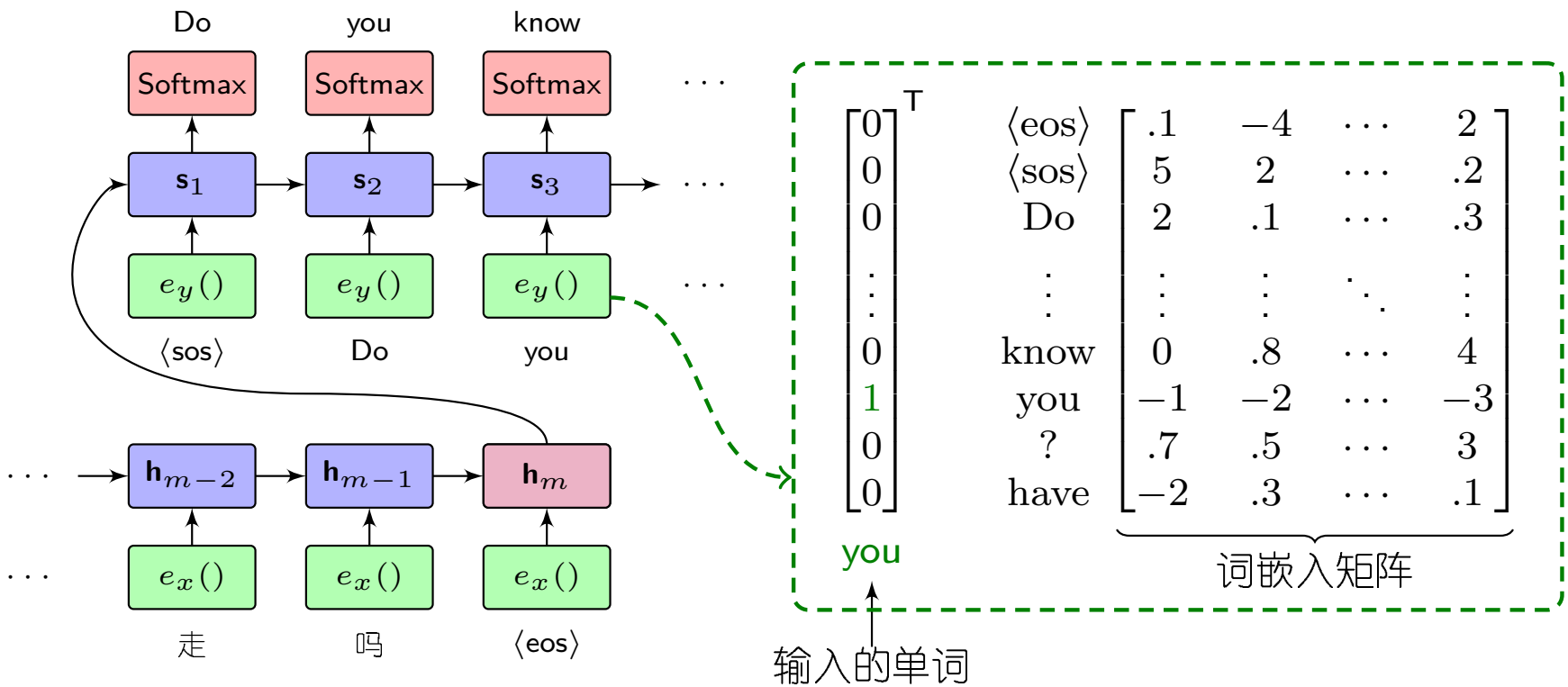
模块1：词嵌入层

- 词嵌入的作用是把离散化的单词表示转换为连续空间上的分布式表示



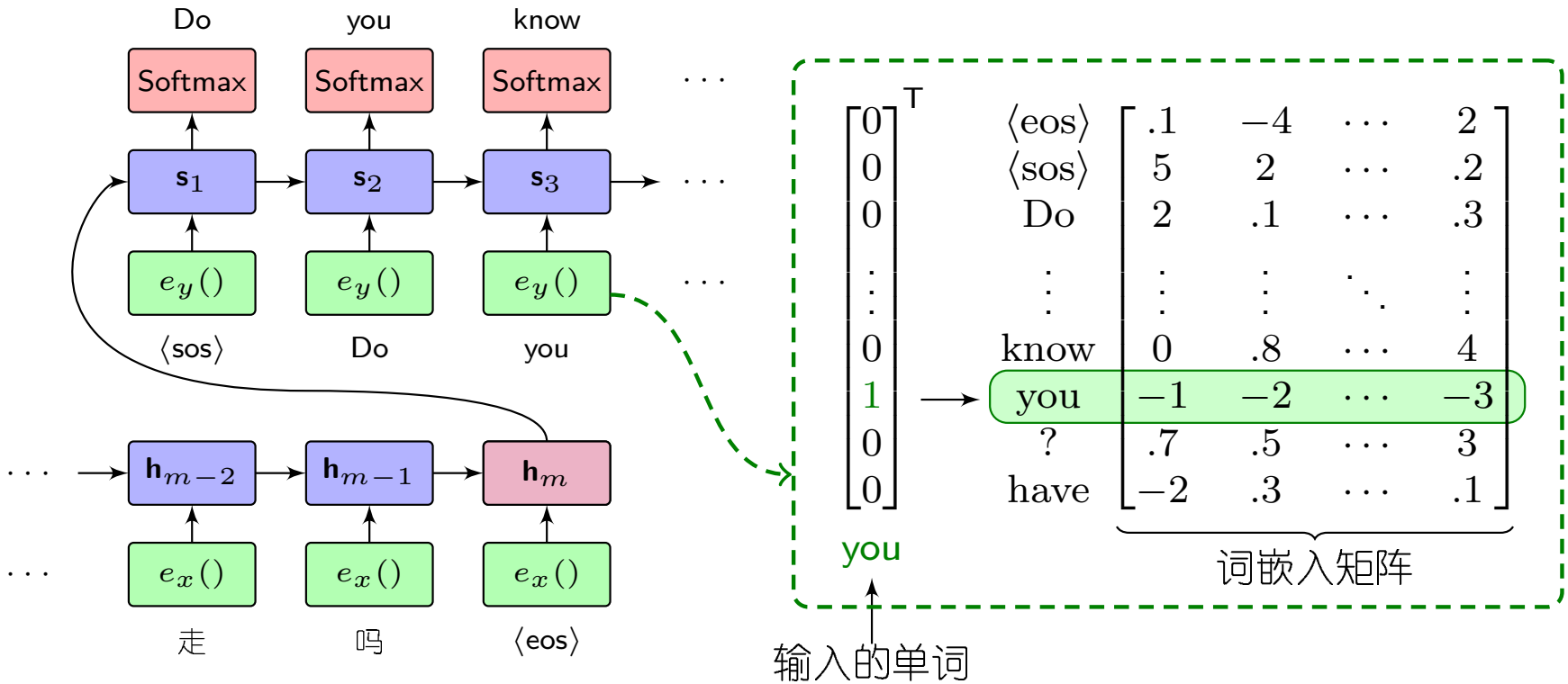
模块1：词嵌入层

- 词嵌入的作用是把离散化的单词表示转换为连续空间上的分布式表示
  - 把输入的词转换成唯一对应的词表大小的0-1向量



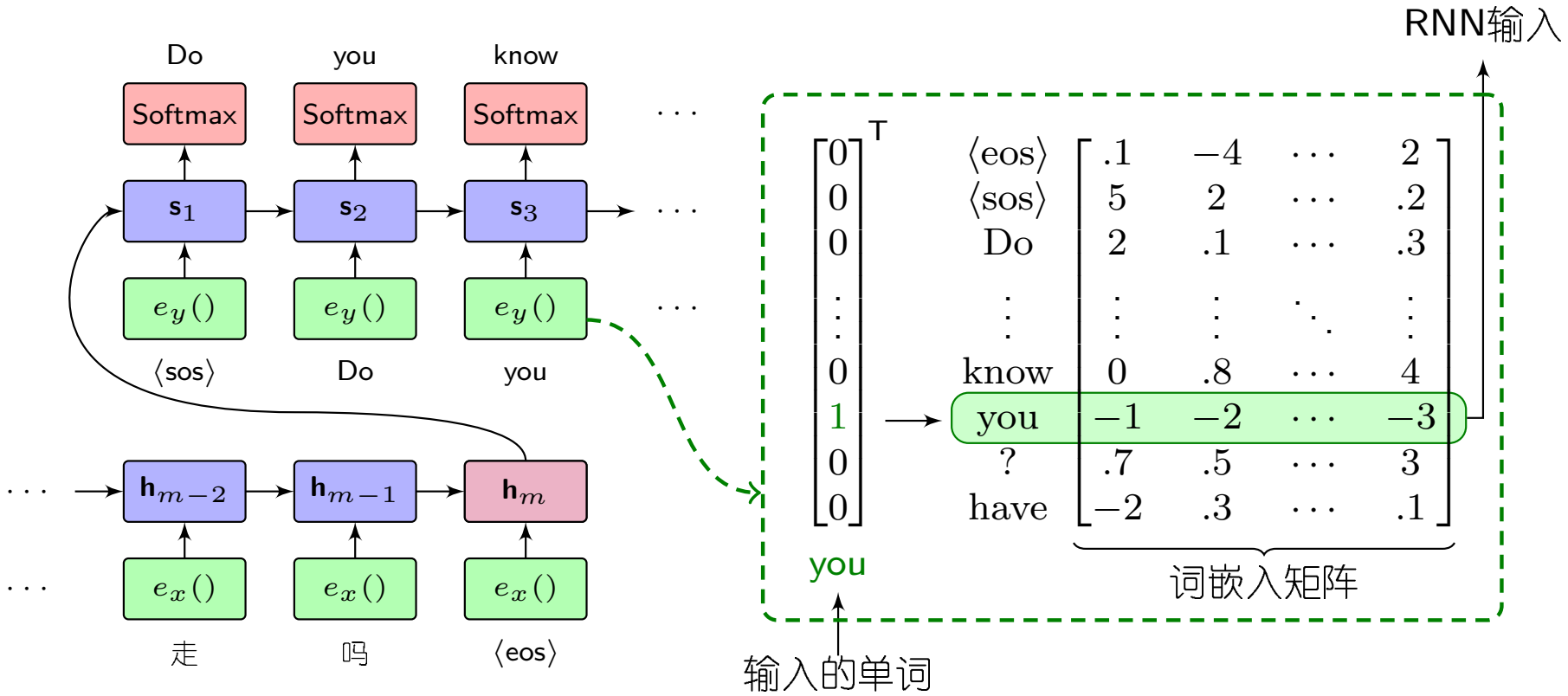
模块1：词嵌入层

- 词嵌入的作用是把离散化的单词表示转换为连续空间上的分布式表示
  - 把输入的词转换成唯一对应的词表大小的0-1向量
  - 根据0-1向量，从词嵌入矩阵中取出对应的词嵌入$e()$



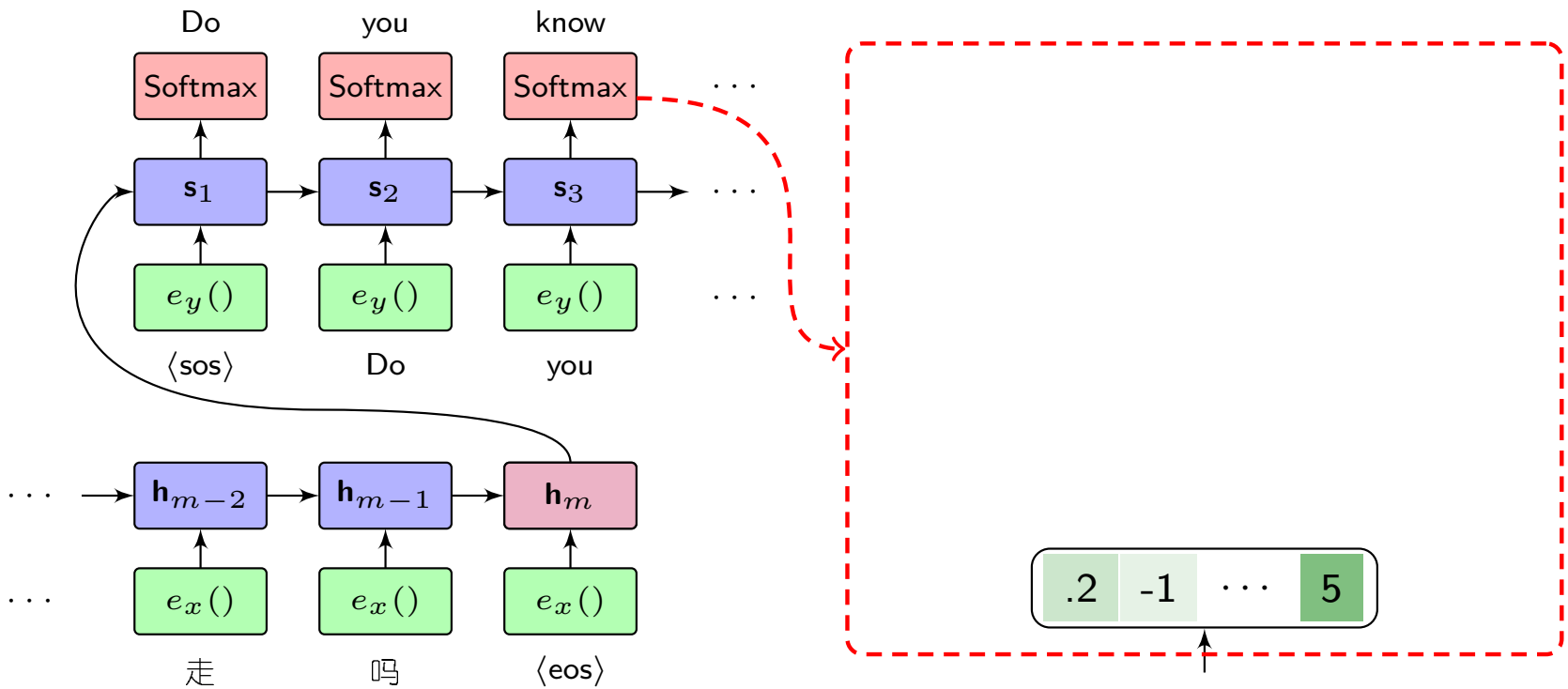
模块1：词嵌入层

- 词嵌入的作用是把离散化的单词表示转换为连续空间上的分布式表示
  - 把输入的词转换成唯一对应的词表大小的0-1向量
  - 根据0-1向量，从词嵌入矩阵中取出对应的词嵌入$e()$
  - 取出的词嵌入$e()$作为循环神经网络的输入



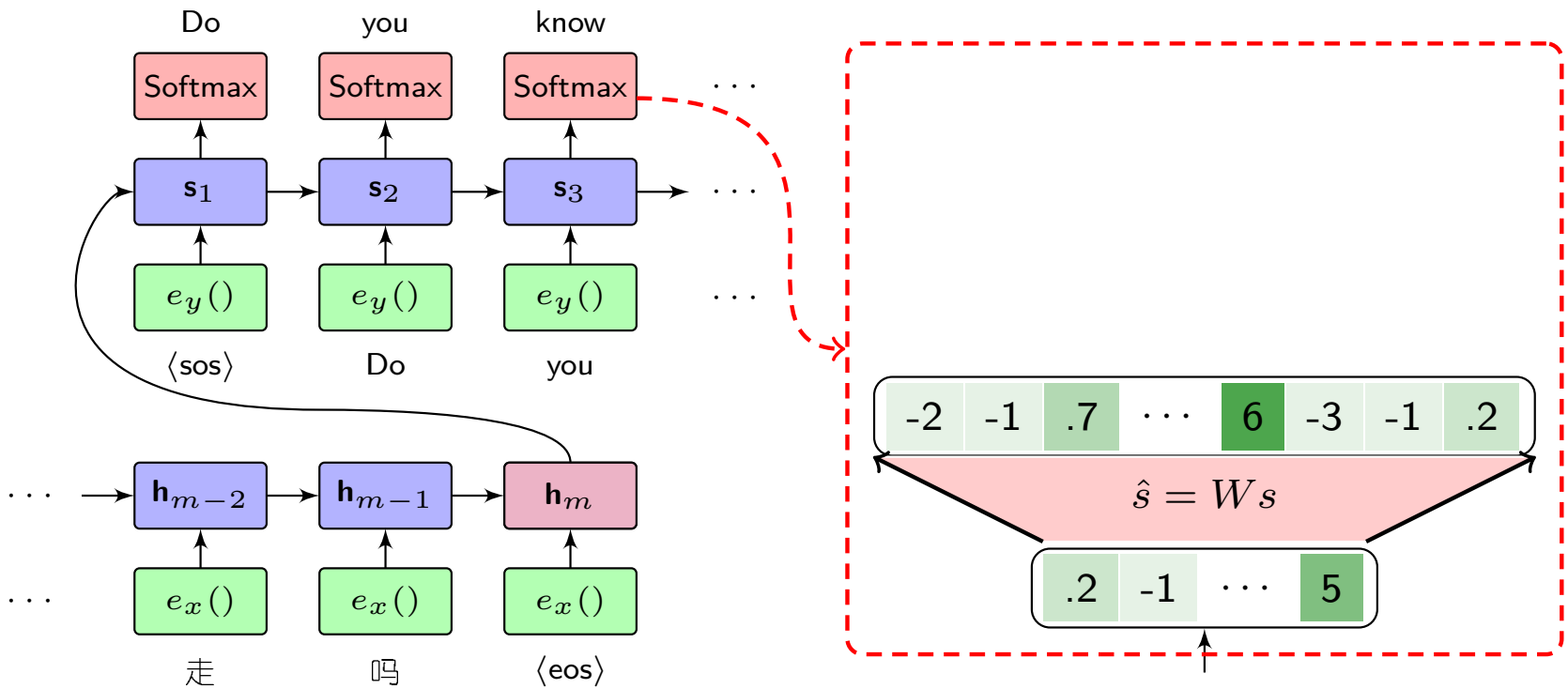
模块2：输出层

- 输出层需要得到每个目标语单词的生成概率，进而选取概率最高的词作为输出。但RNN中的隐藏层并不会输出单词概率，而是输出s，其每一行对应一个单词表示



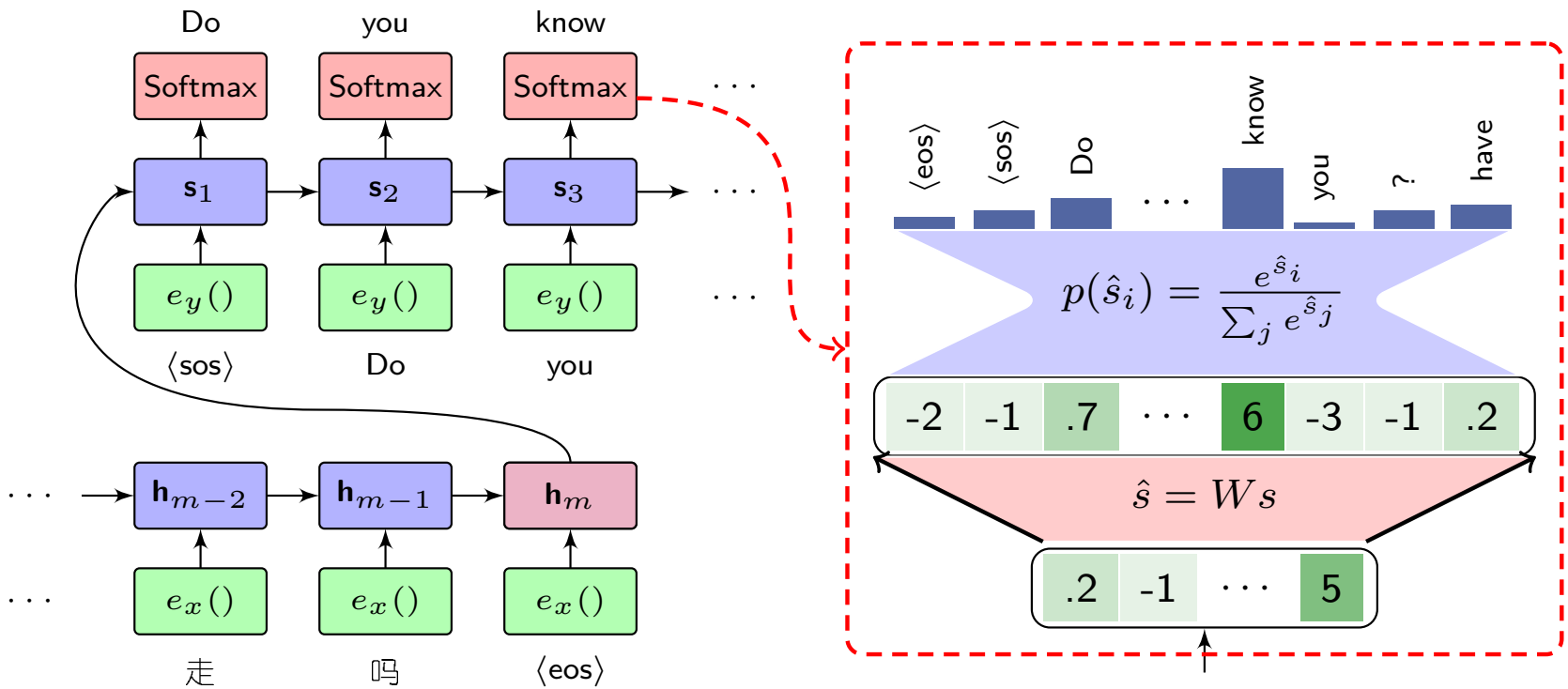
模块2：输出层

- 输出层需要得到每个目标语单词的生成概率，进而选取概率最高的词作为输出。但RNN中的隐藏层并不会输出单词概率，而是输出s，其每一行对应一个单词表示
  - s经过权重矩阵W变成$\hat{s}$，其隐藏层维度变换成词表的大小



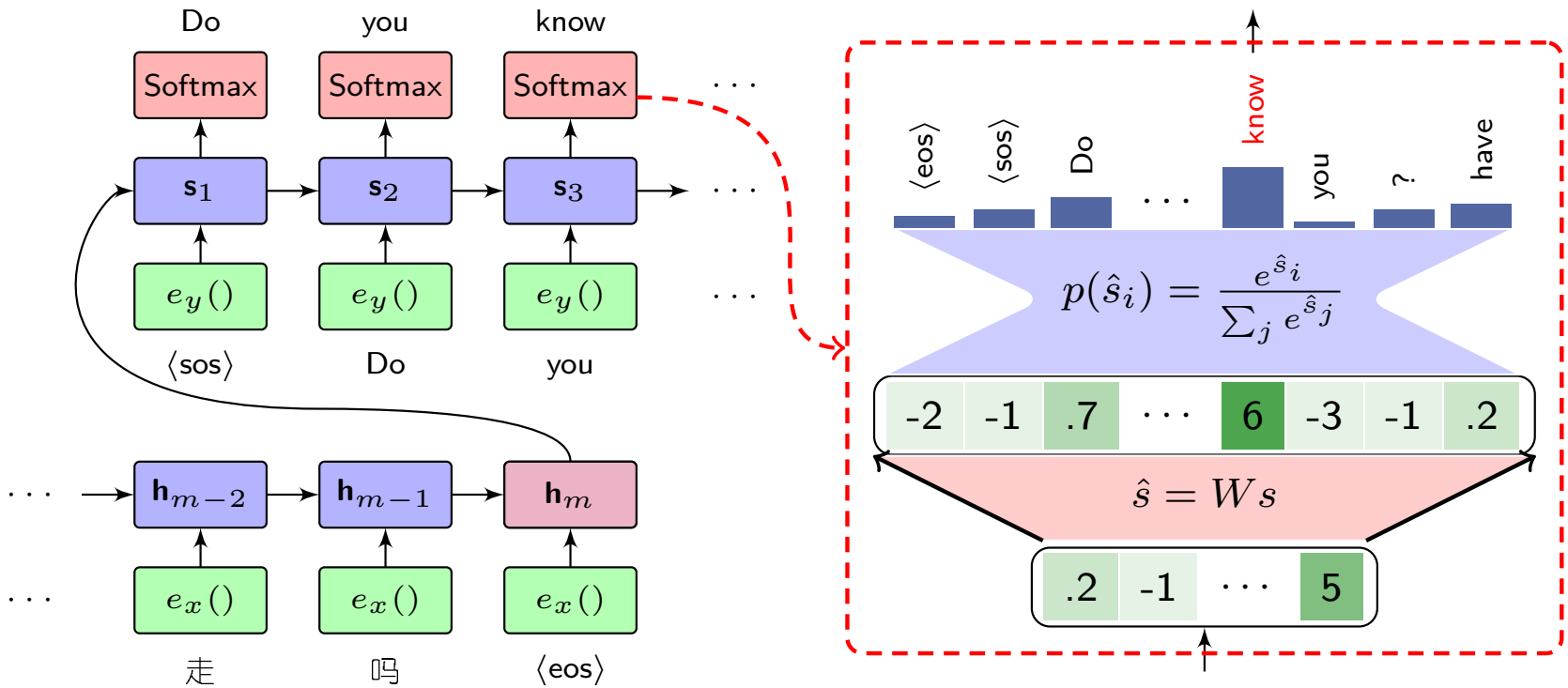
模块2：输出层

- 输出层需要得到每个目标语单词的生成概率，进而选取概率最高的词作为输出。但RNN中的隐藏层并不会输出单词概率，而是输出s，其每一行对应一个单词表示
  - s经过权重矩阵W变成$\hat{s}$，其隐藏层维度变换成词表的大小
  - $\hat{s}$经过Softmax变换得到不同词作为输出的概率，即单词i的概率$p_i = \text{Softmax}(i) = \frac{e^{\hat{s}_i}}{\sum_j e^{\hat{s}_j}}$



模块2：输出层

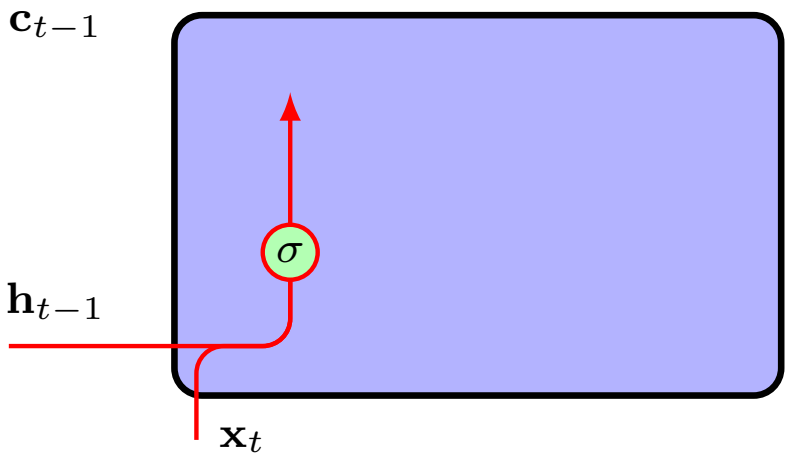
- 输出层需要得到每个目标语单词的生成概率，进而选取概率最高的词作为输出。但RNN中的隐藏层并不会输出单词概率，而是输出s，其每一行对应一个单词表示
  - s经过权重矩阵W变成$\hat{s}$，其隐藏层维度变换成词表的大小
  - $\hat{s}$经过Softmax变换得到不同词作为输出的概率，即单词i的概率$p_i = \text{Softmax}(i) = \frac{e^{\hat{s}_i}}{\sum_j e^{\hat{s}_j}}$



模块3：循环单元 - 长短时记忆模型(LSTM)

- LSTM是最常用的循环单元结构，它一种典型的记忆网络，通过“门”单元来动态地选择遗忘多少以前的信息

遗忘门
$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$

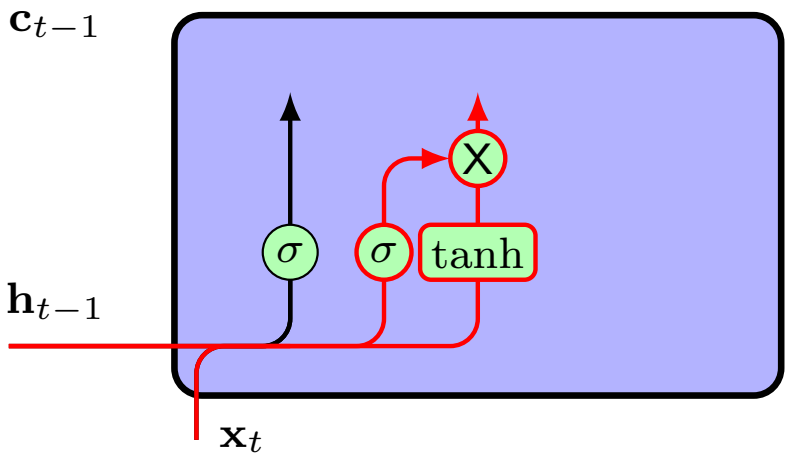


*$\mathbf{x}_t$：上一层的输出，$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态
*$\mathbf{c}_{t-1}$：同一层上一时刻的记忆

模块3：循环单元 - 长短时记忆模型(LSTM)

- LSTM是最常用的循环单元结构，它一种典型的记忆网络，通过“门”单元来动态地选择遗忘多少以前的信息

遗忘门
$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$



输入门
$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$
$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$

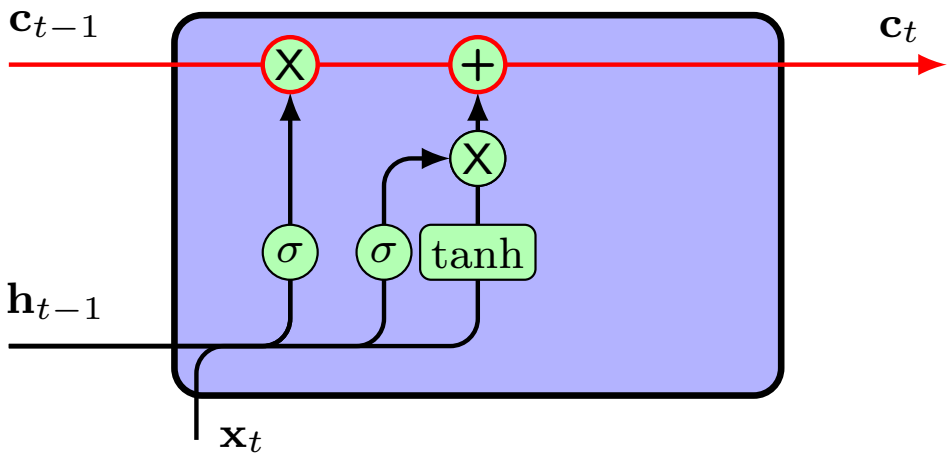
*$\mathbf{x}_t$：上一层的输出，$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态
*$\mathbf{c}_{t-1}$：同一层上一时刻的记忆

模块3：循环单元 - 长短时记忆模型(LSTM)

- LSTM是最常用的循环单元结构，它一种典型的记忆网络，通过“门”单元来动态地选择遗忘多少以前的信息

遗忘门
$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$

记忆更新
$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \hat{\mathbf{c}}_t$



输入门
$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$
$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$

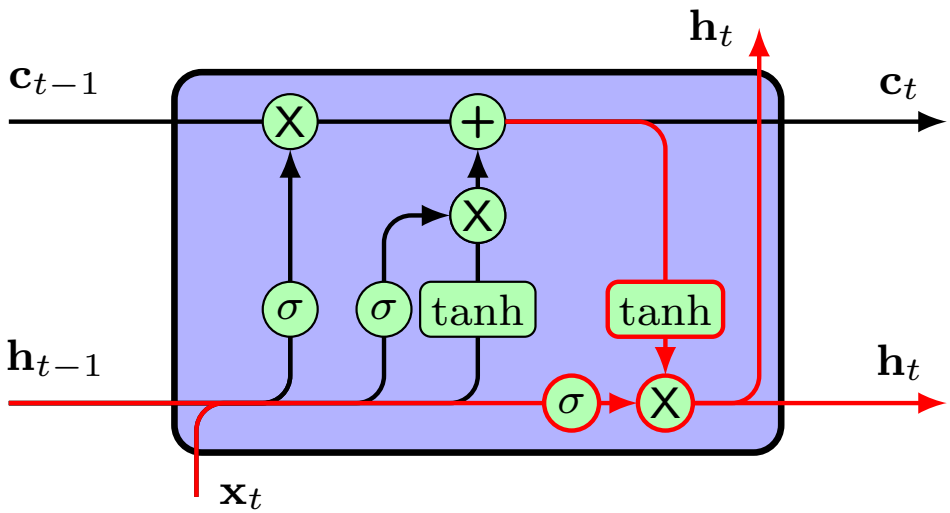
*$\mathbf{x}_t$：上一层的输出，$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态
*$\mathbf{c}_{t-1}$：同一层上一时刻的记忆

模块3：循环单元 - 长短时记忆模型(LSTM)

- LSTM是最常用的循环单元结构，它一种典型的记忆网络，通过“门”单元来动态地选择遗忘多少以前的信息

遗忘门
$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$

记忆更新
$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \hat{\mathbf{c}}_t$



输入门
$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$
$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$

输出门
$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$
$\mathbf{h}_t = \mathbf{o}_t \cdot \tanh(\mathbf{c}_t)$

*$\mathbf{x}_t$：上一层的输出，$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态
*$\mathbf{c}_{t-1}$：同一层上一时刻的记忆

模块3：循环单元 - 长短时记忆模型(LSTM)

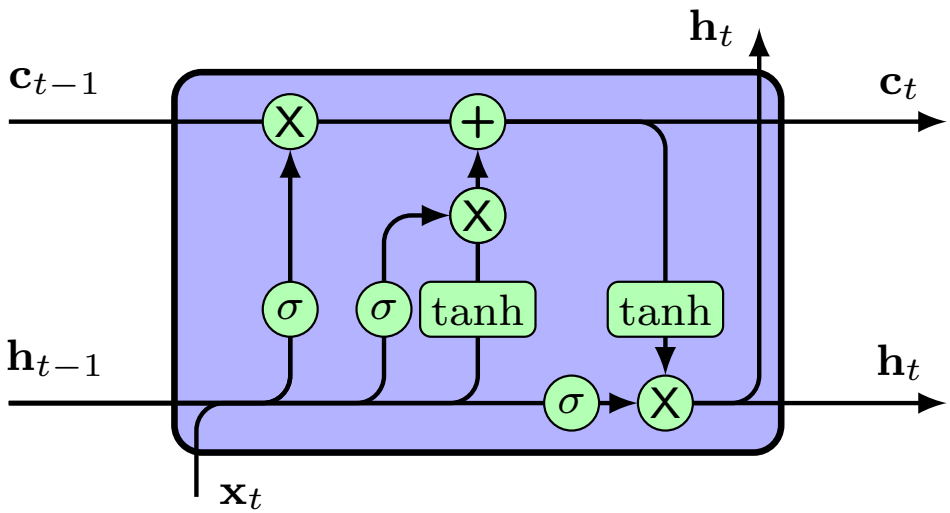
- LSTM是最常用的循环单元结构，它一种典型的记忆网络，通过“门”单元来动态地选择遗忘多少以前的信息

遗忘门

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

记忆更新

$$\mathbf{c}_t = \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \hat{\mathbf{c}}_t$$



输入门

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$
$$\hat{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$

输出门

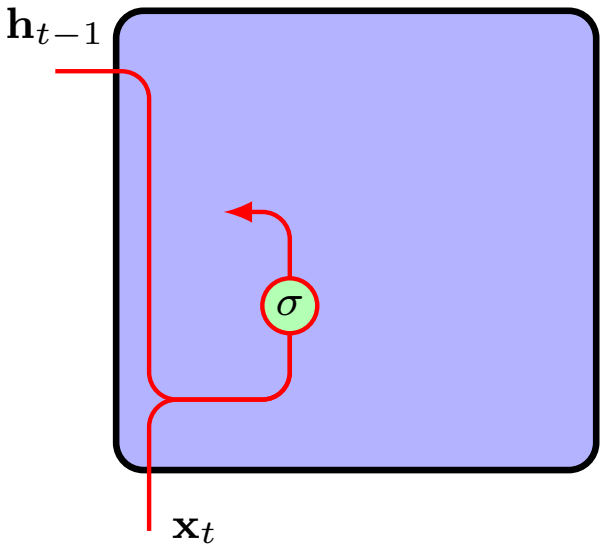
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$
$$\mathbf{h}_t = \mathbf{o}_t \cdot \tanh(\mathbf{c}_t)$$

*$\mathbf{x}_t$：上一层的输出，$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态

*$\mathbf{c}_{t-1}$：同一层上一时刻的记忆

另一种循环单元 - 门循环单元(GRU)

- GRU是LSTM的一个变种，它把隐藏状态h和记忆c合并成一个隐藏状态h，同时使用了更少的“门”单元，大大提升了计算效率
  - 在NMT中GRU会带来20-25%的速度提升

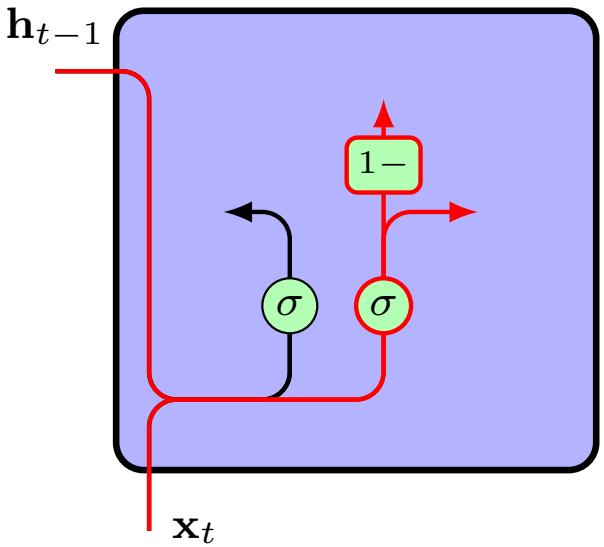


重置门
$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$

*$\mathbf{x}_t$: 上一层的输出
*$\mathbf{h}_{t-1}$: 同一层上一时刻的隐藏状态

另一种循环单元 - 门循环单元(GRU)

- GRU是LSTM的一个变种，它把隐藏状态h和记忆c合并成一个隐藏状态h，同时使用了更少的“门”单元，大大提升了计算效率
  - 在NMT中GRU会带来20-25%的速度提升



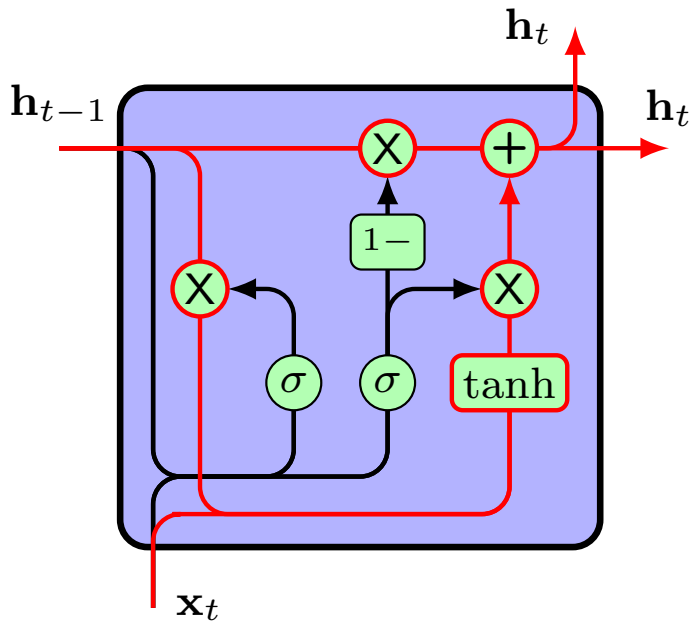
重置门
$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$

更新门
$\mathbf{u}_t = \sigma(\mathbf{W}_u[\mathbf{h}_{t-1}, \mathbf{x}_t])$

*$\mathbf{x}_t$：上一层的输出
*$\mathbf{h}_{t-1}$：同一层上一时刻的隐藏状态

另一种循环单元 - 门循环单元(GRU)

- GRU是LSTM的一个变种，它把隐藏状态h和记忆c合并成一个隐藏状态h，同时使用了更少的“门”单元，大大提升了计算效率
  - 在NMT中GRU会带来20-25%的速度提升



重置门
$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$

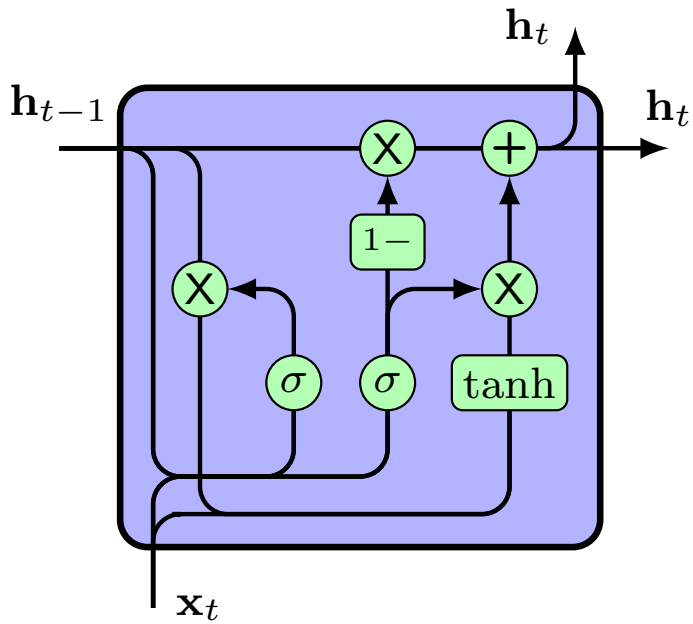
更新门
$\mathbf{u}_t = \sigma(\mathbf{W}_u[\mathbf{h}_{t-1}, \mathbf{x}_t])$

隐藏状态更新
$\hat{\mathbf{h}}_t = \tanh(\mathbf{W}[\mathbf{r}_t \cdot \mathbf{h}_{t-1}, \mathbf{x}_t])$
$\mathbf{h}_t = (1 - \mathbf{u}_t) \cdot \mathbf{h}_{t-1} + \mathbf{u}_t \cdot \hat{\mathbf{h}}_t$

*$\mathbf{x}_t$: 上一层的输出
*$\mathbf{h}_{t-1}$: 同一层上一时刻的隐藏状态

另一种循环单元 - 门循环单元(GRU)

- GRU是LSTM的一个变种，它把隐藏状态h和记忆c合并成一个隐藏状态h，同时使用了更少的“门”单元，大大提升了计算效率
  - 在NMT中GRU会带来20-25%的速度提升



重置门
$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$

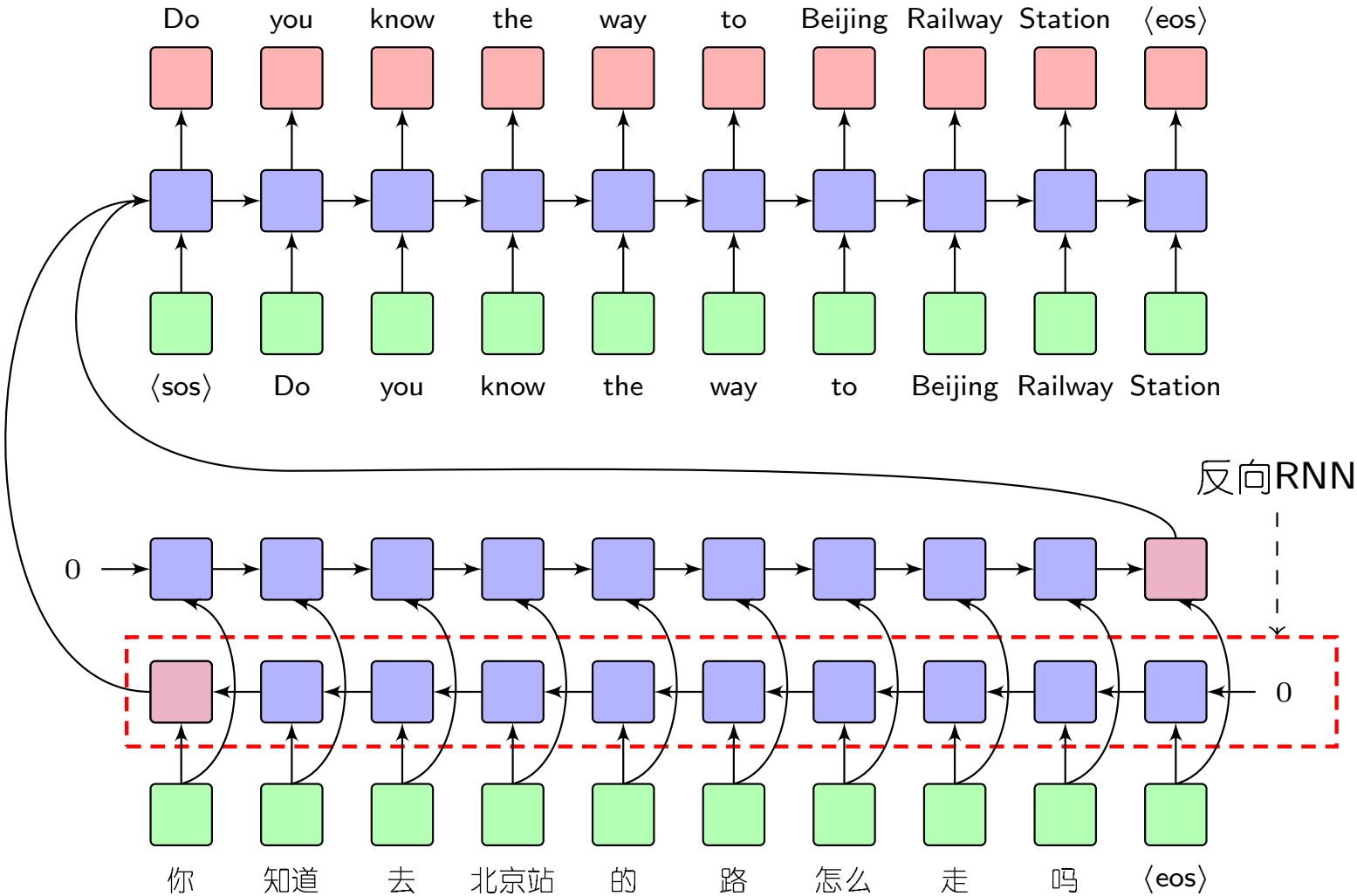
更新门
$\mathbf{u}_t = \sigma(\mathbf{W}_u[\mathbf{h}_{t-1}, \mathbf{x}_t])$

隐藏状态更新
$\hat{\mathbf{h}}_t = \tanh(\mathbf{W}[\mathbf{r}_t \cdot \mathbf{h}_{t-1}, \mathbf{x}_t])$
$\mathbf{h}_t = (1 - \mathbf{u}_t) \cdot \mathbf{h}_{t-1} + \mathbf{u}_t \cdot \hat{\mathbf{h}}_t$

*$\mathbf{x}_t$: 上一层的输出
*$\mathbf{h}_{t-1}$: 同一层上一时刻的隐藏状态

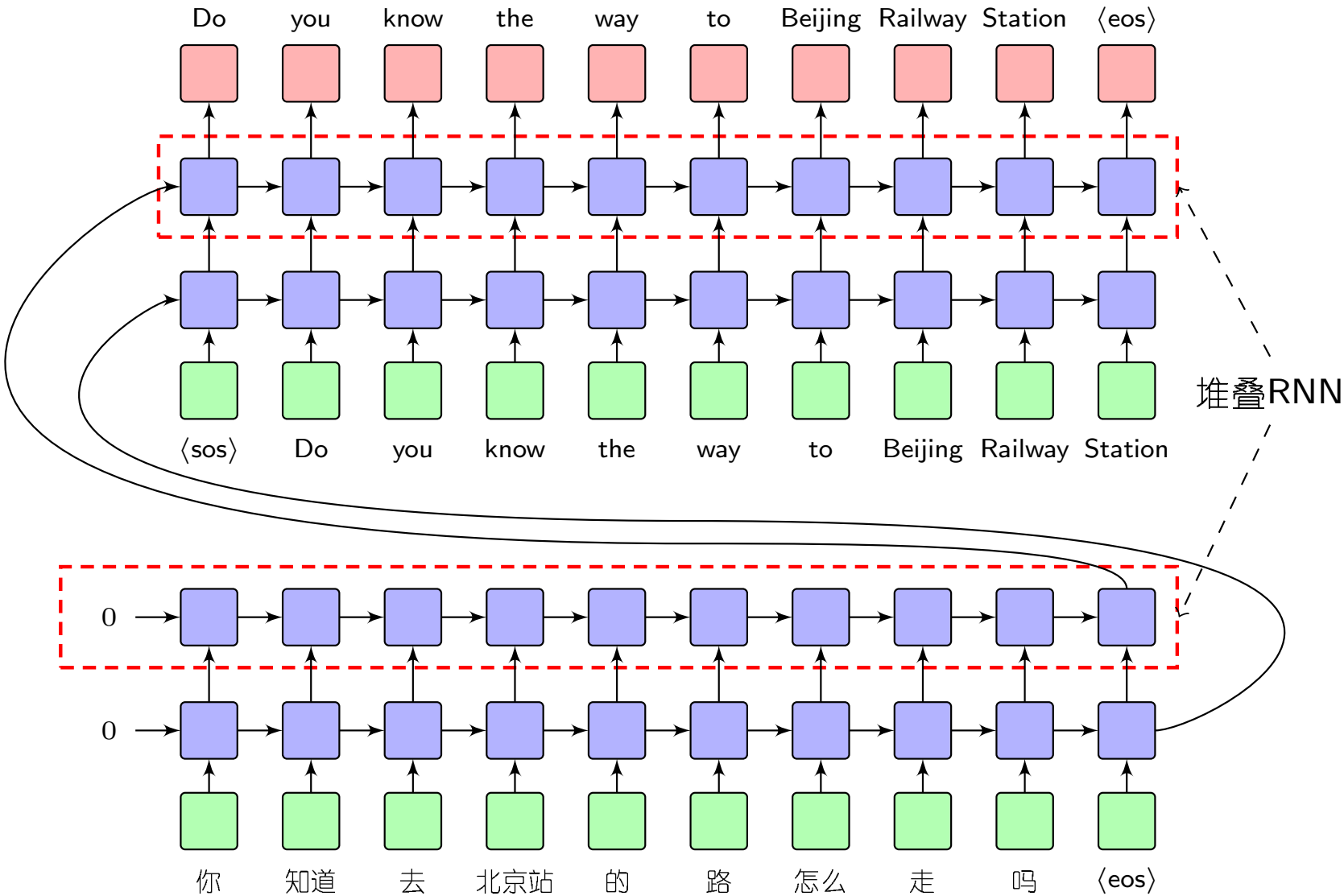
改进 - 双向模型

- 自左向右的模型只考虑了左侧的上下文，因此可以用自右向左的模型对右侧上下文建模
  - 最终将两个模型融合同时送给编码端



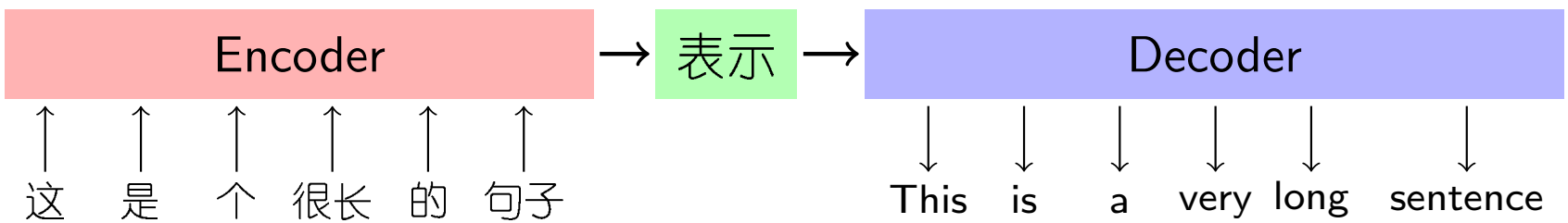
改进 - 多层网络

- 堆叠更多层的网络，可以提升模型的表示能力
  - 常见的NMT系统有2-8层



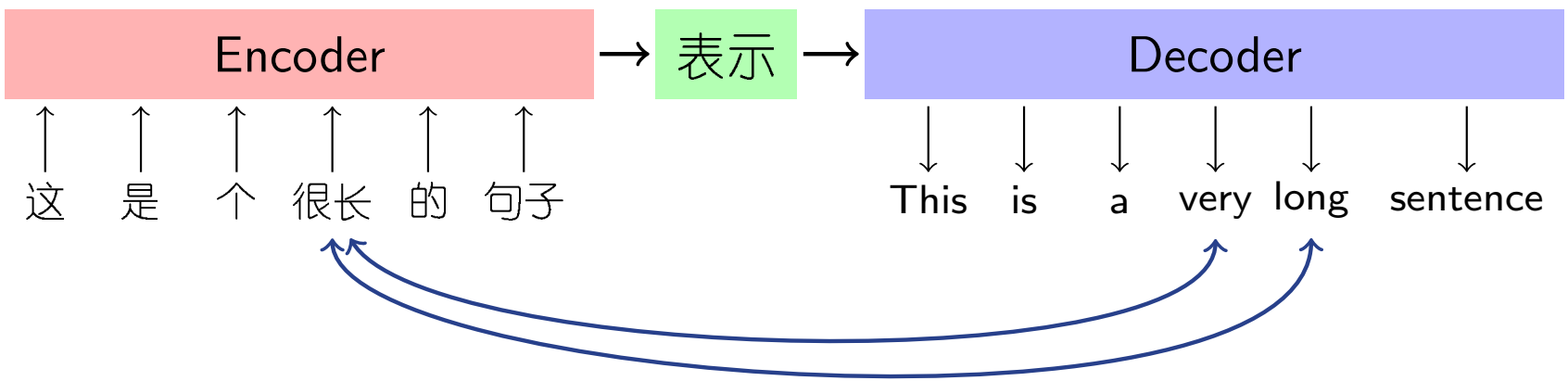
简单的编码器-解码器就足够了？

- 将源语言句子编码为一个实数向量确实很神奇，但是也有明显问题
  - 整个句子编码到一个向量里可能会有信息丢失
  - 缺少源语单词与目标语单词之间的对应。某种意义上讲，一个目标语单词的生成无法区分不同源语单词的贡献



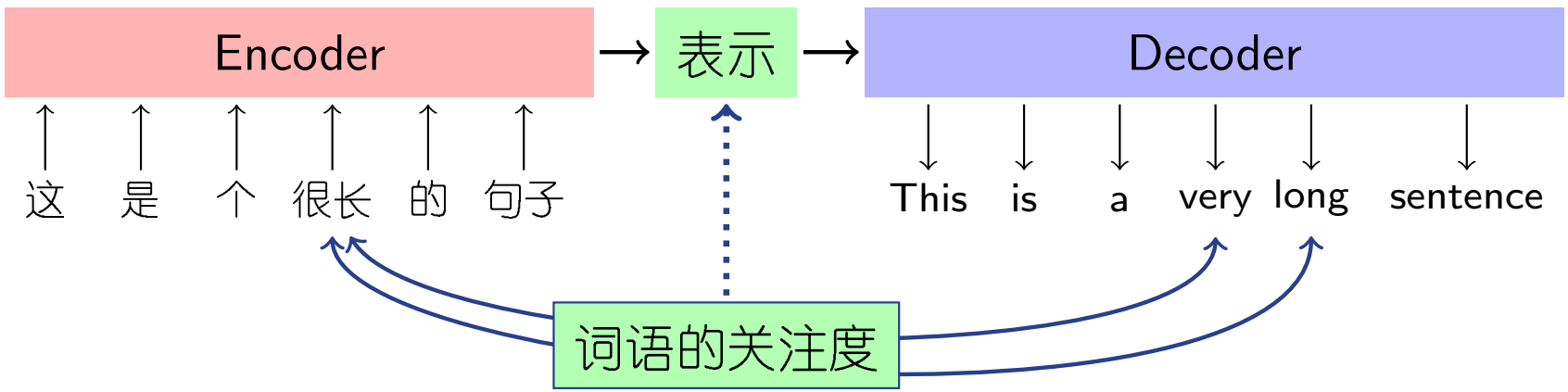
简单的编码器-解码器就足够了？

- 将源语言句子编码为一个实数向量确实很神奇，但是也有明显问题
  - 整个句子编码到一个向量里可能会有信息丢失
  - 缺少源语单词与目标语单词之间的对应。某种意义上讲，一个目标语单词的生成无法区分不同源语单词的贡献
- 但是，翻译是具有很强的局部性的，有些词之间会有更紧密的关系
  - 源语词和目标语词的对应并不是均匀的，甚至非常稀疏
  - 比如，一些短语的生成仅依赖于源文中的少数词



简单的编码器-解码器就足够了？

- 将源语言句子编码为一个实数向量确实很神奇，但是也有明显问题
  - 整个句子编码到一个向量里可能会有信息丢失
  - 缺少源语单词与目标语单词之间的对应。某种意义上讲，一个目标语单词的生成无法区分不同源语单词的贡献
- 但是，翻译是具有很强的局部性的，有些词之间会有更紧密的关系
  - 源语词和目标语词的对应并不是均匀的，甚至非常稀疏
  - 比如，一些短语的生成仅依赖于源文中的少数词
  - 这些关系可以在表示模型中考虑



注意力机制

- 关注的“局部性”在图像处理、语音识别等领域也有广泛讨论，比如，对于下图
  - 关注的顺序：大狗的帽子 → 大狗 → 小狗的帽子 → 小狗
- 人往往不是“均匀地”看图像中的所有区域，翻译是一个道理，生成一个目标语单词时参考的源语单词不会太多

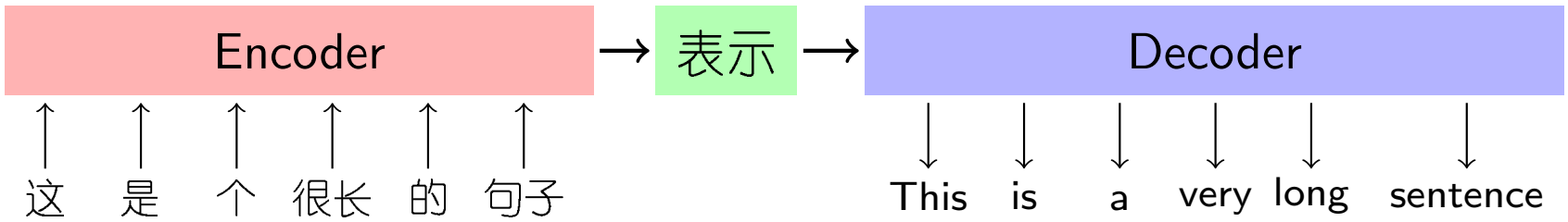
注意力机制

- 关注的“局部性”在图像处理、语音识别等领域也有广泛讨论，比如，对于下图
  - 关注的顺序：大狗的帽子 → 大狗 → 小狗的帽子 → 小狗
- 人往往不是“均匀地”看图像中的所有区域，翻译是一个道理，生成一个目标语单词时参考的源语单词不会太多

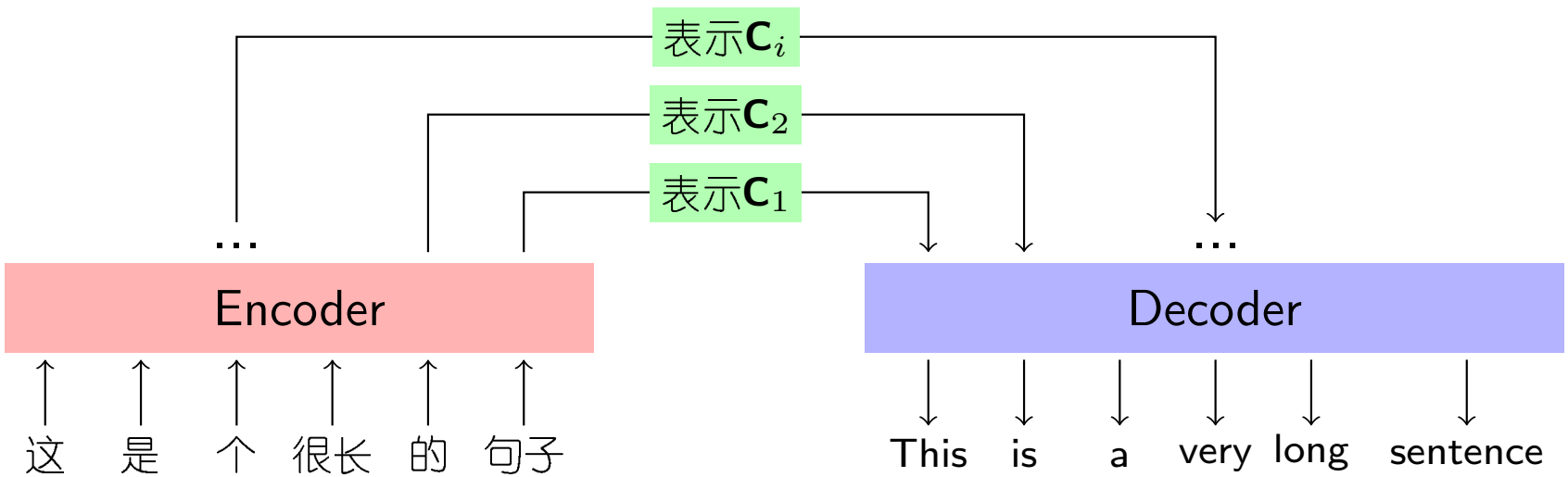
- 注意力机制在机器翻译中已经成功应用，经典的论文
  Neural Machine Translation by Jointly Learning to Align and Translate
  Bahdanau et al., 2015, In Proc of ICLR

神经机器翻译的注意力机制

- 在注意力机制中，每个目标语单词的生成会使用一个动态的源语表示，而非一个统一的固定表示
  - 这里$\mathbf{C}_j$表示第j个目标语单词所使用的源语表示



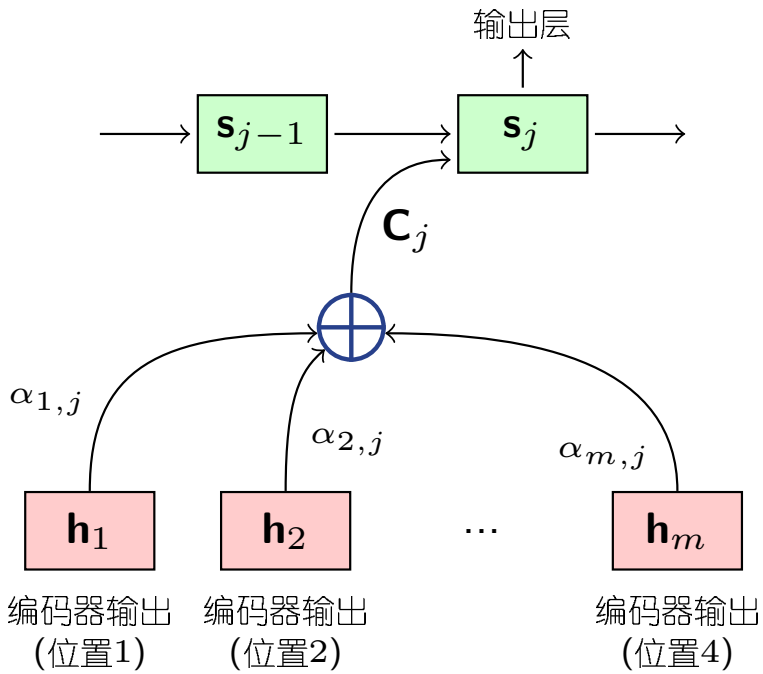
(a) 简单的编码器-解码器框架



(b) 引入注意力机制的编码器-解码器框架

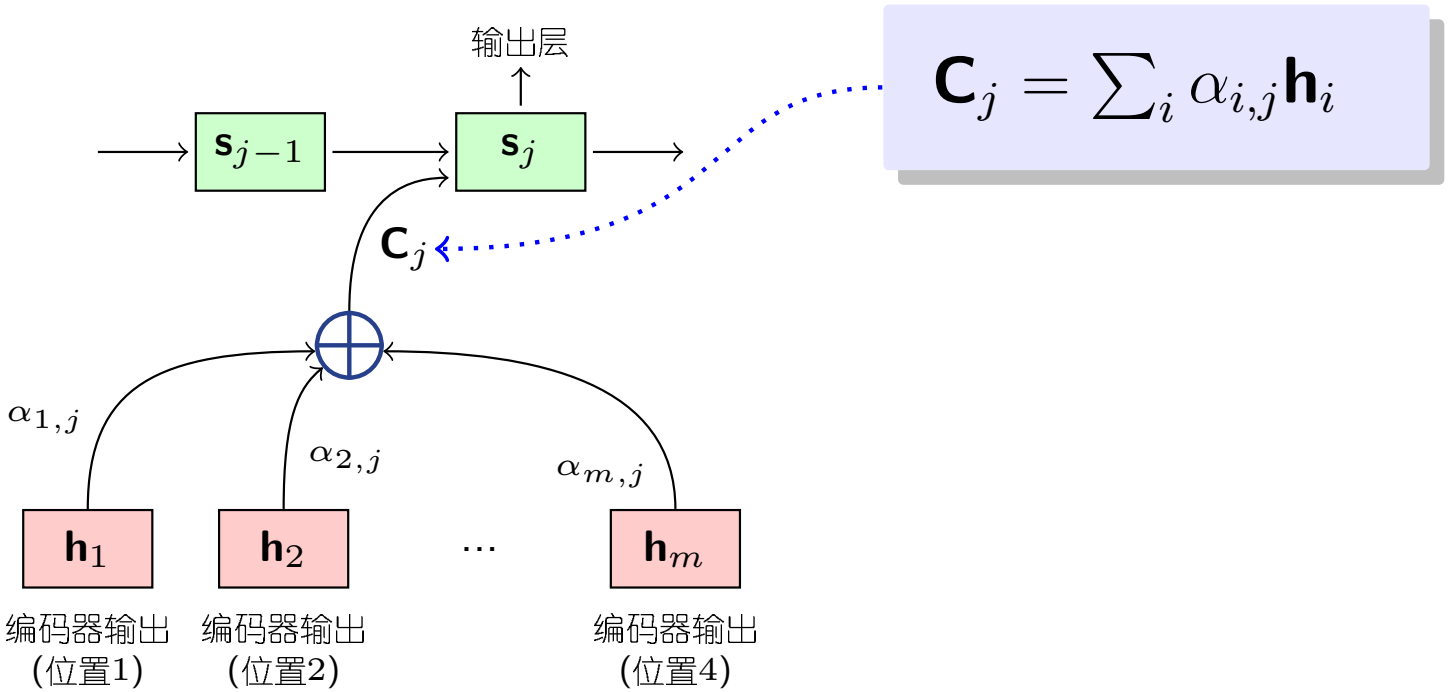
上下文向量$\mathbf{C}_j$

- 对于目标语位置j，$\mathbf{C}_j$是目标语j使用的上下文向量
  - $\mathbf{h}_i$表示编码器第i个位置的隐层状态
  - $\mathbf{s}_j$表示解码器第j个位置的隐层状态



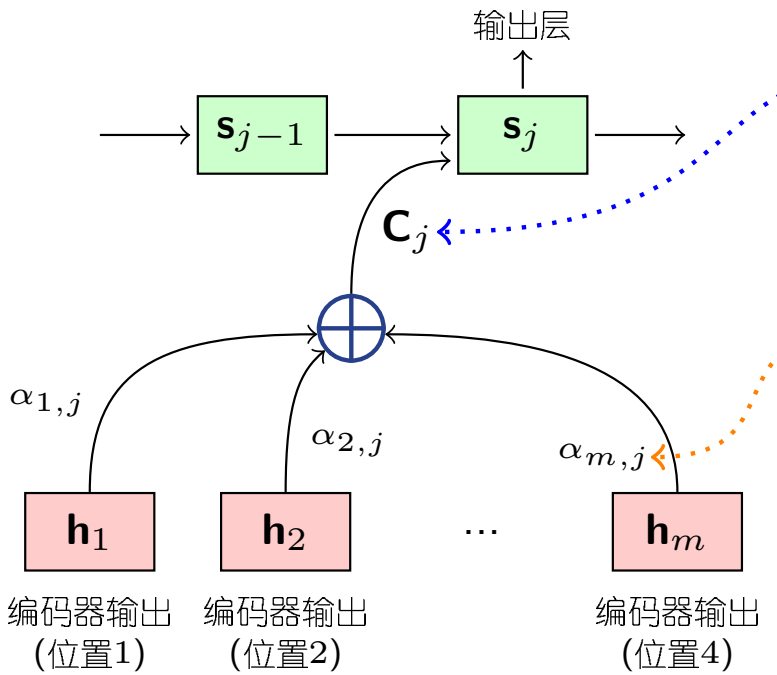
上下文向量$\mathbf{C}_j$

- 对于目标语位置j，$\mathbf{C}_j$是目标语j使用的上下文向量
  - $\mathbf{h}_i$表示编码器第i个位置的隐层状态
  - $\mathbf{s}_j$表示解码器第j个位置的隐层状态
  - $\alpha_{i,j}$表示注意力权重，表示目标语第j个位置与源语第i个位置之间的相关性大小
  - $a(\cdot)$表示注意力函数，计算$\mathbf{s}_{j-1}$和$\mathbf{h}_i$之间的相关性



上下文向量$\mathbf{C}_j$

- 对于目标语位置j，$\mathbf{C}_j$是目标语j使用的上下文向量
  - $\mathbf{h}_i$表示编码器第i个位置的隐层状态
  - $\mathbf{s}_j$表示解码器第j个位置的隐层状态
  - $\alpha_{i,j}$表示注意力权重，表示目标语第j个位置与源语第i个位置之间的相关性大小
  - $a(\cdot)$表示注意力函数，计算$\mathbf{s}_{j-1}$和$\mathbf{h}_i$之间的相关性
  - $\mathbf{C}_j$是所有源语编码表示$\{\mathbf{h}_i\}$的加权求和，权重为$\{\alpha_{i,j}\}$



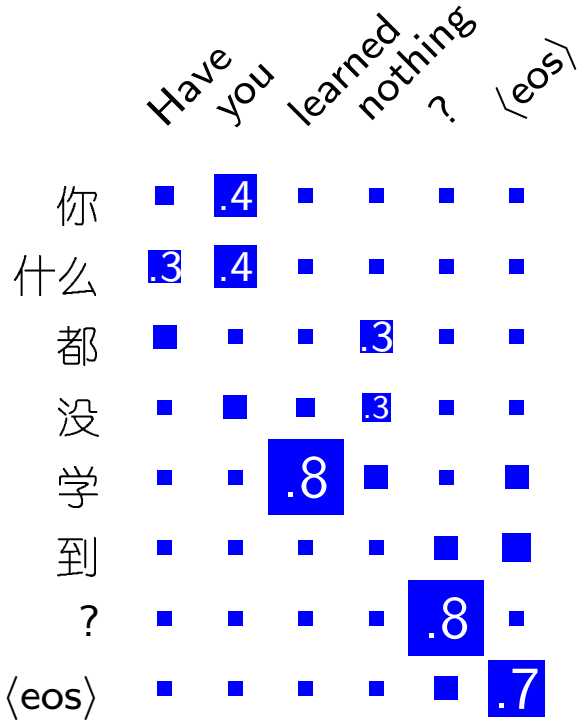
$$\mathbf{C}_j = \sum_i \alpha_{i,j}\mathbf{h}_i$$

$$\alpha_{i,j} = \frac{\exp(\beta_{i,j})}{\sum_{i'}\exp(\beta_{i',j})}$$

$$\beta_{i,j} = a(\mathbf{s}_{j-1}, \mathbf{h}_i)$$

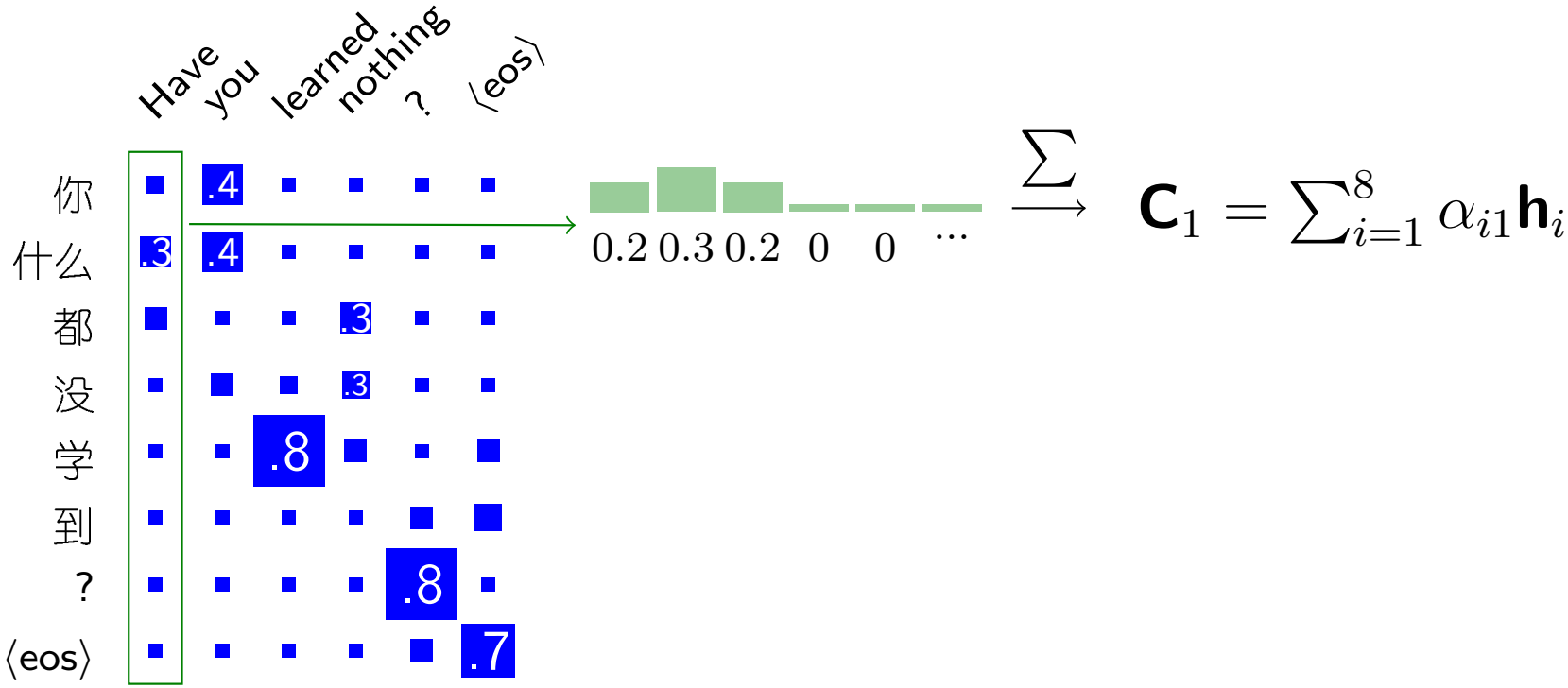
注意力权重α_{ij}

- 注意力权重α_{ij}的可视化



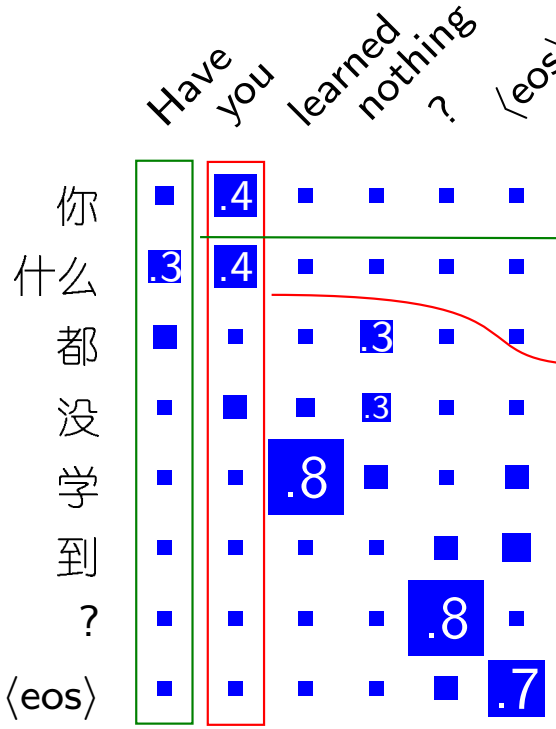
注意力权重α_{ij}

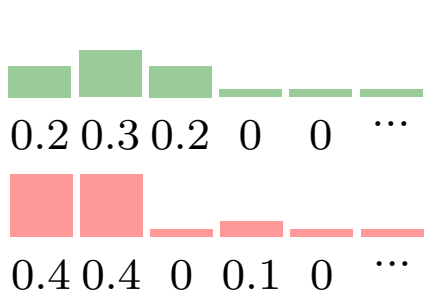
- 注意力权重α_{ij}的可视化



注意力权重α_{ij}

- 注意力权重α_{ij}的可视化





$$\mathbf{C}_1 = \sum_{i=1}^{8} \alpha_{i1}\mathbf{h}_i$$

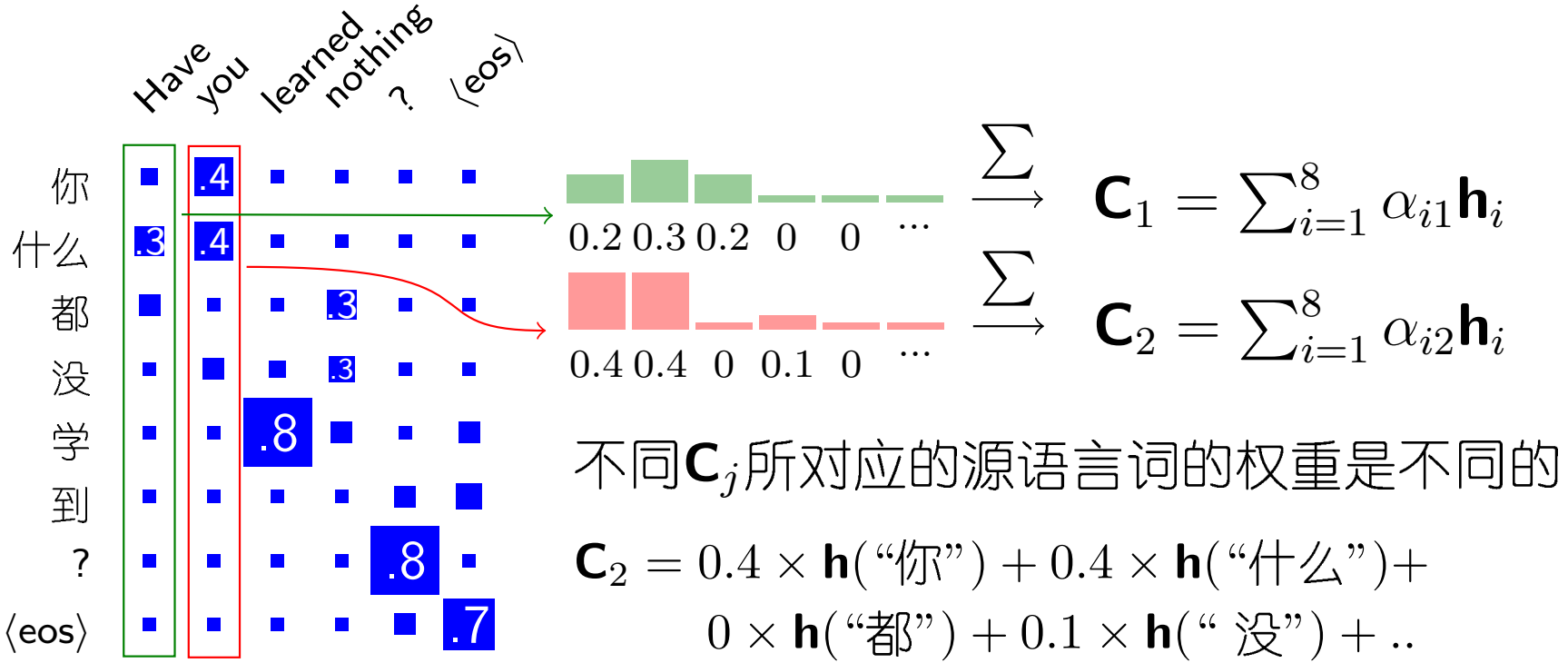
$$\mathbf{C}_2 = \sum_{i=1}^{8} \alpha_{i2}\mathbf{h}_i$$

不同$\mathbf{C}_j$所对应的源语言词的权重是不同的

$$\mathbf{C}_2 = 0.4 \times \mathbf{h}(\text{“你”}) + 0.4 \times \mathbf{h}(\text{“什么”}) + 0 \times \mathbf{h}(\text{“都”}) + 0.1 \times \mathbf{h}(\text{“ 没”}) + ..$$

注意力权重α_{ij}

- 注意力权重α_{ij}的可视化



不同$\mathbf{C}_j$所对应的源语言词的权重是不同的

$$\mathbf{C}_2 = 0.4 \times \mathbf{h}(\text{“你”}) + 0.4 \times \mathbf{h}(\text{“什么”}) + 0 \times \mathbf{h}(\text{“都”}) + 0.1 \times \mathbf{h}(\text{“ 没”}) + ..$$

- 对比

引入注意力机制以前	引入注意力机制以后
“Have” $= \arg\max_{y_1} \mathrm{P}(y_1 \mid \mathbf{C}, y_0)$	“Have” $= \arg\max_{y_1} \mathrm{P}(y_1 \mid 0, \mathbf{C}_1, y_0)$
“you” $= \arg\max_{y_2} \mathrm{P}(y_2 \mid \mathbf{s}_1, y_1)$	“you” $= \arg\max_{y_2} \mathrm{P}(y_2 \mid \mathbf{s}_1, \mathbf{C}_2, y_1)$

计算注意力权重 - 注意力函数

- 再来看一下注意力权重的定义。这个过程实际上是对$a(\cdot,\cdot)$做指数归一化：

$$\alpha_{i,j} = \frac{\exp(a(\mathbf{s}_{j-1}, \mathbf{h}_i))}{\sum_{i'} \exp(a(\mathbf{s}_{j-1}, \mathbf{h}_{i'}))}$$

计算注意力权重 - 注意力函数

- 再来看一下注意力权重的定义。这个过程实际上是对$a(\cdot,\cdot)$做指数归一化：

$$\alpha_{i,j} = \frac{\exp(a(\mathbf{s}_{j-1}, \mathbf{h}_i))}{\sum_{i'} \exp(a(\mathbf{s}_{j-1}, \mathbf{h}_{i'}))}$$

- 注意力函数$a(\mathbf{s}, \mathbf{h})$的目的是捕捉**s**和**h**之间的相似性，这也可以被看作是目标语表示和源语言表示的一种“统一化”，即把源语言和目标语表示在同一个语义空间，进而语义相近的内容有更大的相似性。

计算注意力权重 - 注意力函数

- 再来看一下注意力权重的定义。这个过程实际上是对$a(\cdot,\cdot)$做指数归一化：

$$\alpha_{i,j} = \frac{\exp(a(\mathbf{s}_{j-1}, \mathbf{h}_i))}{\sum_{i'} \exp(a(\mathbf{s}_{j-1}, \mathbf{h}_{i'}))}$$

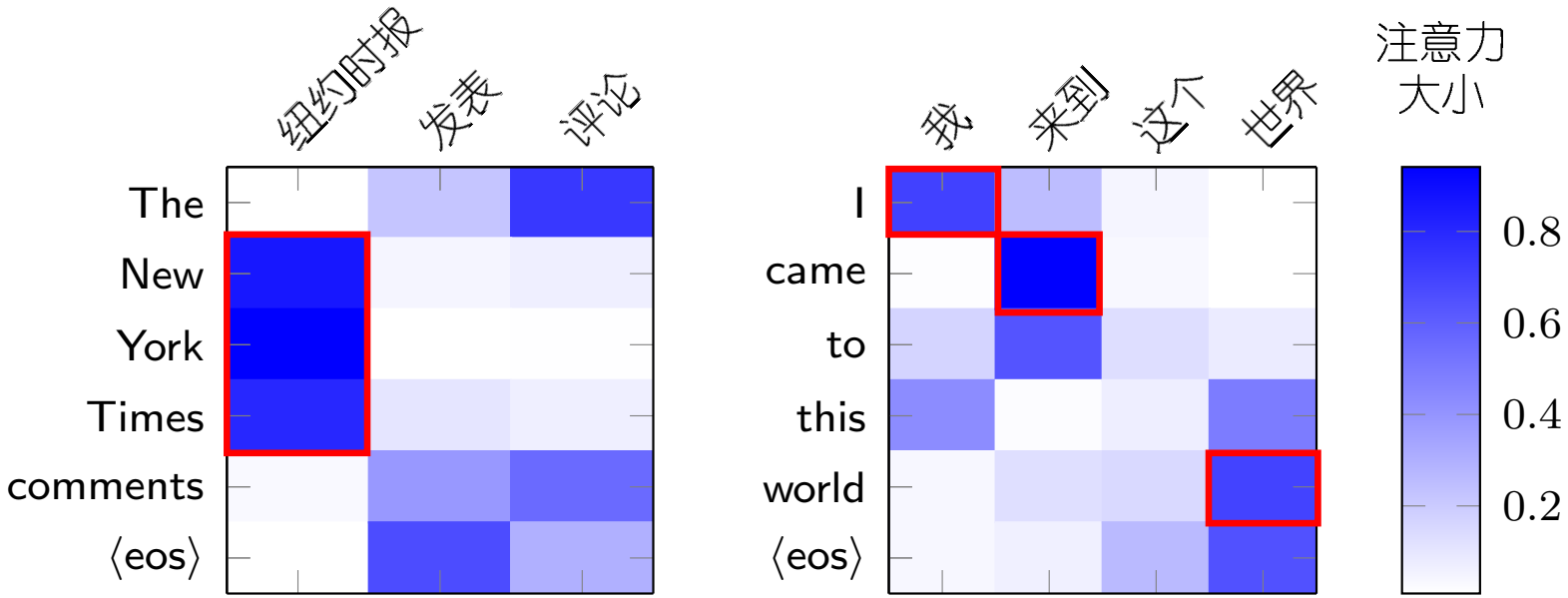
- 注意力函数$a(\mathbf{s}, \mathbf{h})$的目的是捕捉$\mathbf{s}$和$\mathbf{h}$之间的相似性，这也可以被看作是目标语表示和源语言表示的一种“统一化”，即把源语言和目标语表示在同一个语义空间，进而语义相近的内容有更大的相似性。定义$a(\mathbf{s}, \mathbf{h})$的方式：

$$a(\mathbf{s}, \mathbf{h}) = \begin{cases} \mathbf{s}\mathbf{h}^{\mathrm{T}} & \text{向量乘} \\ \cos(\mathbf{s}, \mathbf{h}) & \text{向量夹角} \\ \mathbf{s}\mathbf{W}\mathbf{h}^{\mathrm{T}} & \text{线性模型} \\ \mathrm{TanH}(\mathbf{W}[\mathbf{s}, \mathbf{h}])\mathbf{v}^{\mathrm{T}} & \text{拼接}[\mathbf{s}, \mathbf{h}] + \text{单层网络} \end{cases}$$

$\mathbf{W}$和$\mathbf{v}$是可学习参数

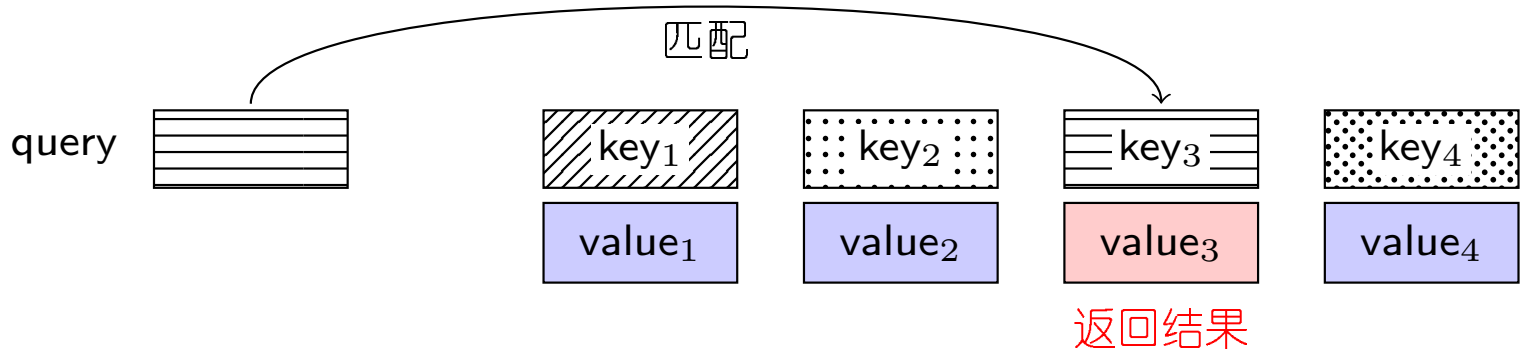
真实的实例

- 注意力的权重符合双语对应的规律
  1. 翻译出“New York Times”的时候“纽约时报”的权重很大
  2. 翻译出“I”的时候“我”的权重很大
  3. 翻译出“came”的时候“来到”的权重很大
  4. 翻译出“world”的时候“世界”的权重很大
- 互译的词通常都会产生较大的注意力权重
- 注意力的权重一定程度上反应了词语间的对应关系



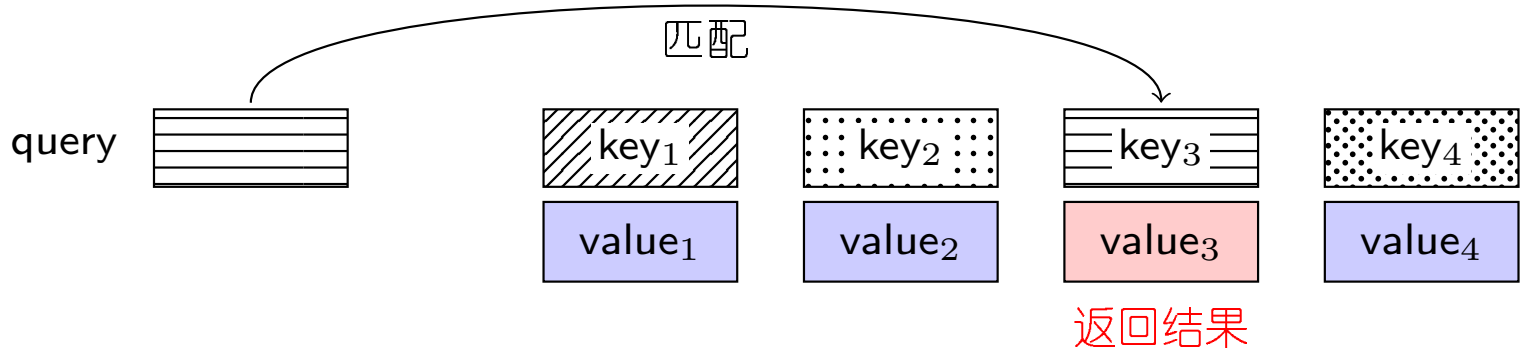
重新解释注意力机制

- 换一个问题，假设有若干key-value单元，其中key是这个单元的索引表示，value是这个单元的值。对于任意一个query，可以找到匹配的key，并输出其对应的value

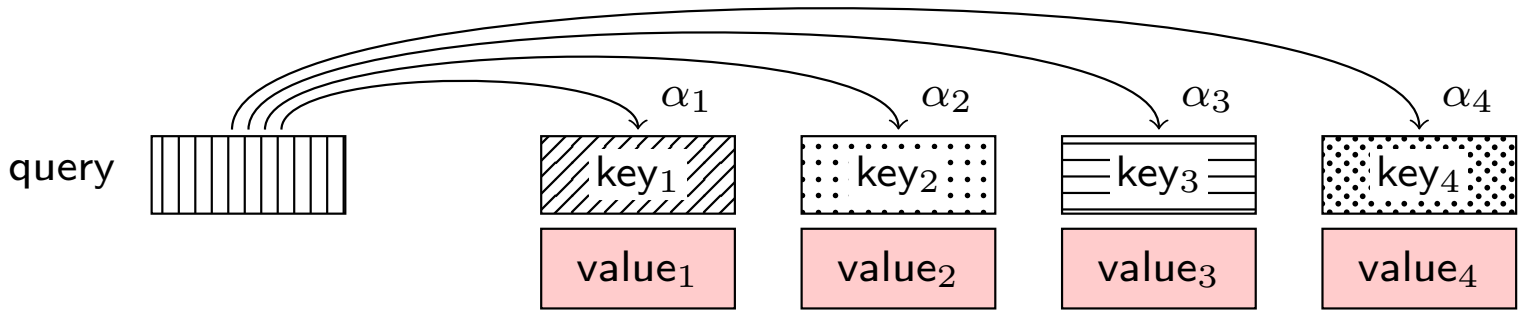


重新解释注意力机制

- 换一个问题，假设有若干key-value单元，其中key是这个单元的索引表示，value是这个单元的值。对于任意一个query，可以找到匹配的key，并输出其对应的value



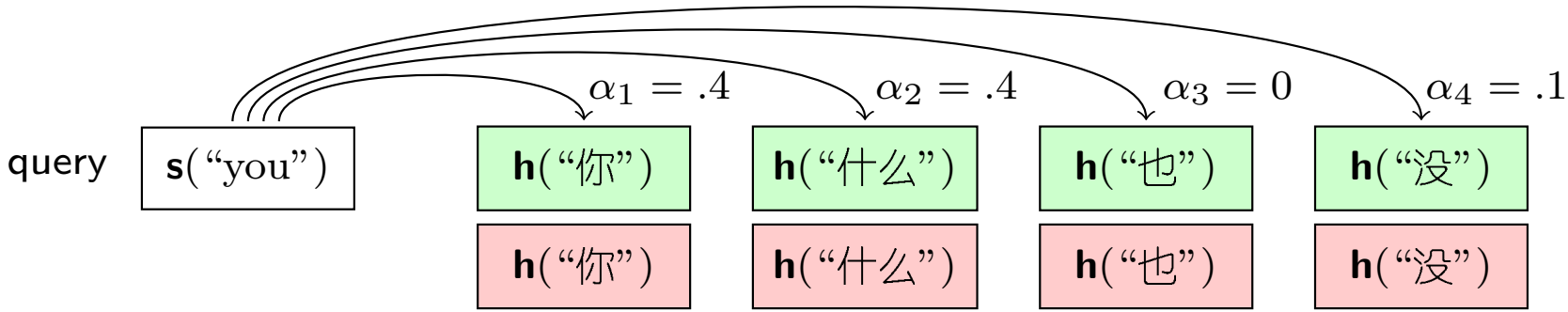
- 注意力机制也可以被看做对key-value单元的查询，但是所有key和query之间都有一种匹配程度，返回结果是对所有value的加权



返回结果$=\alpha_1 \cdot \text{value}_1 + \alpha_2 \cdot \text{value}_2 + \alpha_3 \cdot \text{value}_3 + \alpha_4 \cdot \text{value}_4$

重新解释注意力机制(续)

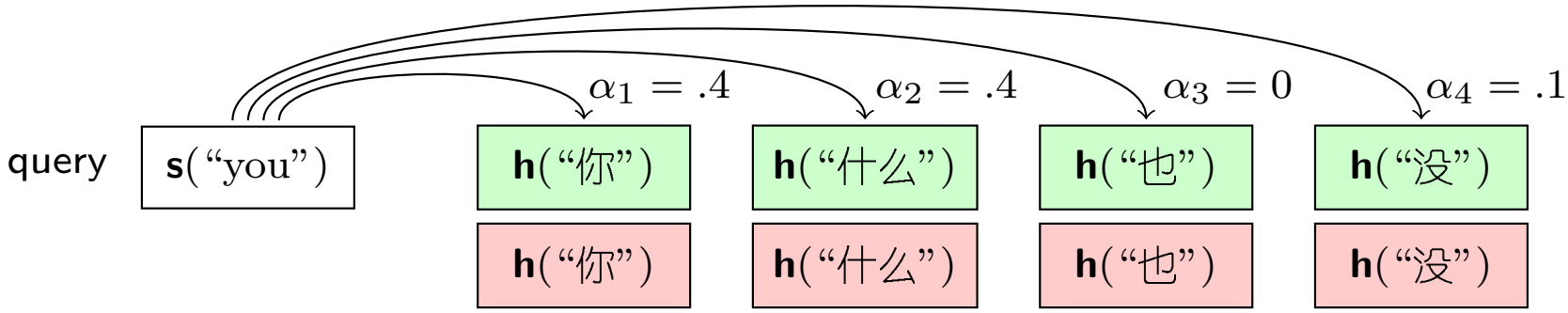
- 回到机器翻译，如果把目标语状态$\mathbf{s}_{j-1}$看做query，而把源语言所有位置的最上层RNN表示$\mathbf{h}_i$看做**key**和**value**



$$\mathbf{C}_3 = 0.4 \times \mathbf{h}(\text{“你”}) + 0.4 \times \mathbf{h}(\text{“什么”}) + 0 \times \mathbf{h}(\text{“也”}) + 0.1 \times \mathbf{h}(\text{“没”})$$

重新解释注意力机制(续)

- 回到机器翻译，如果把目标语状态$\mathbf{s}_{j-1}$看做query，而把源语言所有位置的最上层RNN表示$\mathbf{h}_i$看做**key**和**value**

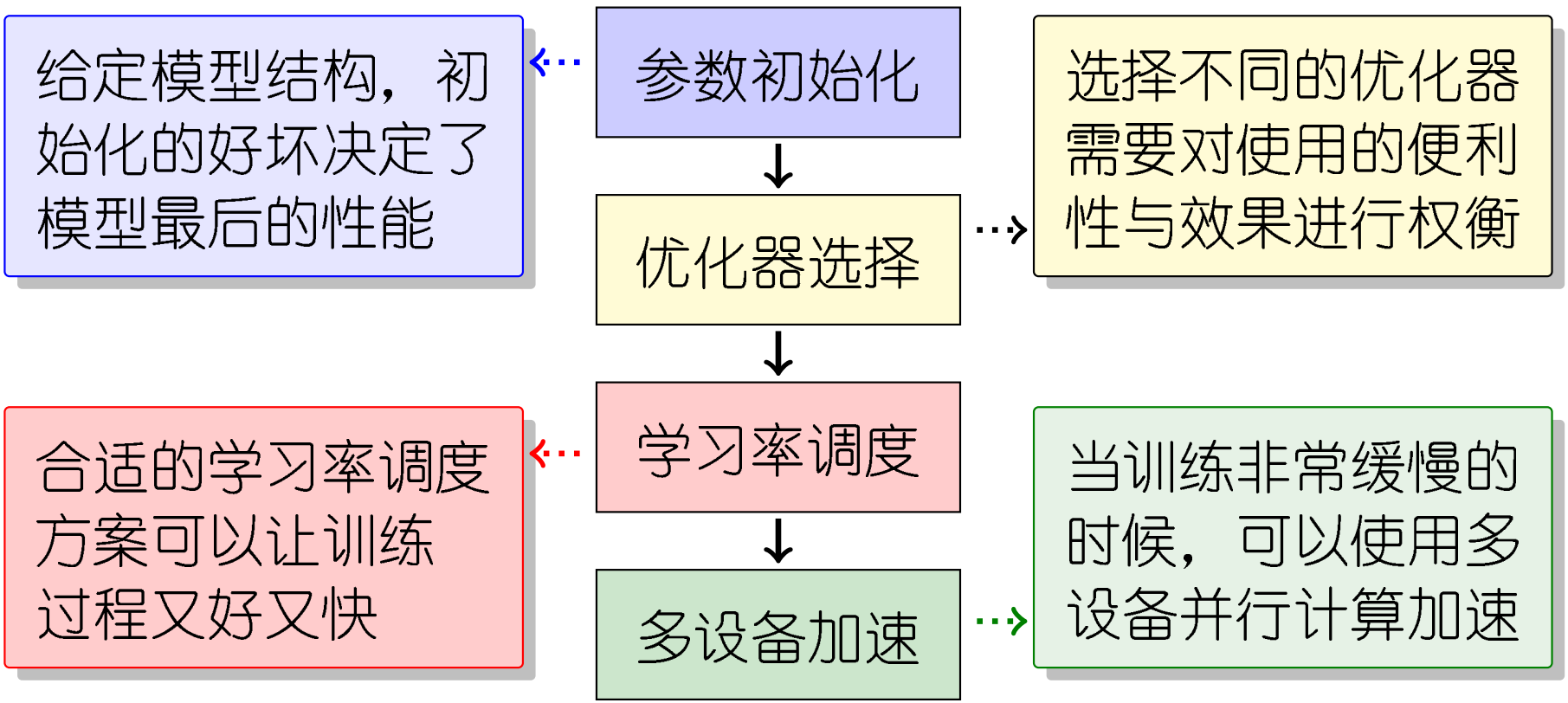


$$\mathbf{C}_3 = 0.4 \times \mathbf{h}(\text{“你”}) + 0.4 \times \mathbf{h}(\text{“什么”}) + 0 \times \mathbf{h}(\text{“也”}) + 0.1 \times \mathbf{h}(\text{“没”})$$

- 注意力机制也可以被看做是一个重新生成value的过程：对于一组value值，注意力模型对他们加权求和，并得到一个新的value。而这个新的value实际上就是query所对应查询结果，在机器翻译中被看做是目标语所对应的源语言上下文表示。

训练 - 整体流程

- 有了一个NMT模型，我们应该怎么使用梯度下降算法来训练一个翻译模型呢？或者说哪些因素会对RNN训练产生影响？



训练 - 初始化

- 模型结构是确定了，但是我们初始化参数还有很多需要注意的地方，否则训练不了一个优秀的模型
  - LSTM遗忘门偏置初始为1，也就是始终选择遗忘记忆c，可以有效防止初始时c里包含的错误信号传播后面所有时刻
  - 网络的其他偏置一般都初始化成0，可以有效防止加入过大或过小的偏置后使得激活函数的输出跑到“饱和区”，也就是梯度接近0的区域，使得训练一开始就无法跳出局部极小

训练 - 初始化

- 模型结构是确定了，但是我们初始化参数还有很多需要注意的地方，否则训练不了一个优秀的模型
  - LSTM遗忘门偏置初始为1，也就是始终选择遗忘记忆c，可以有效防止初始时c里包含的错误信号传播后面所有时刻
  - 网络的其他偏置一般都初始化成0，可以有效防止加入过大或过小的偏置后使得激活函数的输出跑到“饱和区”，也就是梯度接近0的区域，使得训练一开始就无法跳出局部极小
  - 网络的权重矩阵W一般使用Xavier参数初始化方法，可以有效稳定训练过程，特别是对于比较“深”的网络

$$W \sim \mathcal{U}(-\sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}})$$

d_{in}和d_{out}分别是W的输入和输出的维度大小，参考论文
Understanding the difficulty of training deep feedforward neural networks
Glorot, X., & Bengio, Y., 2010, In Proc of AISTATS

训练 - 优化器

- 训练RNN我们通常会使用Adam或者SGD两种优化器，它们各有优劣

	使用	性能
Adam	一套配置包打天下	不算差，但没到极限
SGD	换一个任务就得调	效果杠杠的

- 因此需要快速得到模型看一下初步效果，选择Adam

训练 - 优化器

- 训练RNN我们通常会使用Adam或者SGD两种优化器，它们各有优劣

	使用	性能
Adam	一套配置包打天下	不算差，但没到极限
SGD	换一个任务就得调	效果杠杠的

- 因此需要快速得到模型看一下初步效果，选择Adam
- 若是需要在一个任务上得到最优的结果，选择SGD
  - 需要注意的是，训练RNN的时候，我们通常会遇到梯度爆炸的问题，也就是梯度突然变得很大，这种情况下需要使用“梯度裁剪”来防止梯度π超过阈值

$$\pi' = \pi \cdot \frac{\text{threshold}}{\max(\text{threshold}, \| \pi \|_2)}$$

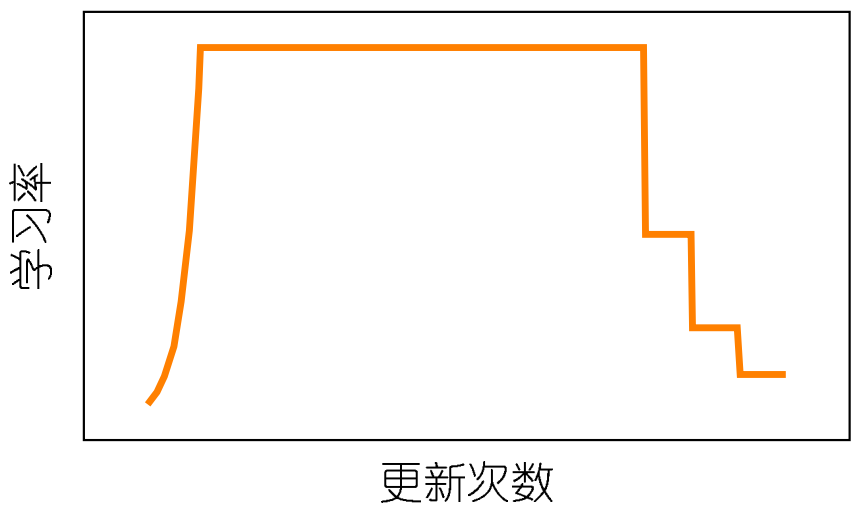
  - 其中threshold是手工设定的梯度大小阈值，$\| \cdot \|_2$是L2范数
  - 这个公式含义在于只要梯度大小超过阈值，就按照阈值与当前梯度大小的比例进行放缩

训练 - 学习率

- 不同优化器需要的学习率不同，比如Adam一般使用0.001或0.0001，而SGD则在$0.1 \sim 1$之间挑选
- 但是无论使用哪个优化器，为了保证训练又快又好，我们通常都需要根据当前的更新次数来调整学习率的大小

训练 - 学习率

- 不同优化器需要的学习率不同，比如Adam一般使用0.001或0.0001，而SGD则在0.1 ~ 1之间挑选
- 但是无论使用哪个优化器，为了保证训练又快又好，我们通常都需要根据当前的更新次数来调整学习率的大小
  - 学习率预热：模型训练初期，梯度通常很大，直接使用很大的学习率很容易让模型跑偏，因此需要学习率有一个从小到大的过程
  - 学习率衰减：模型训练接近收敛的时候，使用大学习率会很容易让模型错过局部极小，因此需要学习率逐渐变小来逼近局部最小



训练 - 加速

- 万事俱备，只是为什么训练这么慢？
- 我有钱，是不是多买几台设备会更快？

训练 - 加速

- 万事俱备，只是为什么训练这么慢？- RNN需要等前面所有时刻都完成计算以后才能开始计算当前时刻的输出
- 我有钱，是不是多买几台设备会更快？- 可以，但是需要技巧，而且也不是无限增长的

训练 - 加速

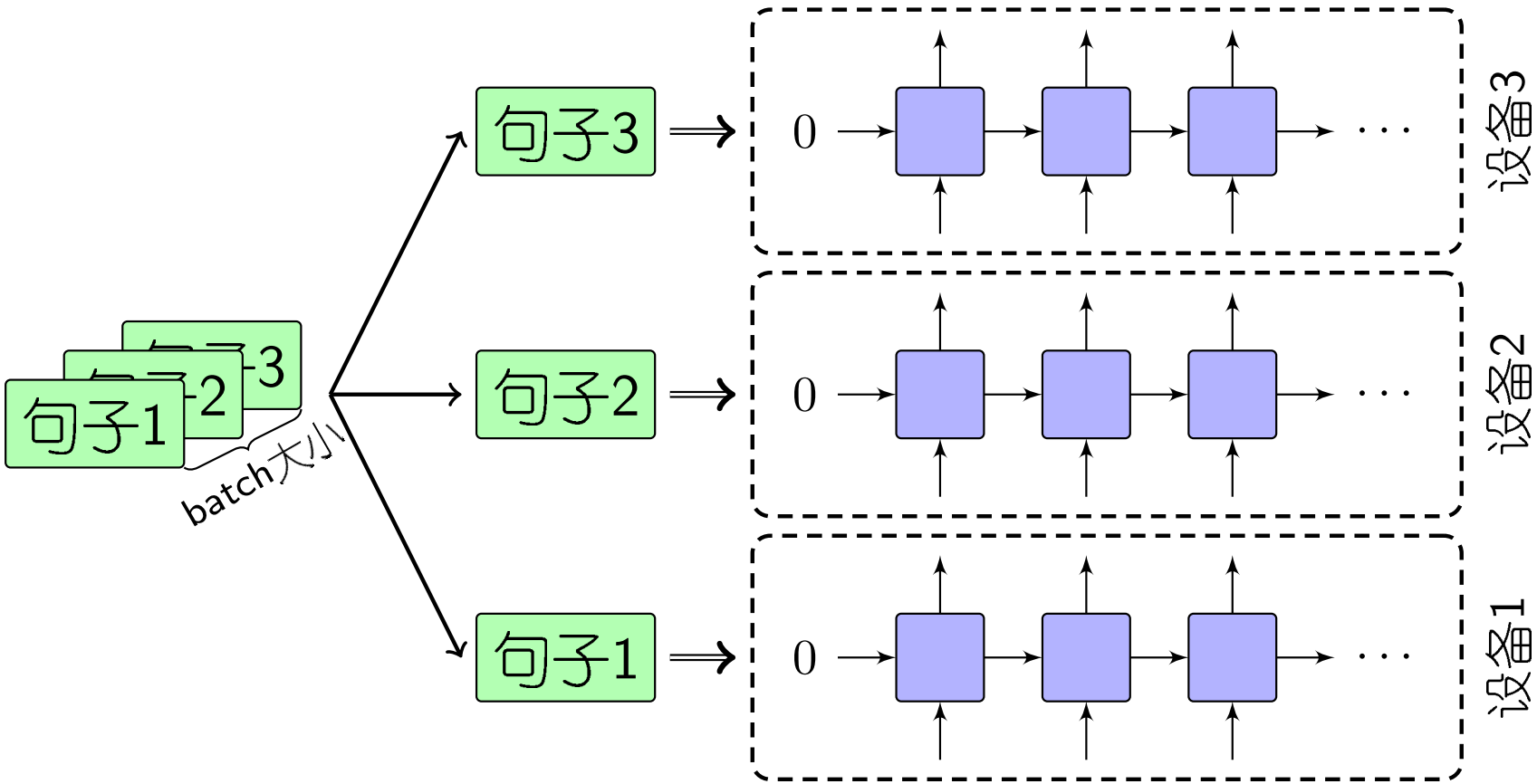
- 万事俱备，只是为什么训练这么慢？- RNN需要等前面所有时刻都完成计算以后才能开始计算当前时刻的输出
- 我有钱，是不是多买几台设备会更快？- 可以，但是需要技巧，而且也不是无限增长的
- 使用多个设备并行计算进行加速的两种方法
  - 数据并行：把“输入”分到不同设备上并行计算
  - 模型并行：把“模型”分到不同设备上并行计算

	优点	缺点
数据并行	并行度高，理论上多大的batch就可以有多少个设备并行计算	模型不能大于单个设备的极限
模型并行	可以对很大的模型进行运算	只能有限并行，比如多少层就多少个设备

- 这两种方法可以一起使用！！！

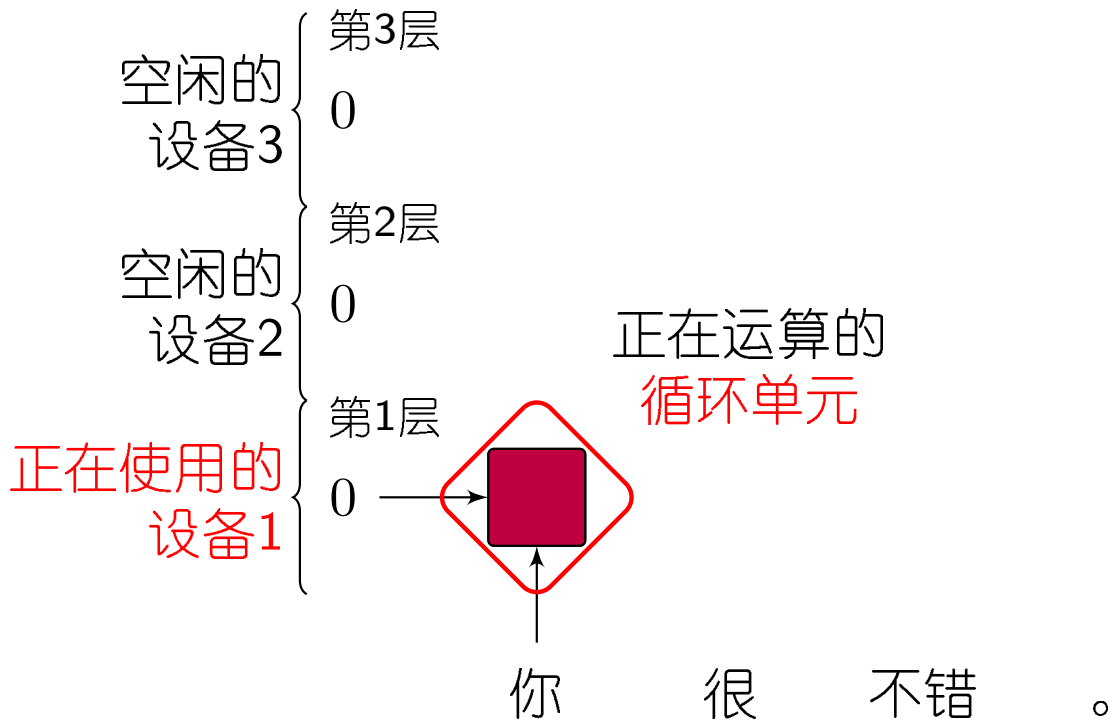
训练 - 数据并行

- 如果一台设备能完整放下一个RNN模型，那么数据并行可以把一个大batch均匀切分成n个小batch，然后分发到n个设备上并行计算，最后把结果汇总，相当于把运算时间变为原来的$1/n$



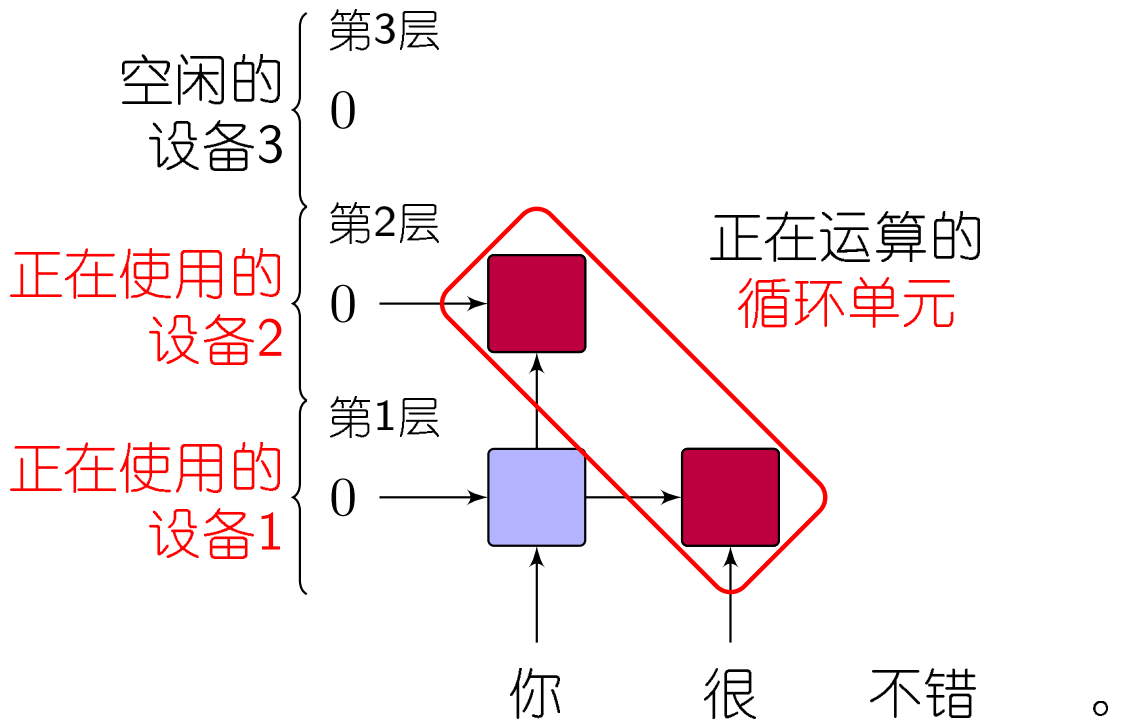
训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



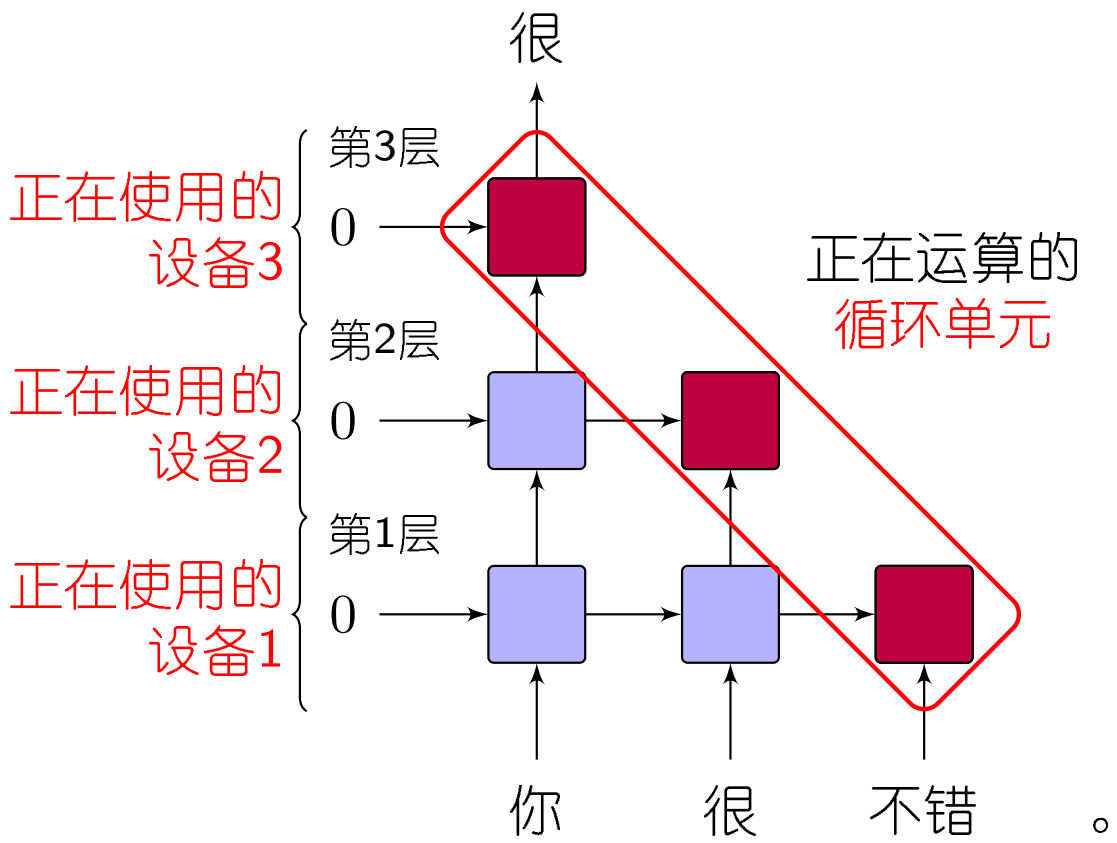
训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



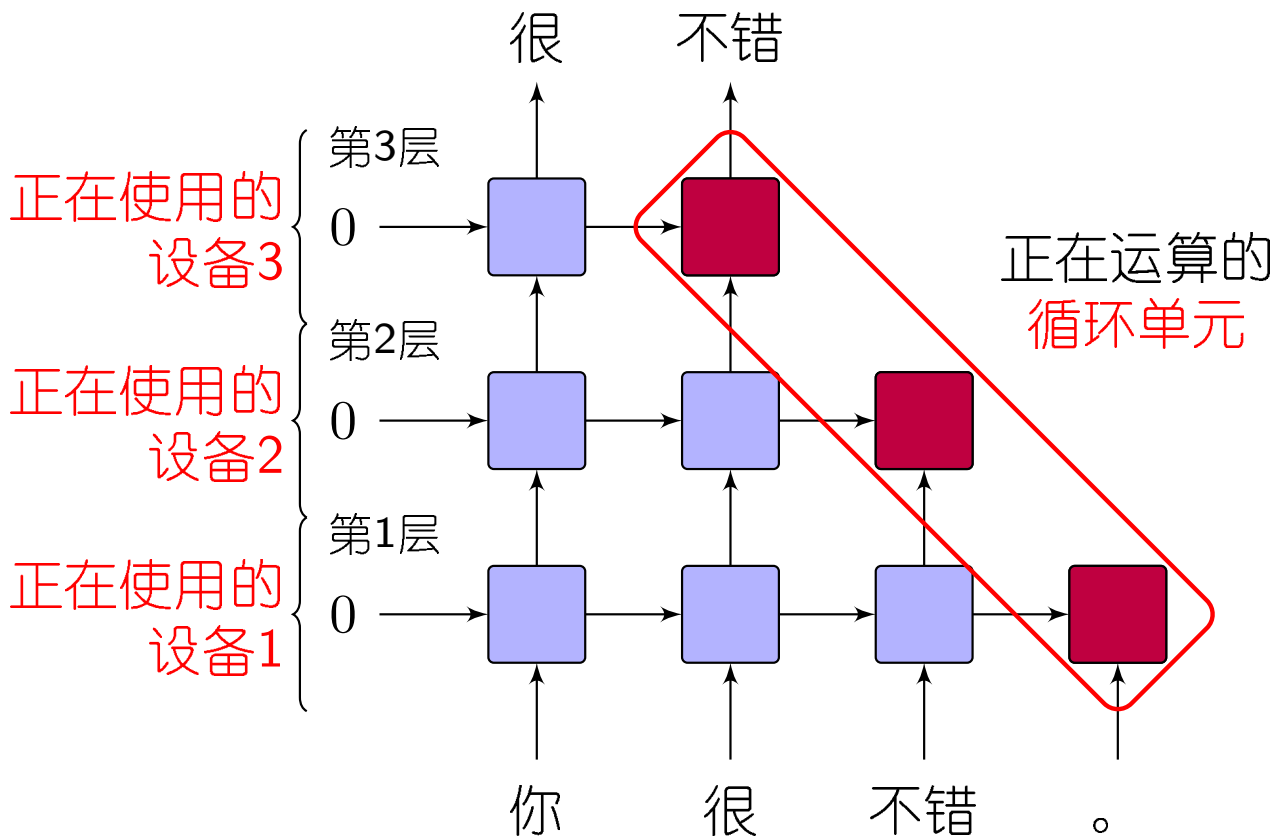
训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



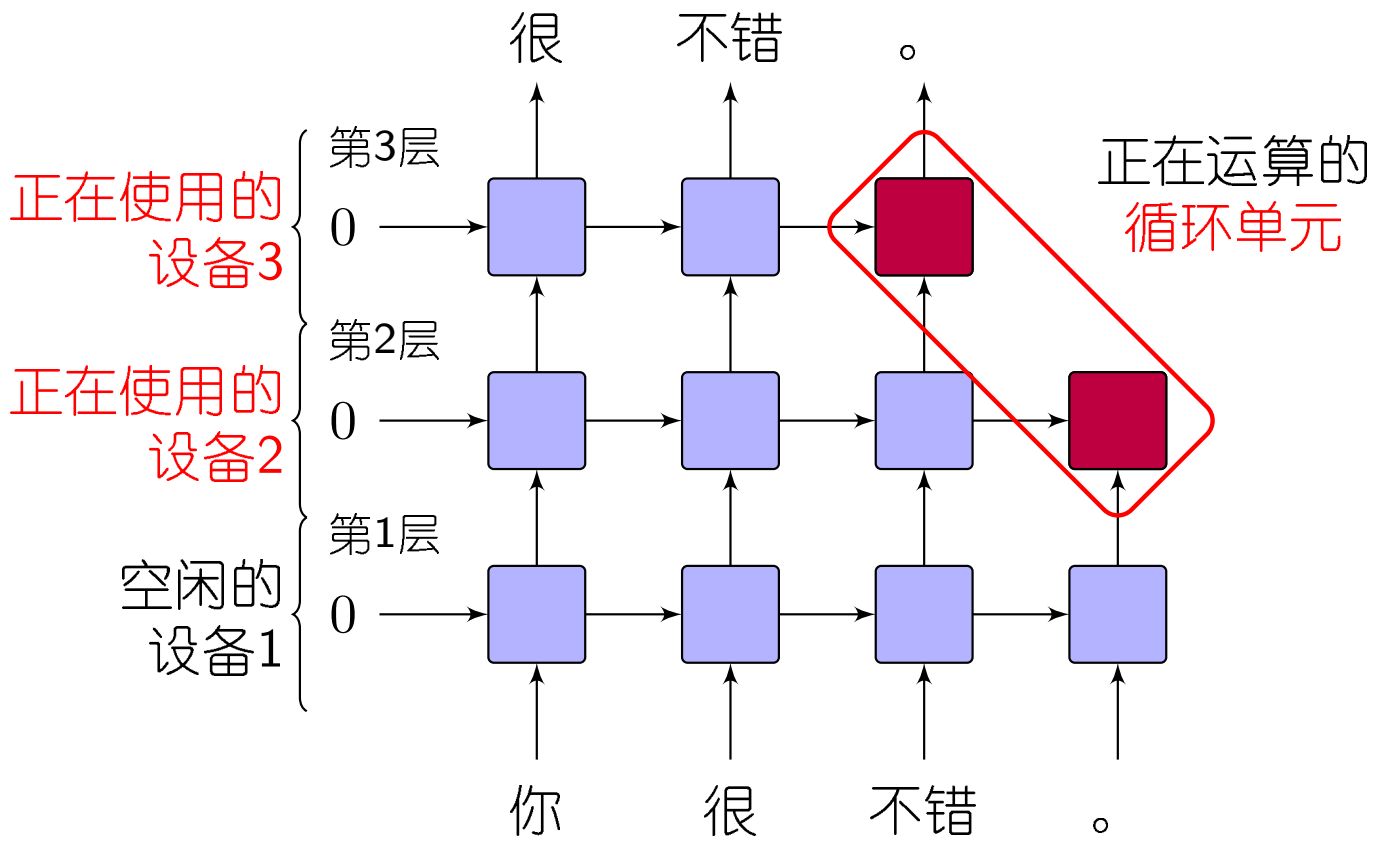
训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



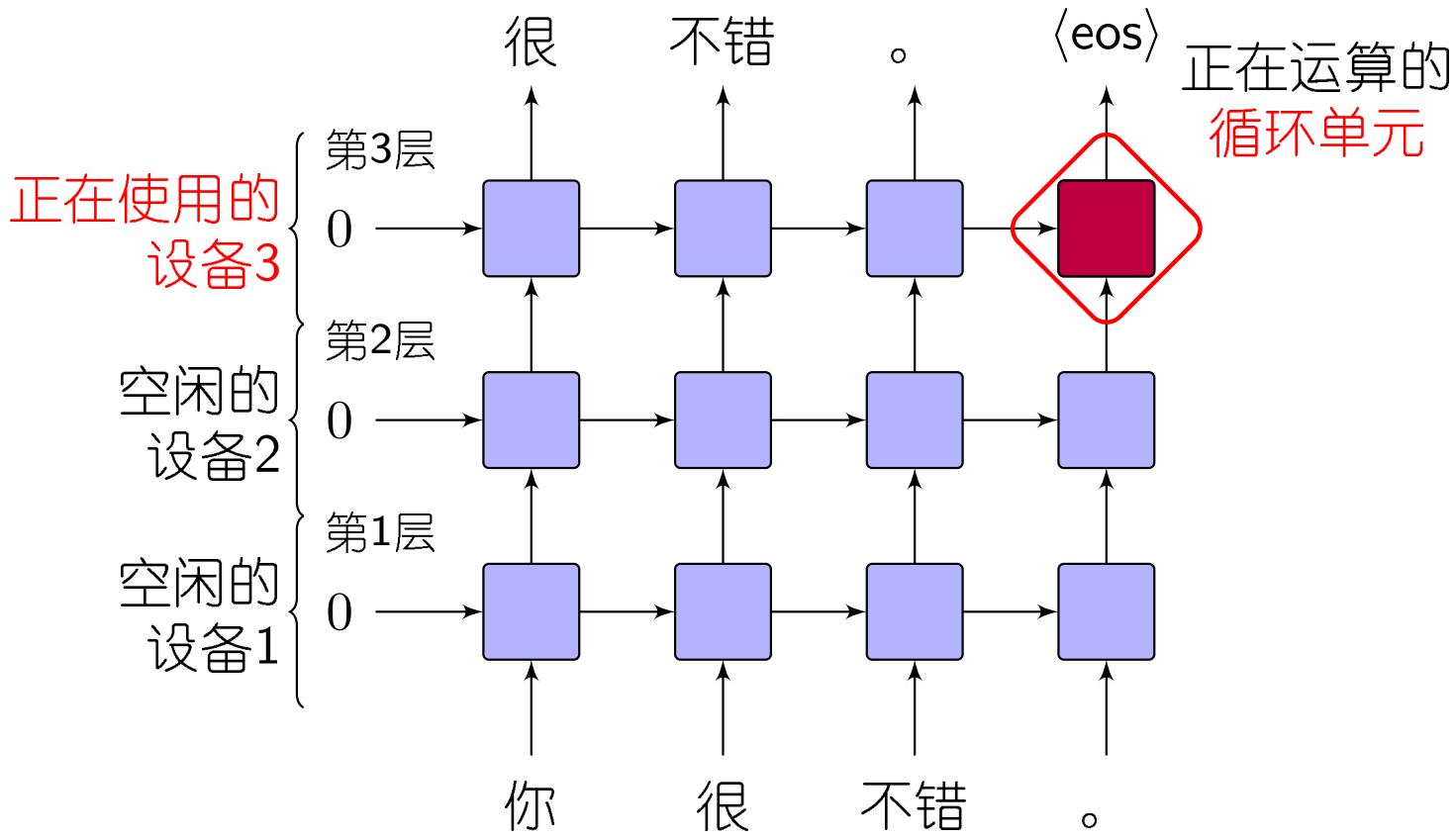
训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



训练 - 模型并行

- 做完了数据并行，仍然太慢了，因为RNN模型太大了，算一个样本也很慢，那么可以把RNN模型按层均匀切分成l个小模型，然后分发到l个设备上并行计算，相当于把运算时间变为原来的$1/l$



推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

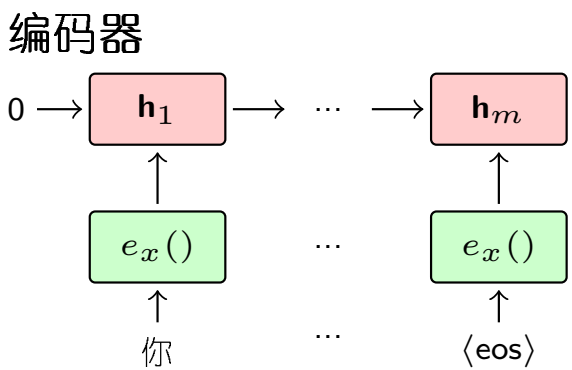
- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, ..., y_n\}$。常用的方法是自左向右逐个单词生成

推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成



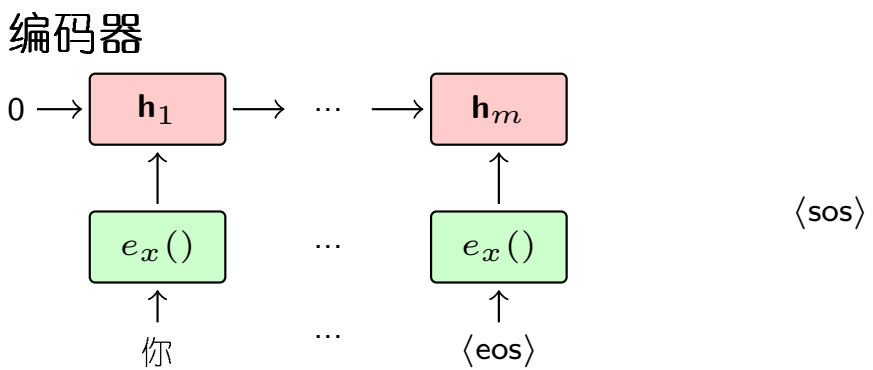
推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, ..., y_n\}$。常用的方法是自左向右逐个单词生成

[step 1]

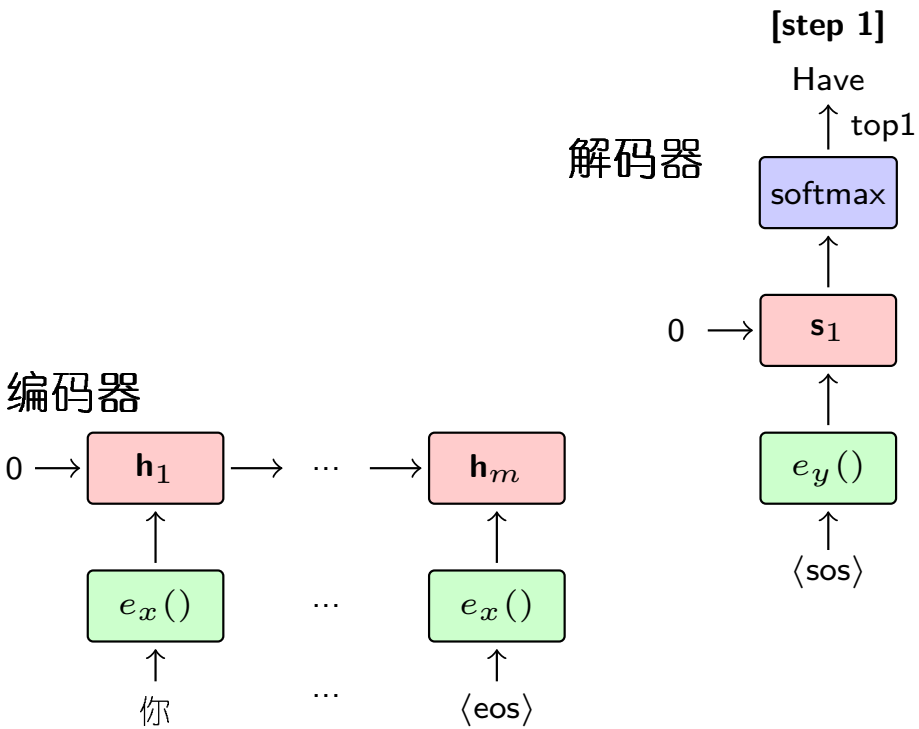


推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成

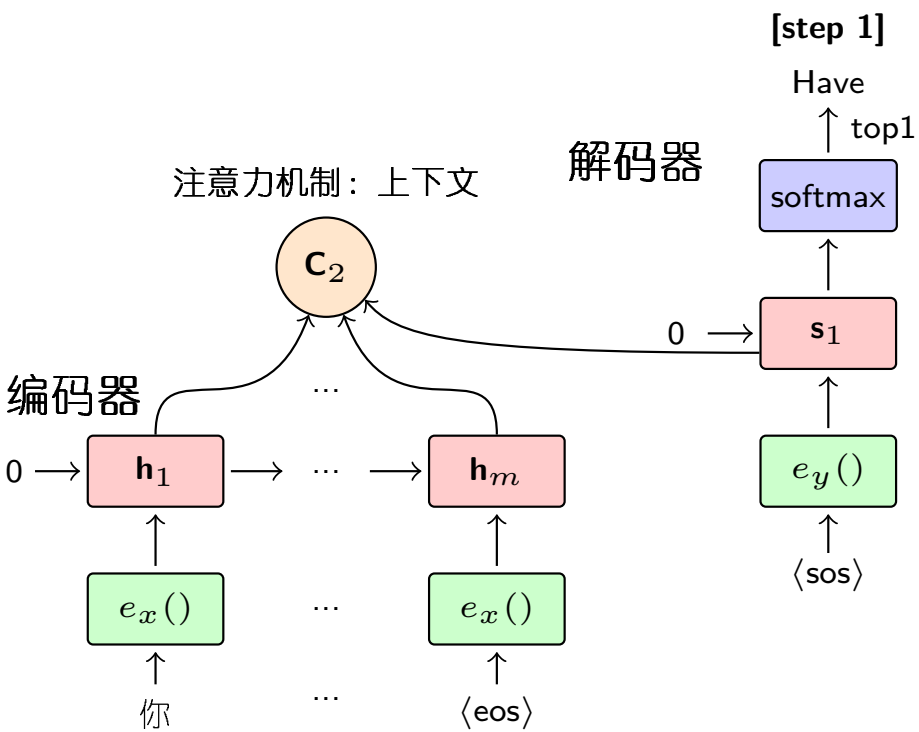


推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成

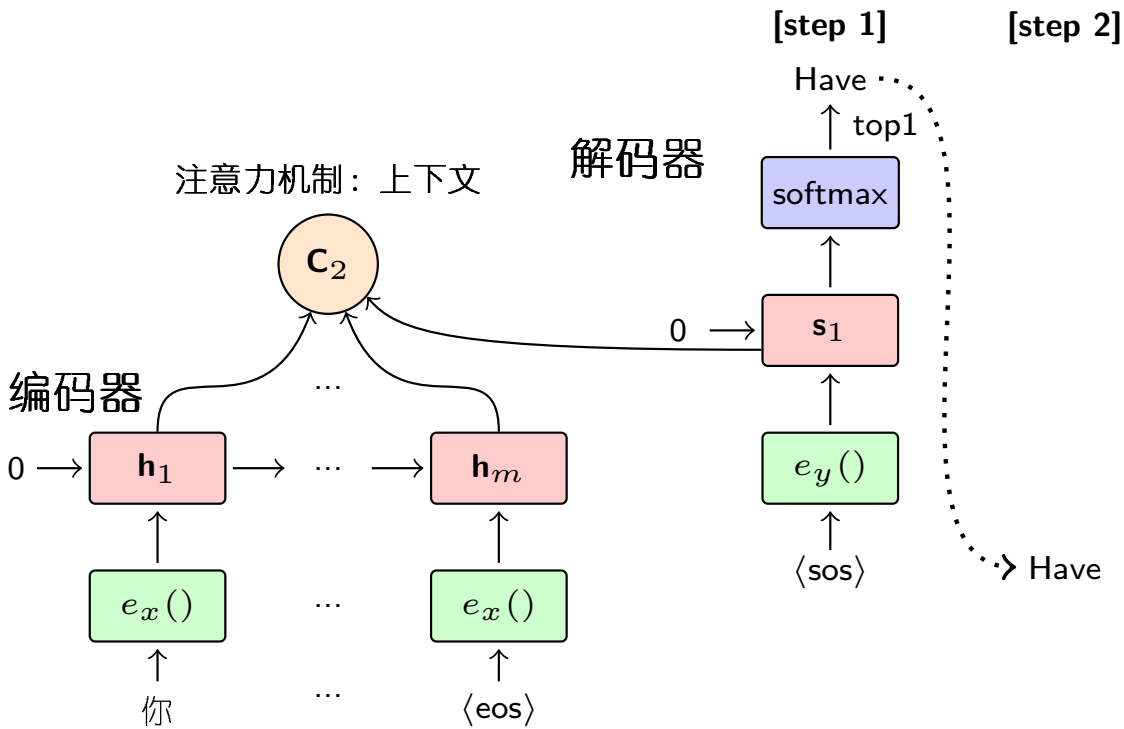


推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成

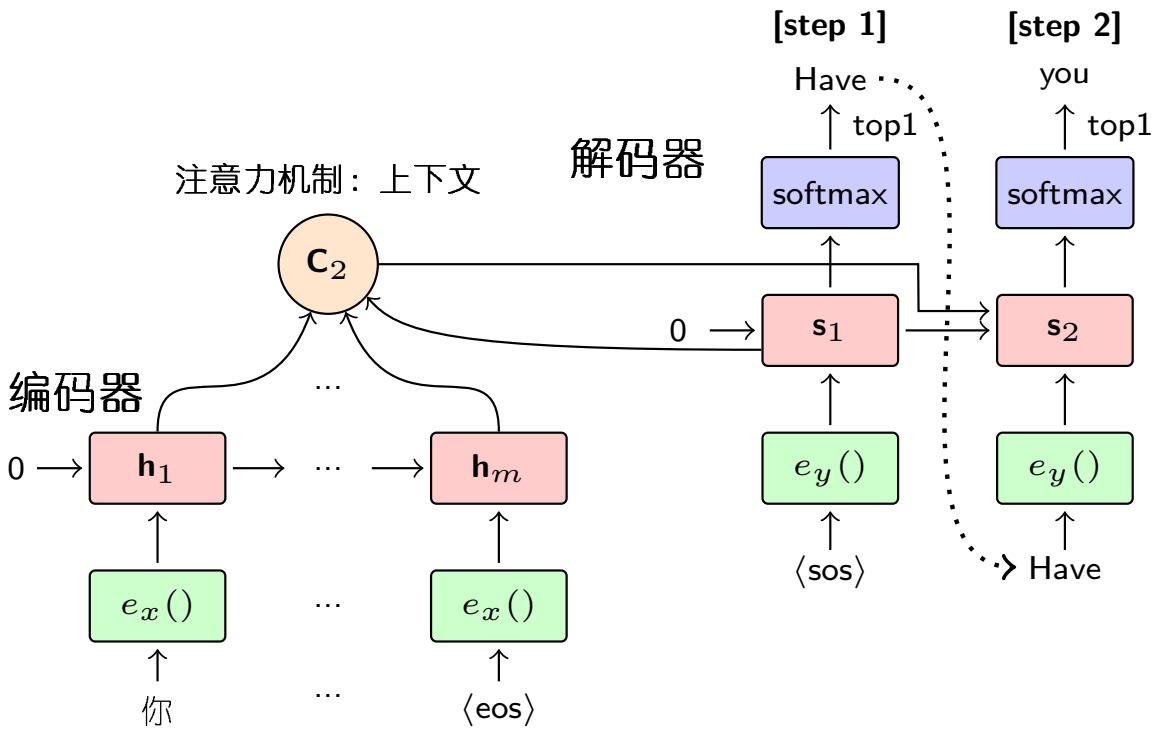


推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成

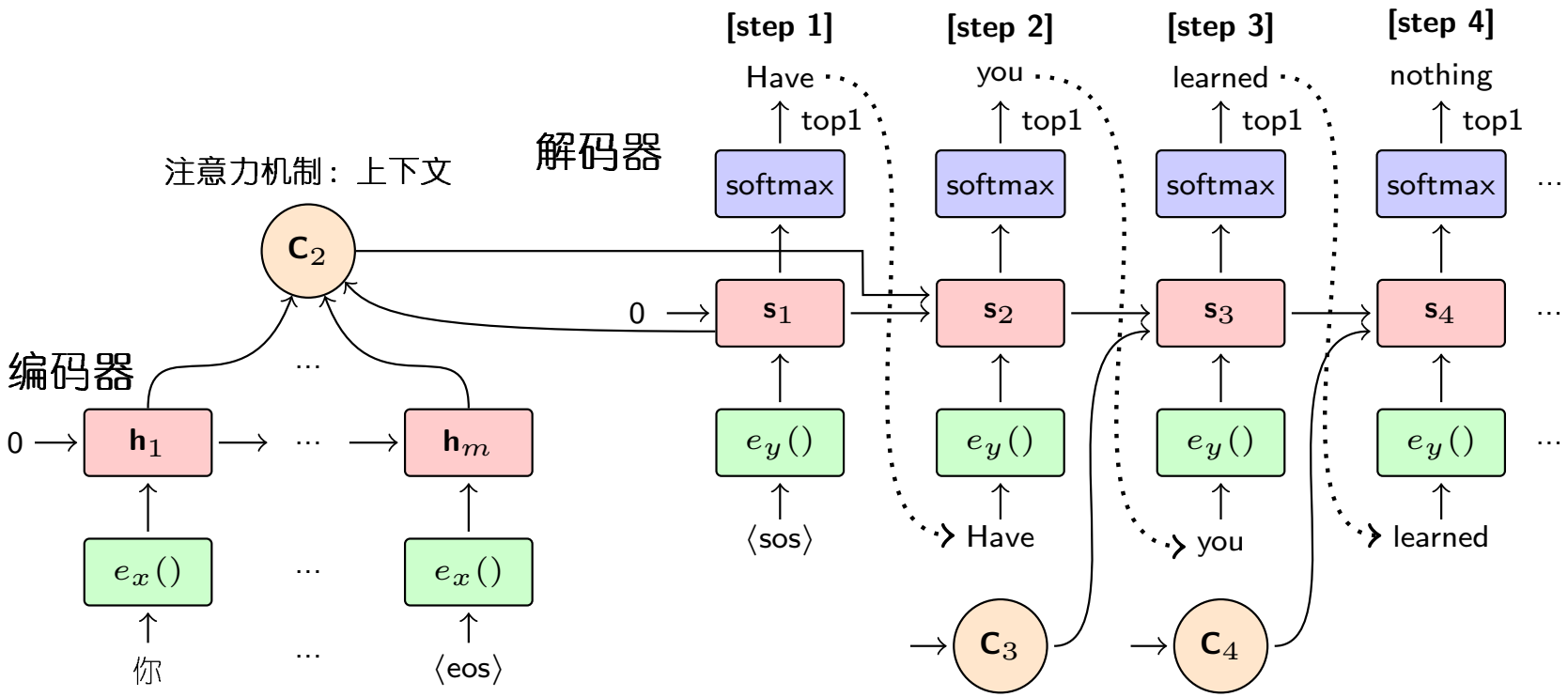


推断

- 使用NMT时，对于源语言句子$\mathbf{x}$，需要得到最优译文$\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \mathrm{P}(\mathbf{y}|\mathbf{x}) = \arg\max_{\mathbf{y}} \sum_{j=1}^{n} \log \mathrm{P}(y_j|\mathbf{y}_{<j}, \mathbf{x})$$

- 由于y_j的生成需要依赖y_{j-1}，因此无法同时生成$\{y_1, \ldots, y_n\}$。常用的方法是自左向右逐个单词生成



推断 - Beam Search

- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文

推断 - Beam Search

- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文

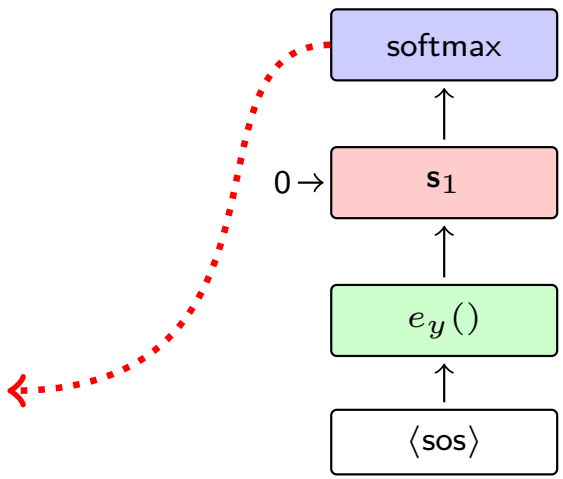
⟨sos⟩

推断 - Beam Search

- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文

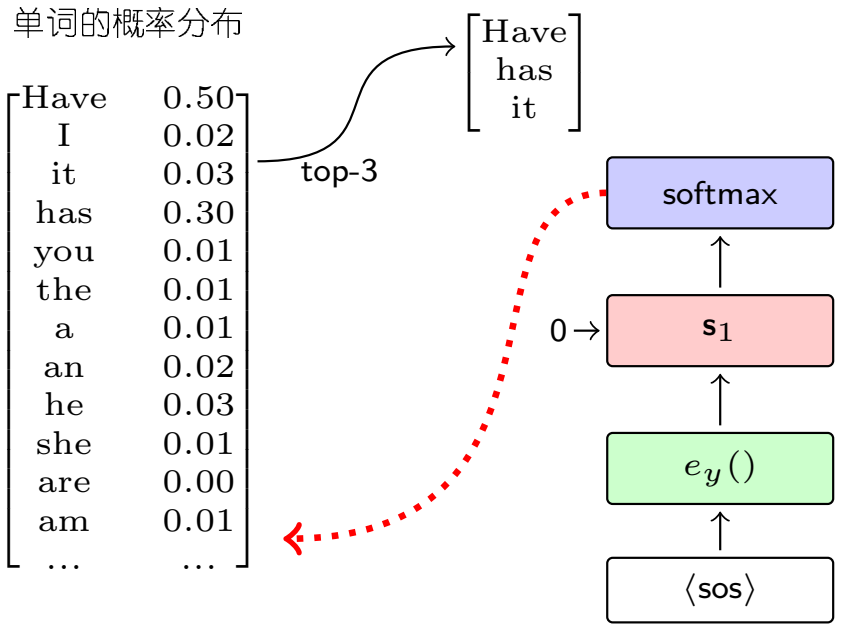
单词的概率分布

Have	0.50
I	0.02
it	0.03
has	0.30
you	0.01
the	0.01
a	0.01
an	0.02
he	0.03
she	0.01
are	0.00
am	0.01
...	...



推断 - Beam Search

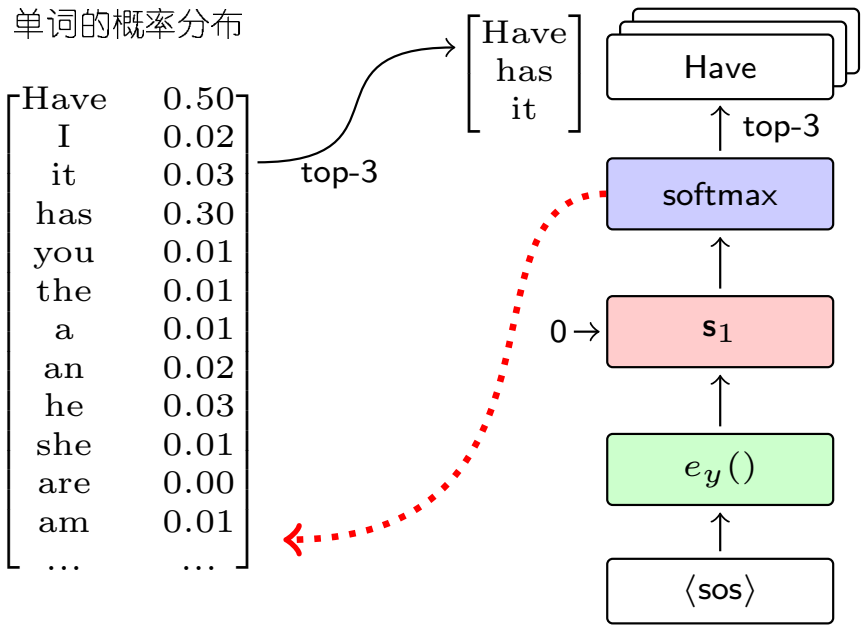
- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文



束搜索($b = 3$)

推断 - Beam Search

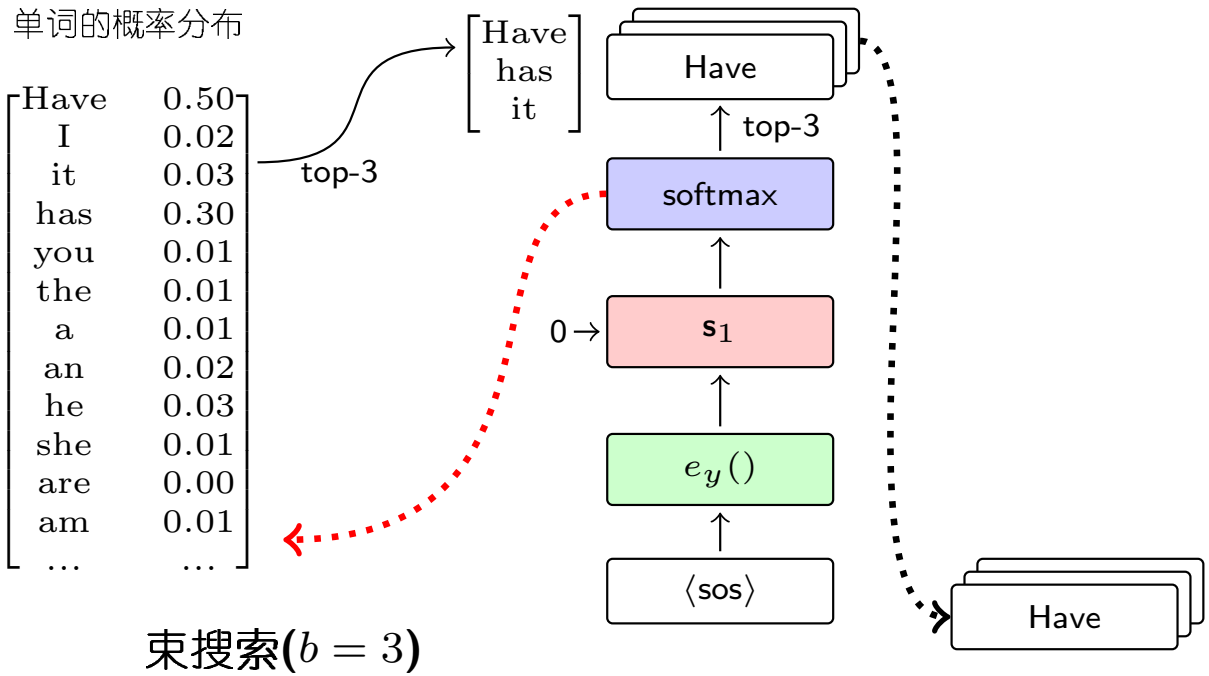
- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文



束搜索($b = 3$)

推断 - Beam Search

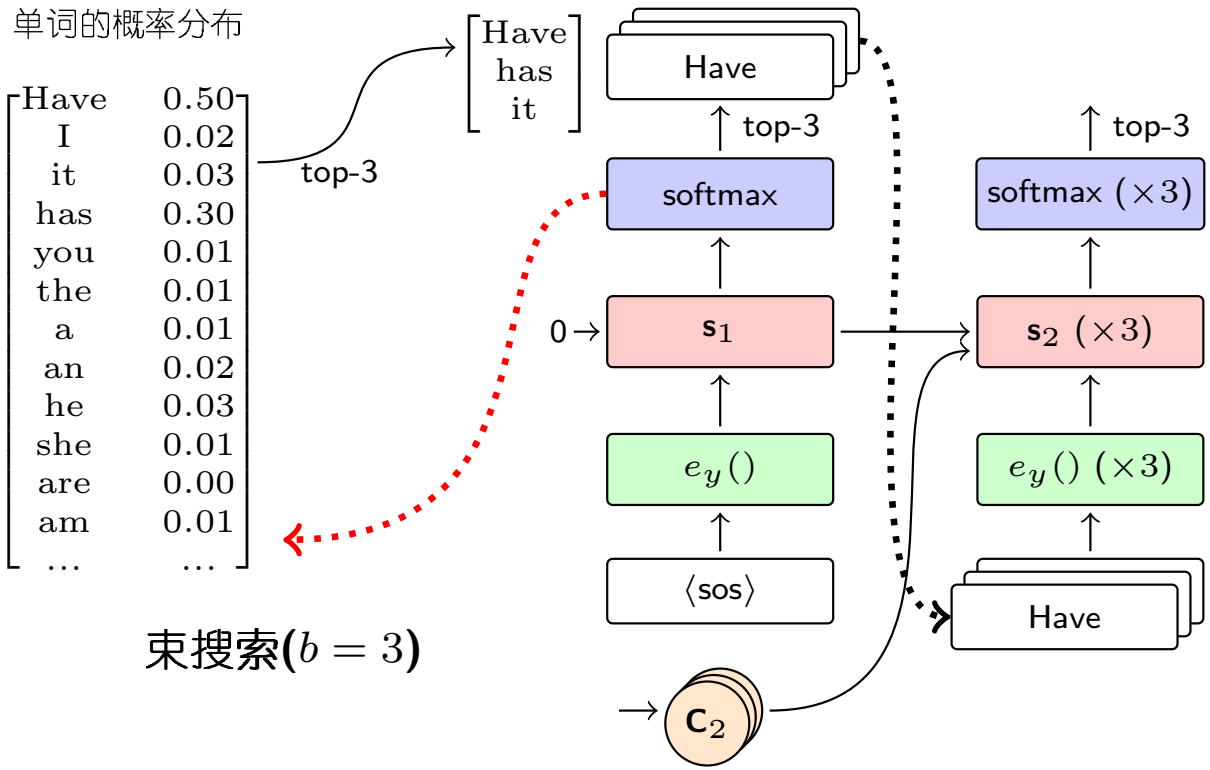
- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文



束搜索($b = 3$)

推断 - Beam Search

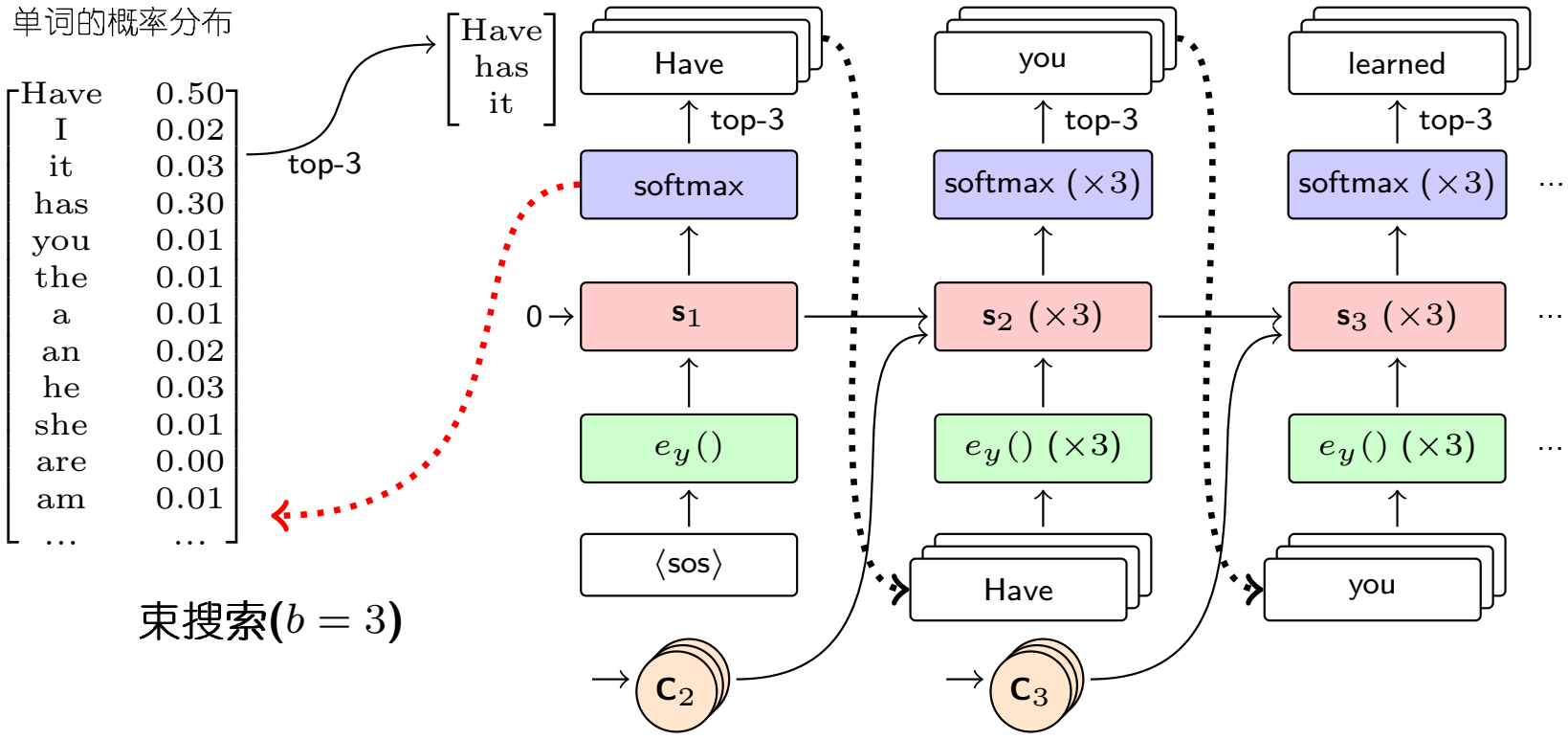
- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文



束搜索($b = 3$)

推断 - Beam Search

- **Greedy Search**: 目标语每一个位置，输出层的Softmax可以得到所有单词的概率，然后选择一个概率最大单词输出，下一个位置的预测就基于这一步输出的单词
- **Beam Search**: 为了避免贪婪方法造成的错误累加，可以每次对b个单词进行扩展，而不是只使用一个单词，其中b称做束的宽度 - 这样可以搜索更多可能的译文



束搜索($b = 3$)

推断 - 其它特征

- 直接用$\mathrm{P}(\mathbf{y}|\mathbf{x})$进行解码，面临两方面问题
  - 对$\mathrm{P}(y_j|\mathbf{y}_{<j},\mathbf{x})$进行乘积会导致长句的概率很低
  - 模型本身并没有考虑每个源语言单词被使用的程度，比如一个单词可能会被翻译了很多“次”

推断 - 其它特征

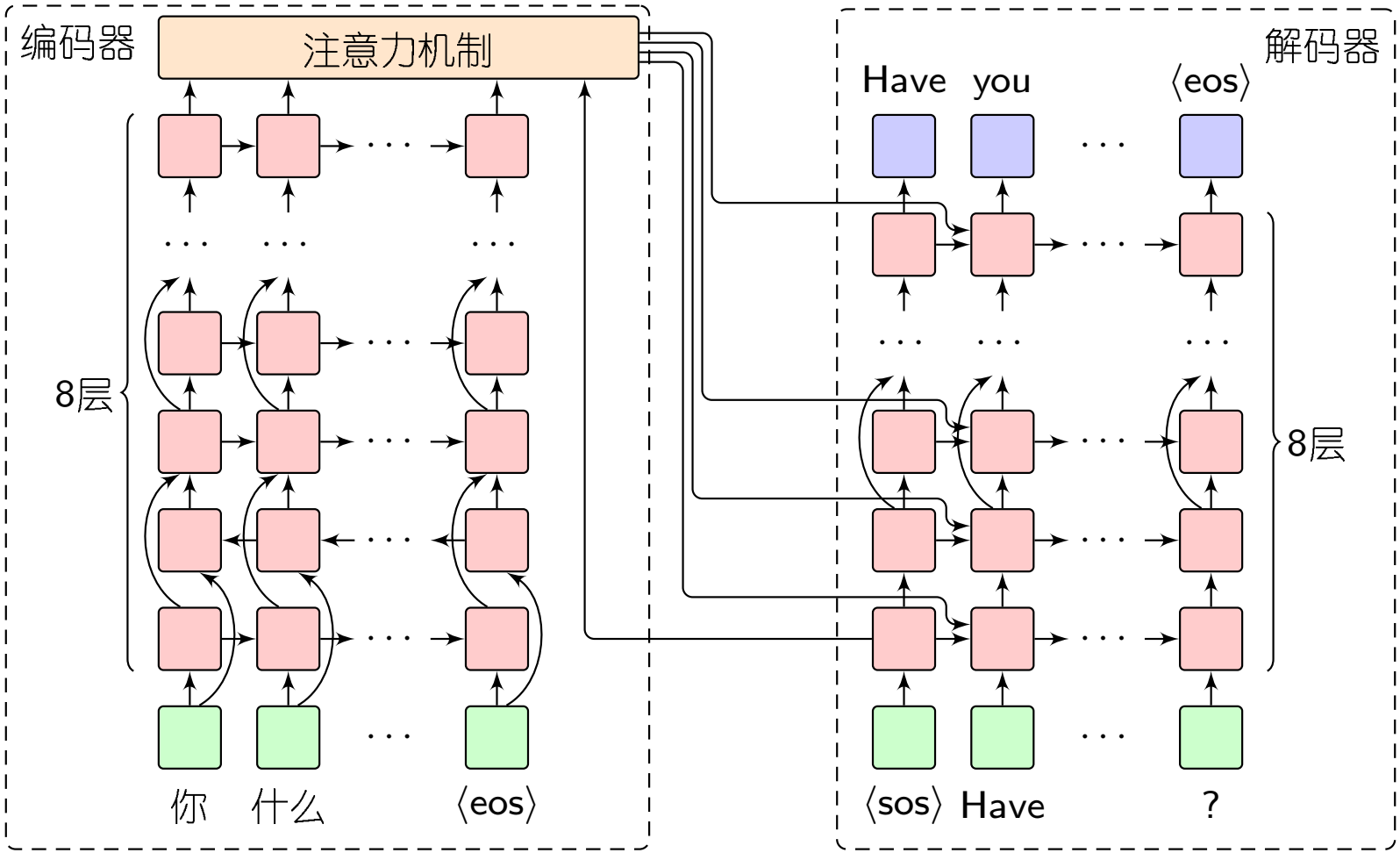
- 直接用$\mathrm{P}(\mathbf{y}|\mathbf{x})$进行解码，面临两方面问题
  - 对$\mathrm{P}(y_j|\mathbf{y}_{<j},\mathbf{x})$进行乘积会导致长句的概率很低
  - 模型本身并没有考虑每个源语言单词被使用的程度，比如一个单词可能会被翻译了很多“次”
- 因此，解码时会使用其它特征与$\mathrm{P}(\mathbf{y}|\mathbf{x})$一起组成模型得分$\mathrm{score}(\mathbf{y},\mathbf{x})$，$\mathrm{score}(\mathbf{y},\mathbf{x})$也作为beam search 的排序依据

$$
\begin{aligned}
\mathrm{score}(\mathbf{y},\mathbf{x}) &= \mathrm{P}(\mathbf{y}|\mathbf{x})/\mathrm{lp}(\mathbf{y}) + \mathrm{cp}(\mathbf{y},\mathbf{x}) \\
\mathrm{lp}(\mathbf{y}) &= \frac{(5+|\mathbf{y}|)^{\alpha}}{(5+1)^{\alpha}} \\
\mathrm{cp}(\mathbf{y},\mathbf{x}) &= \beta \cdot \sum_{i=1}^{|\mathbf{x}|} \log(\min(\sum_{j}^{|\mathbf{y}|} a_{ij}, 1)))
\end{aligned}
$$

  - lp会惩罚译文过短的结果(长度惩罚)；cp会惩罚把某些源语单词对应到很多目标语单词的情况(覆盖度)，被覆盖的程度用$\sum_{j}^{|\mathbf{y}|} a_{ij}$度量；$\alpha$和$\beta$是超参，需要经验性设置

成功案例 - GNMT

- 使用残差连接来训练8层的编码解码器
- 编码器只有最下面2层为双向LSTM
- 解码器只使用最底层输出作为注意力机制的query



效果

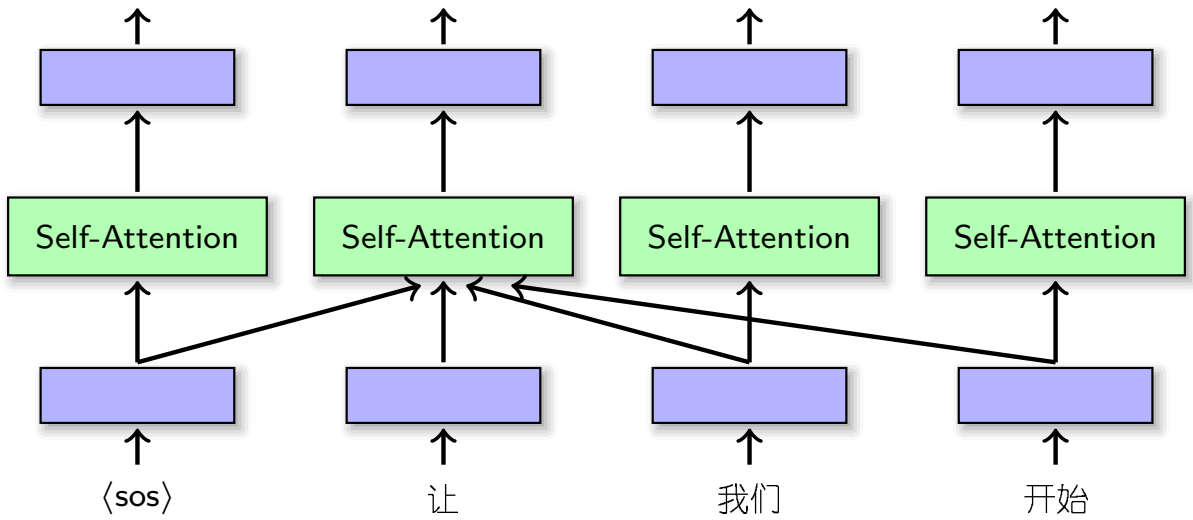
- 在引入注意力机制之前，神经机器翻译（RNNSearch）的性能要弱于统计机器翻译（PBMT）
- 加入注意力机制和深层网络之后，神经机器翻译性能有了很大幅度的提升
- 虽然网络深度增加了，但是通过相应的结构设计和解码策略保证了解码速度

#	BLEU		CPU decoding time
	EN-DE	EN-FR	
PBMT	20.7	37.0	-
RNNSearch	16.5	-	-
LSTM(6 layers)	-	31.5	-
Deep-Att	20.6	37.7	-
GNMT	24.6	39.0	0.2s per sentence

[1]GNMT versus previous state-of-the-art models

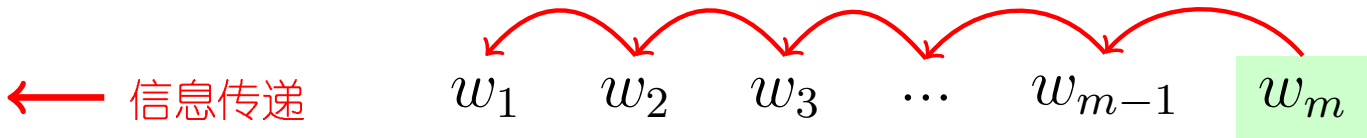
Transformer

Transformer以及自注意机制



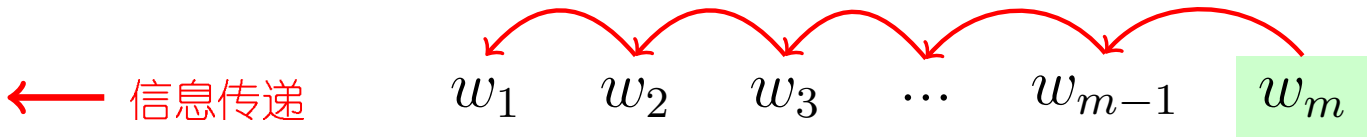
自注意力机制

- 使用循环神经网络对源语、目标语建模进行信息提取效果很好，但是当序列过长时，词汇之间信息传递距离过长，导致模型的信息提取能力变差。

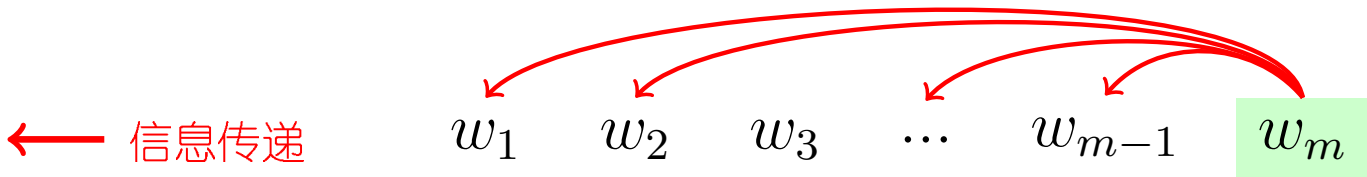


自注意力机制

- 使用循环神经网络对源语、目标语建模进行信息提取效果很好，但是当序列过长时，词汇之间信息传递距离过长，导致模型的信息提取能力变差。

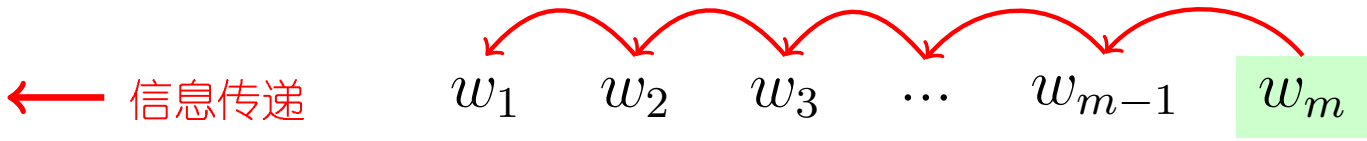


- 能否将不同位置之间的词汇间信息传递的距离拉近为1？

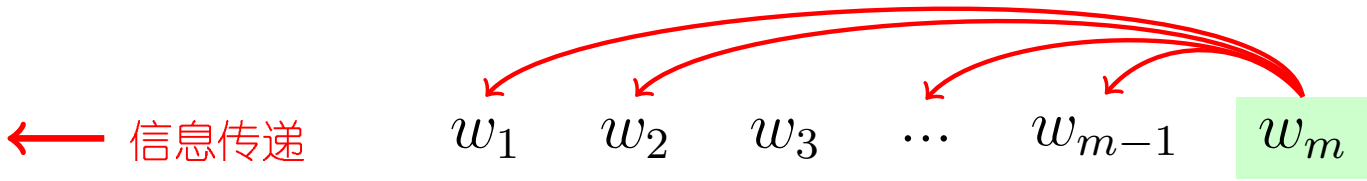


自注意力机制

- 使用循环神经网络对源语、目标语建模进行信息提取效果很好，但是当序列过长时，词汇之间信息传递距离过长，导致模型的信息提取能力变差。



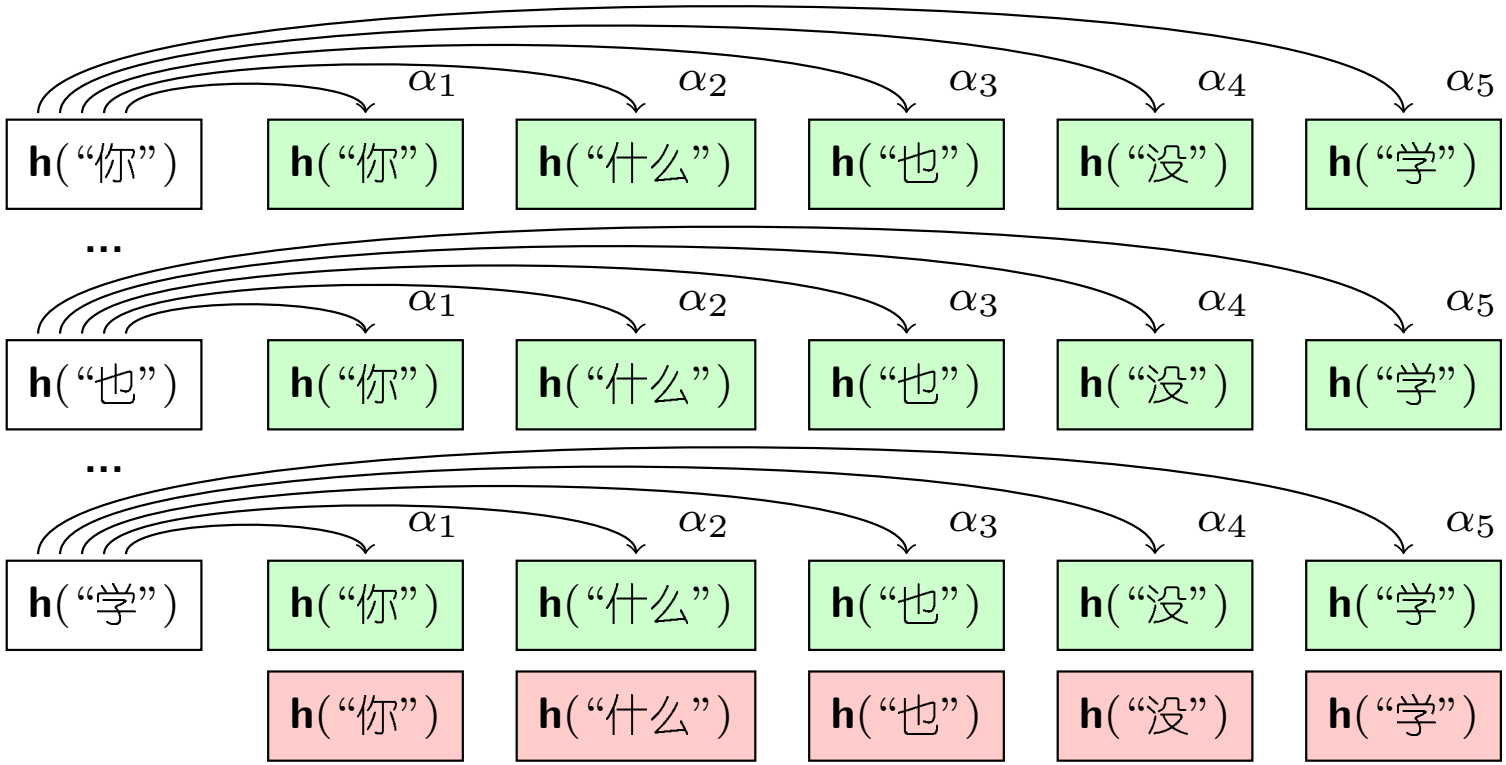
- 能否将不同位置之间的词汇间信息传递的距离拉近为1？



- **自注意力机制(Self-Attention)**可以很好的解决长距离依赖问题，增强信息抽取能力，在长距离语言建模任务取得了很好的效果。 **Attention Is All You Need Vaswani et al., 2017, In Proc. of Neural Information Processing Systems, 6000-6010**

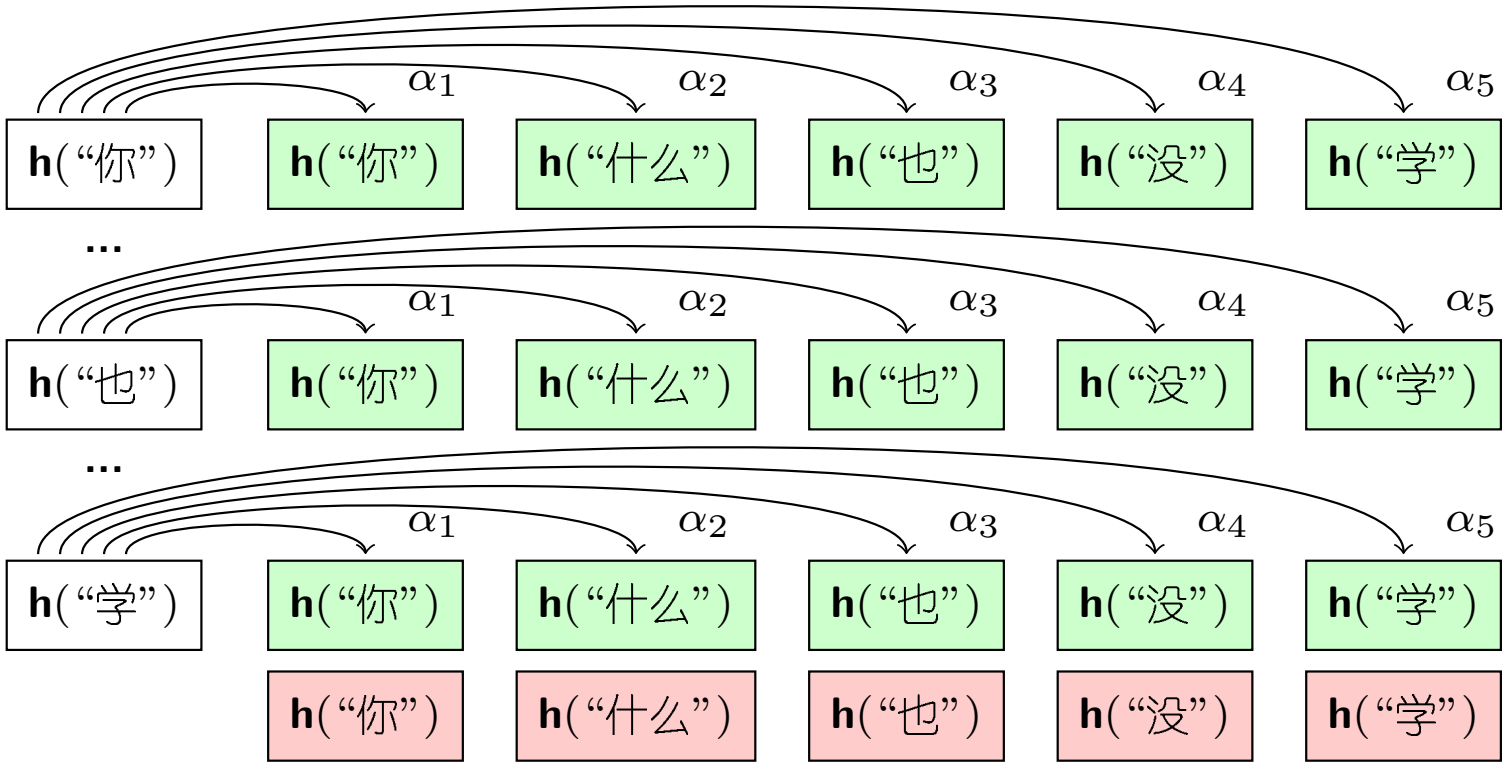
自注意力机制(续)

- 基于前面对注意力机制的介绍，自注意力机制则是将源语言每个位置的表示$\mathbf{h}_i$看做query，同时将所有位置的表示看做**key**和**value**



自注意力机制(续)

- 基于前面对注意力机制的介绍，自注意力机制则是将源语言每个位置的表示$\mathbf{h}_i$看做query，同时将所有位置的表示看做**key**和**value**



- 自注意力模型通过计算源语各位置的匹配程度对value进行加权求和，完成对源语信息的提取

Transformer 介绍

- Transformer是Google在2017年提出的一个新型网络结构，完全基于注意力机制，取得了很好成绩！
- 通过自注意机制能够直接获取全局信息，不像RNN需要逐步进行信息提取，也不像CNN只能获取局部信息，可以并行化操作，提高训练效率

Transformer 介绍

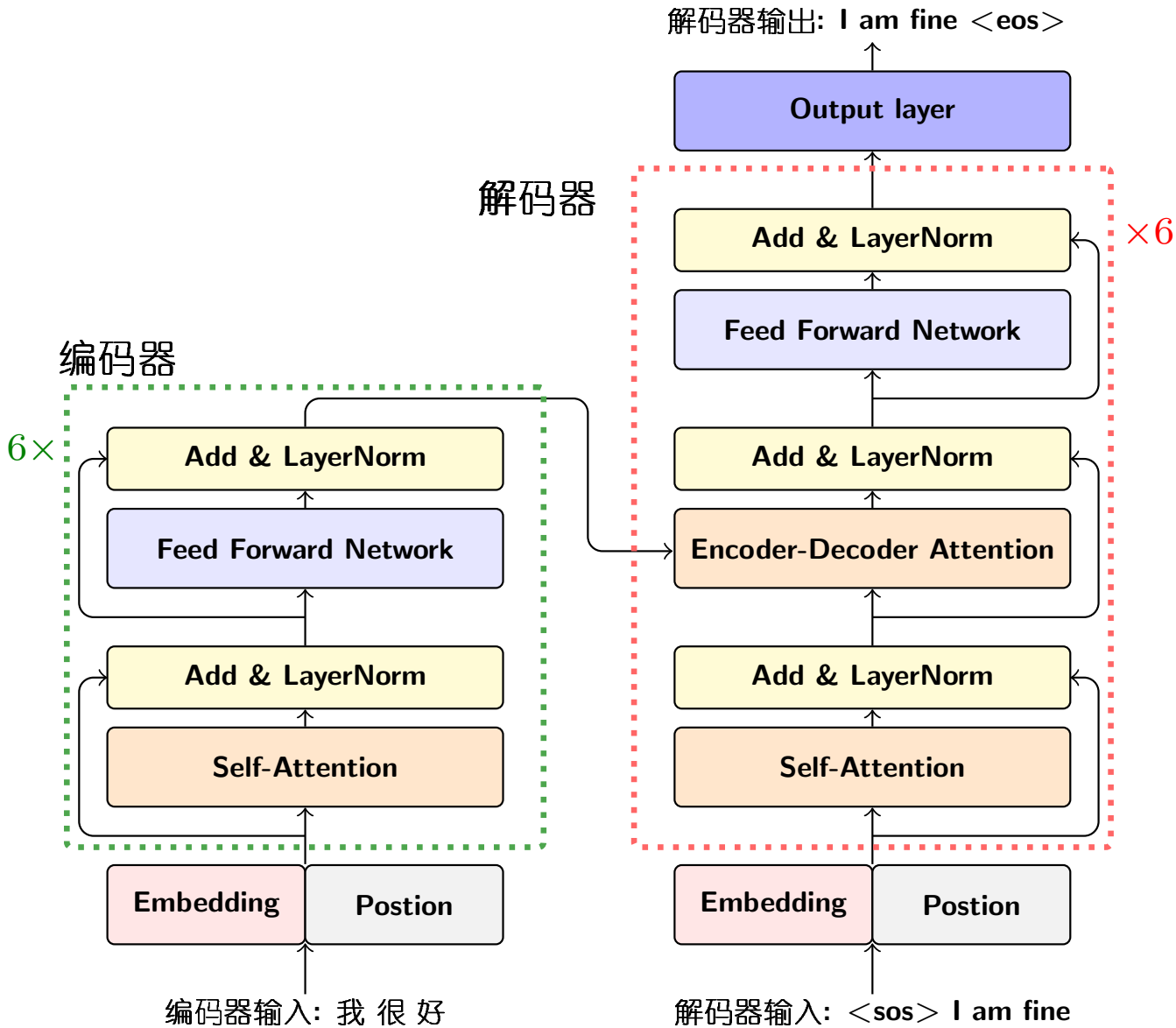
- Transformer是Google在2017年提出的一个新型网络结构，完全基于注意力机制，取得了很好成绩！
- 通过自注意机制能够直接获取全局信息，不像RNN需要逐步进行信息提取，也不像CNN只能获取局部信息，可以并行化操作，提高训练效率
- Transformer不仅仅被用于神经机器翻译任务，还广泛用于其他NLP任务、甚至图像处理任务。目前最火的预训练模型Bert也基于Transformer

#	BLEU		Training Cost(FLOPs)
	EN-DE	EN-FR	
GNMT + RL	24.6	39.92	1.4×10^{20}
ConvS2S	25.16	40.46	1.5×10^{20}
MoE	26.03	40.56	1.2×10^{20}
Transformer (big)	**28.4**	**41.8**	2.3×10^{19}

[2]Transformer versus previous state-of-the-art models

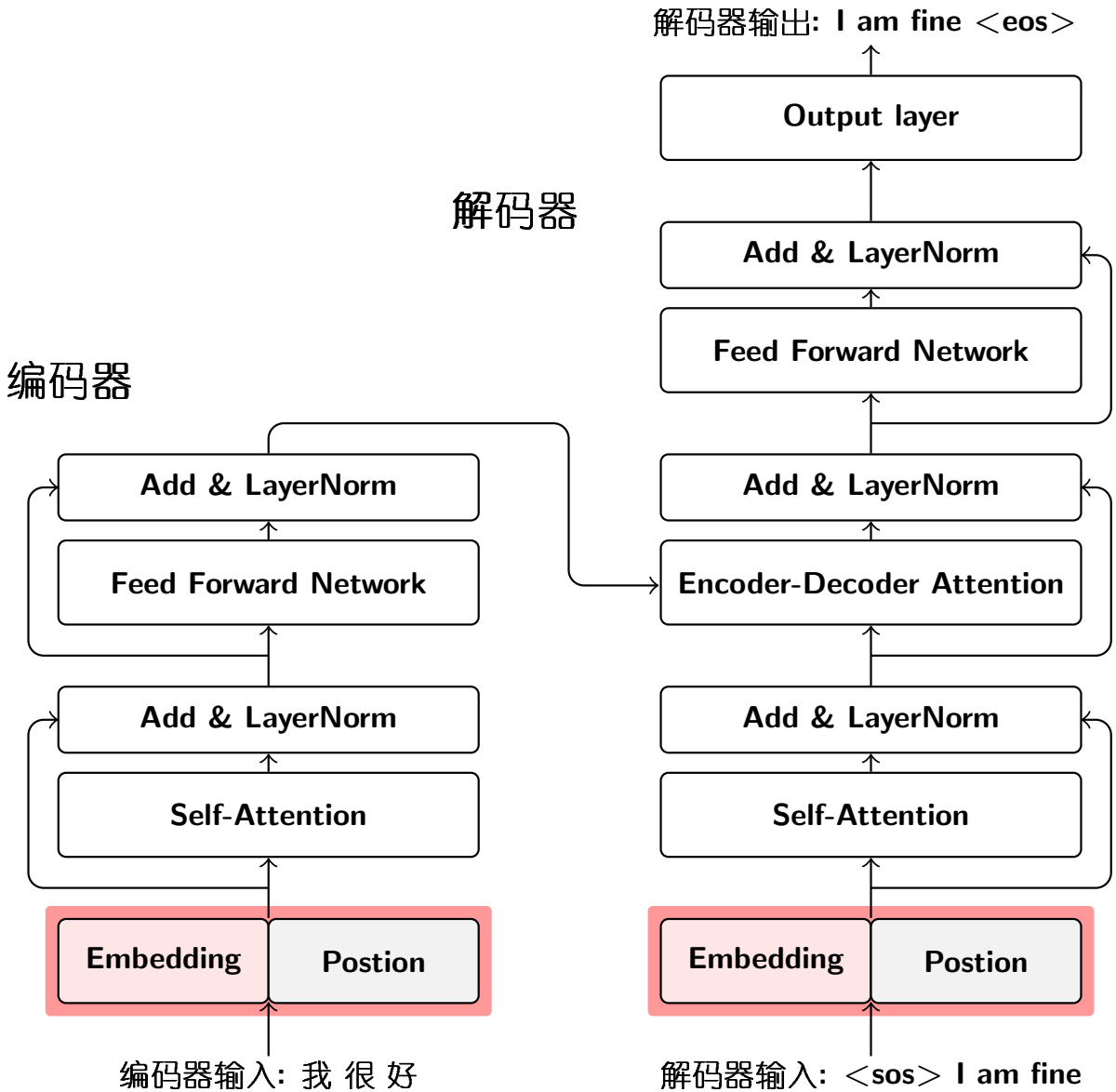
Transformer

- Transformer 总体结构



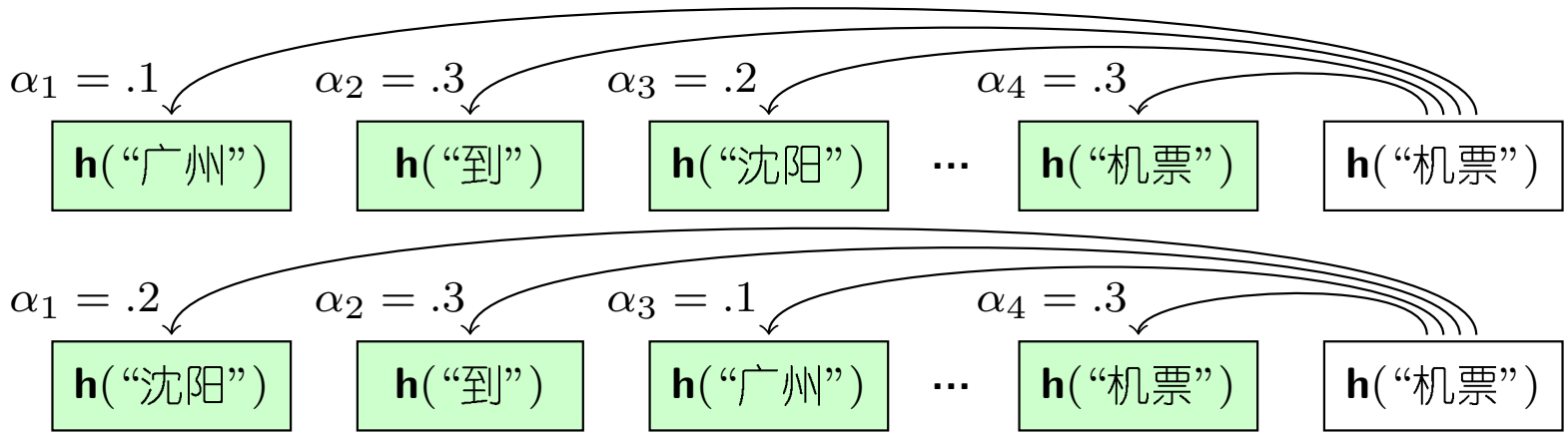
Transformer

- Transformer 输入和位置编码



位置编码

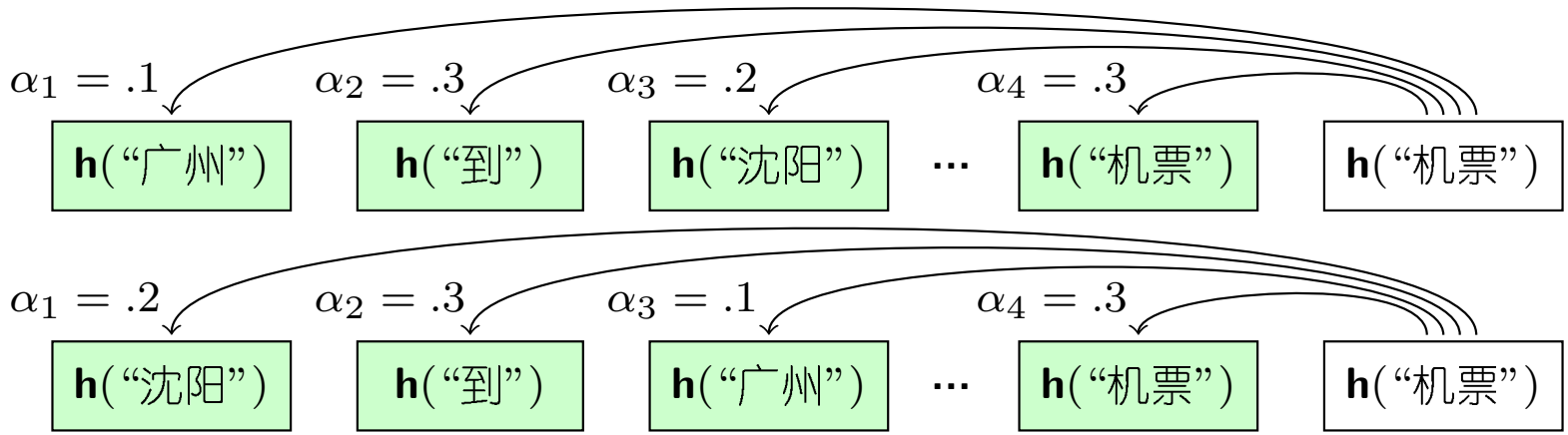
- 自注意力机制与前面的循环神经网络相比，忽略了词之间的顺序关系，例如下面两个语义不同的句子，通过自注意力得到的表示却是相同的



$$\mathbf{C}(\text{“机票”}) = 0.2 \times \mathbf{h}(\text{“沈阳”}) + 0.3 \times \mathbf{h}(\text{“到”}) + 0.1 \times \mathbf{h}(\text{“广州”}) + \ldots + 0.3 \times \mathbf{h}(\text{“机票”})$$

位置编码

- 自注意力机制与前面的循环神经网络相比，忽略了词之间的顺序关系，例如下面两个语义不同的句子，通过自注意力得到的表示却是相同的



$$\mathbf{C}(\text{“机票”}) = 0.2 \times \mathbf{h}(\text{“沈阳”}) + 0.3 \times \mathbf{h}(\text{“到”}) + 0.1 \times \mathbf{h}(\text{“广州”}) + \ldots + 0.3 \times \mathbf{h}(\text{“机票”})$$

- 为了解决这个问题，引入了位置编码

位置编码(续)

- 位置编码的计算方式有很多种，这里使用正余弦函数来编码。式中*pos*代表第几个词，*i*代表词嵌入中的第几维

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

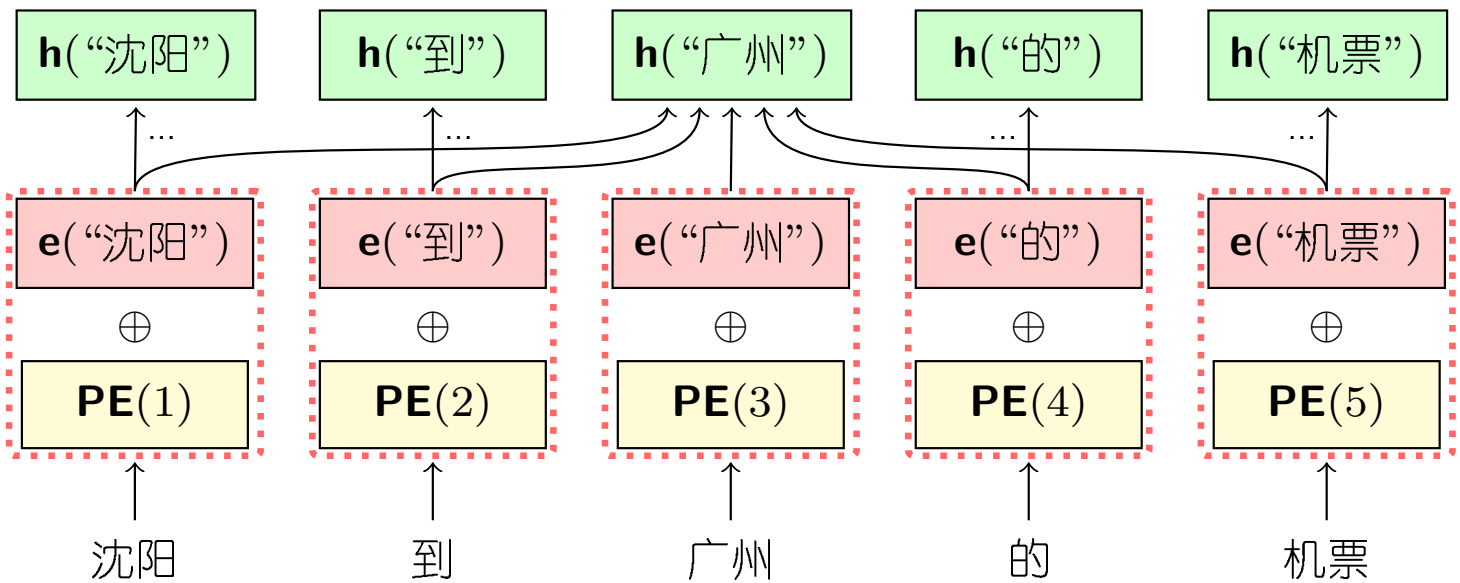
位置编码(续)

- 位置编码的计算方式有很多种，这里使用正余弦函数来编码。式中*pos*代表第几个词，*i*代表词嵌入中的第几维

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

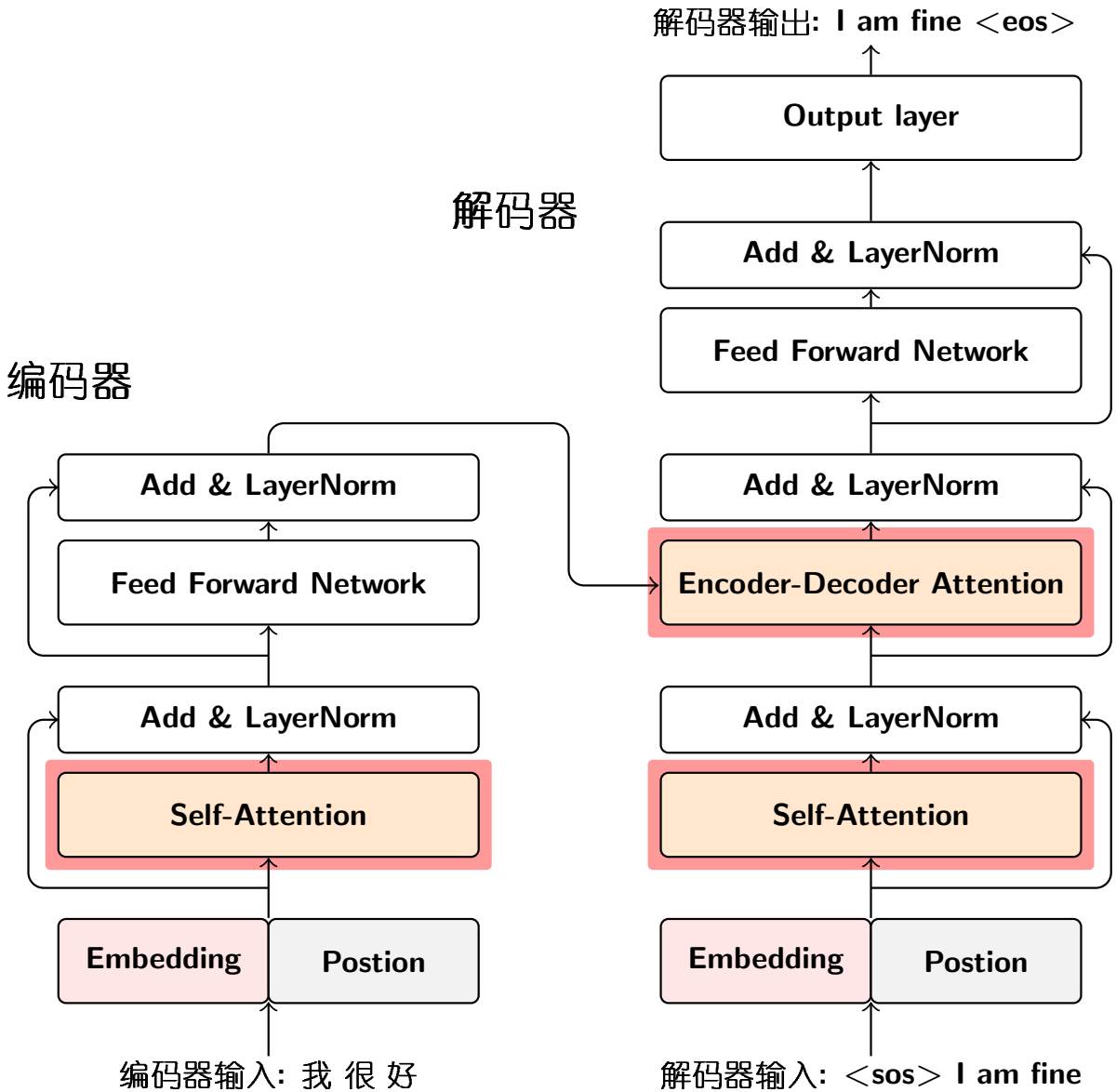
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

- 将得到的位置编码加到原有的词向量中



Transformer

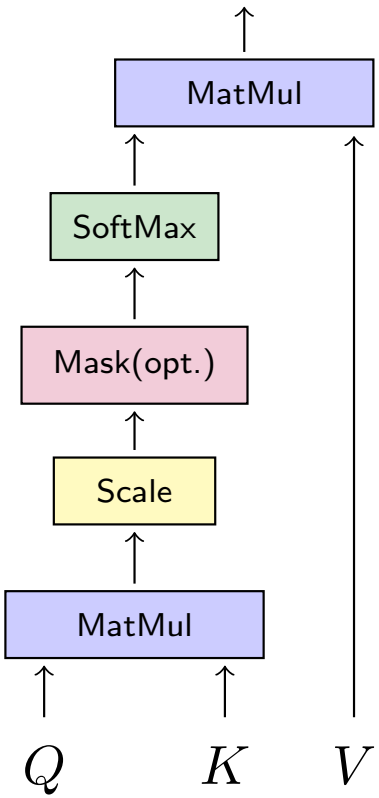
- Transformer 多头自注意力机制



基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

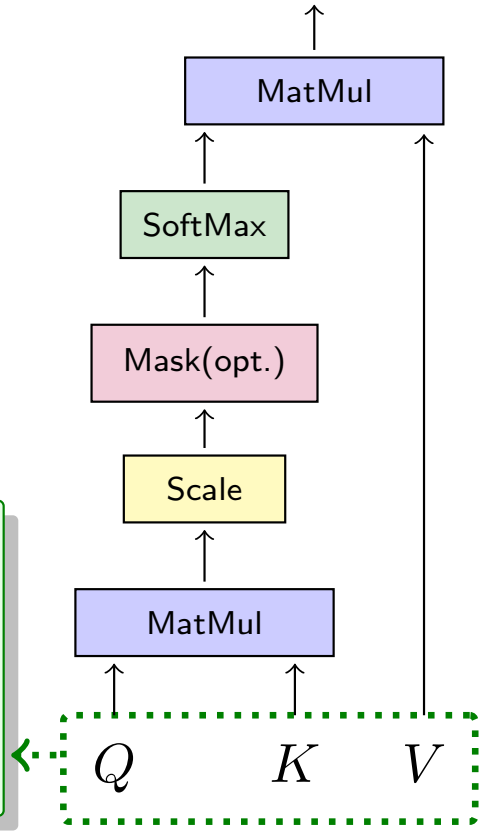
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + Mask)V$$



基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

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



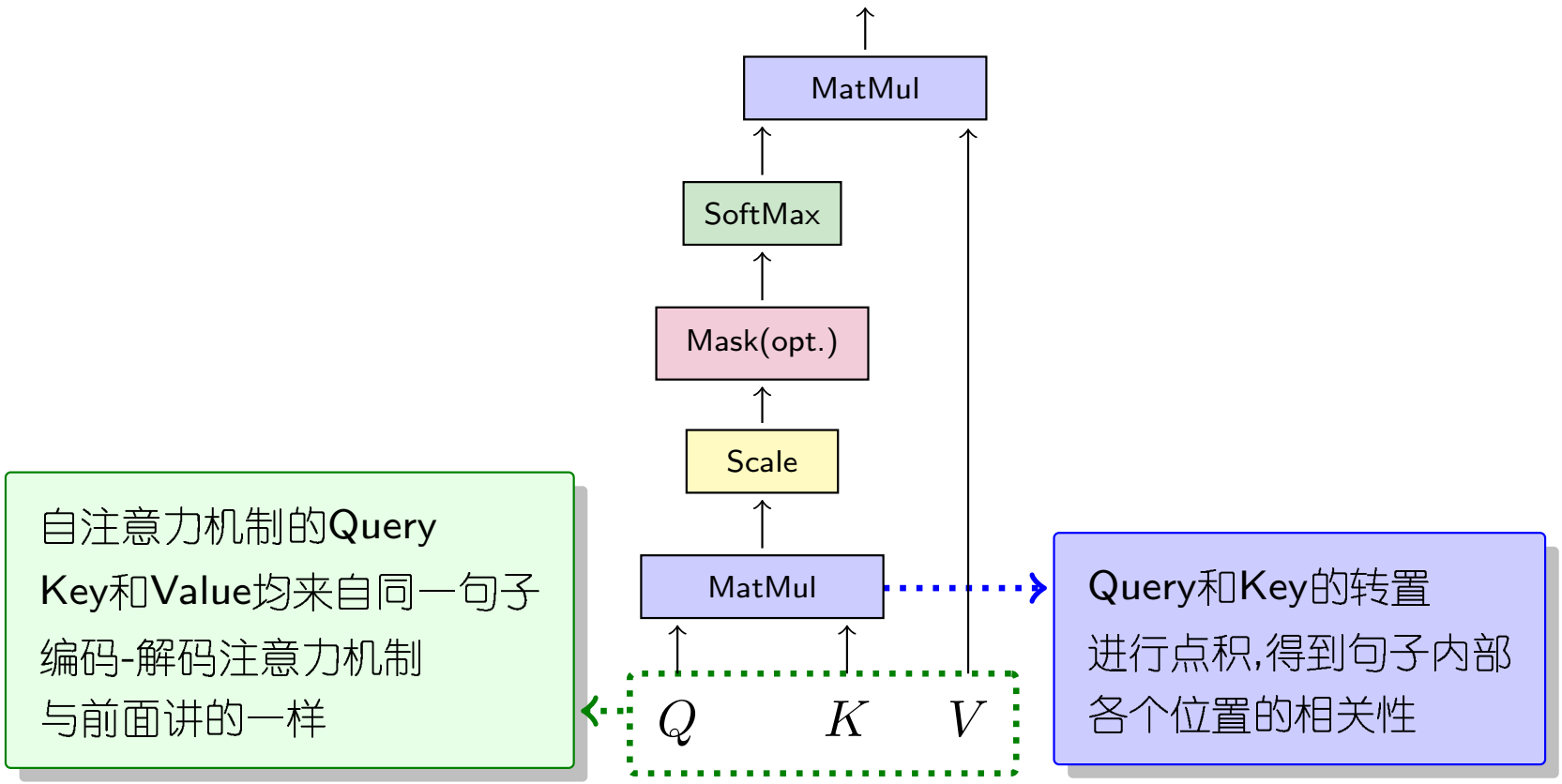
自注意力机制的Query Key和Value均来自同一句子

编码-解码注意力机制与前面讲的一样

基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

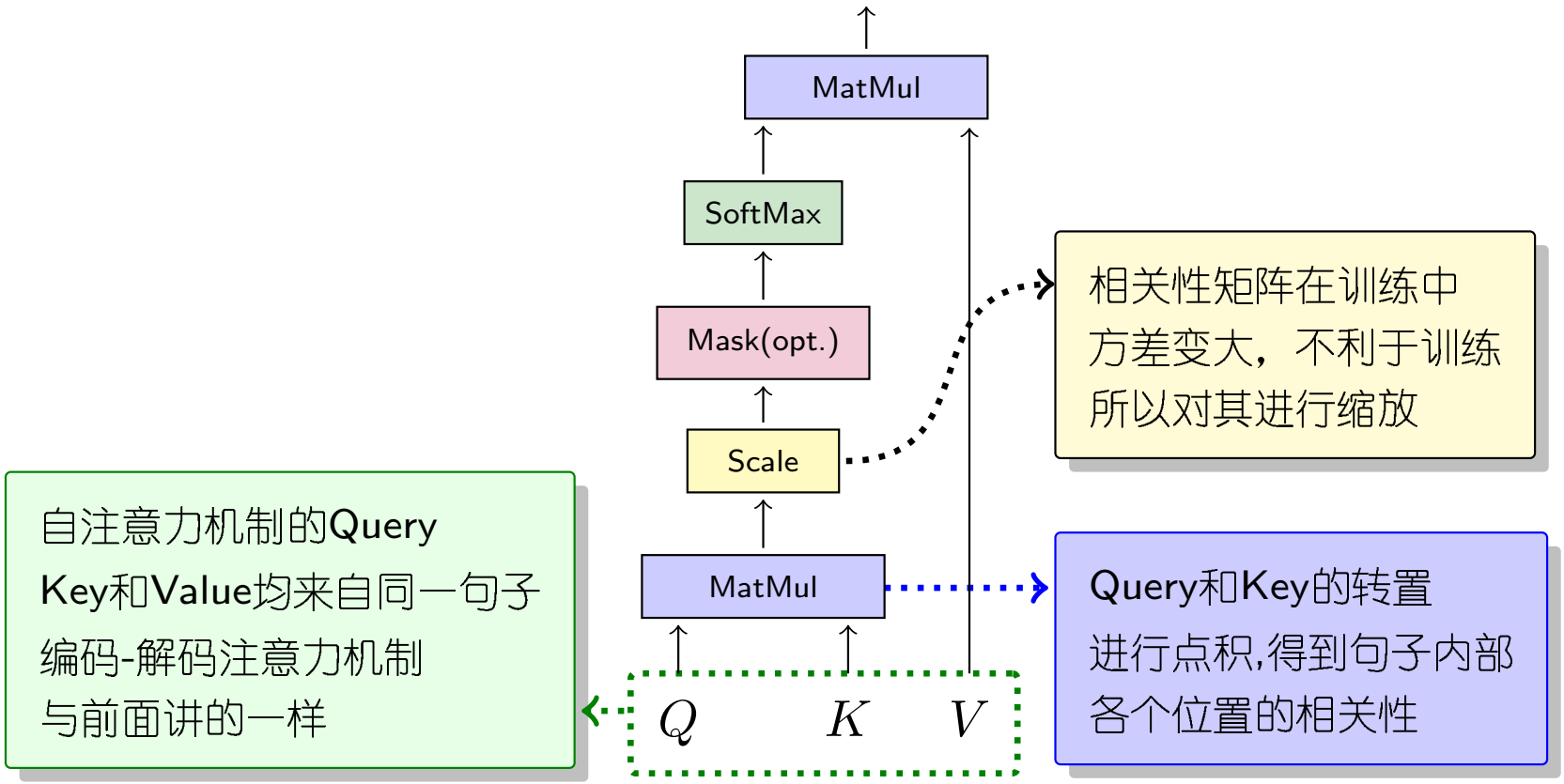
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + Mask)V$$



基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

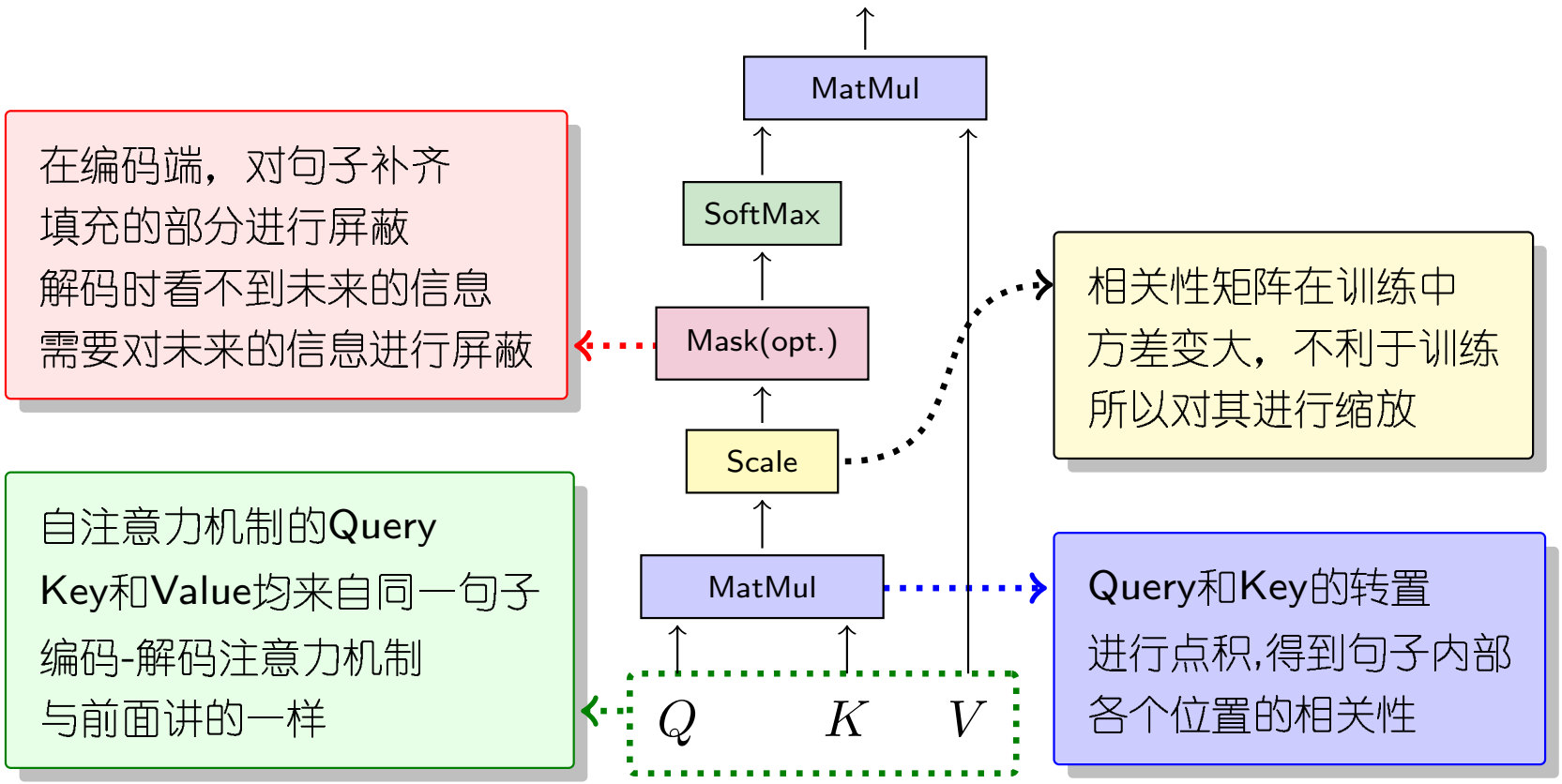
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + Mask)V$$



基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

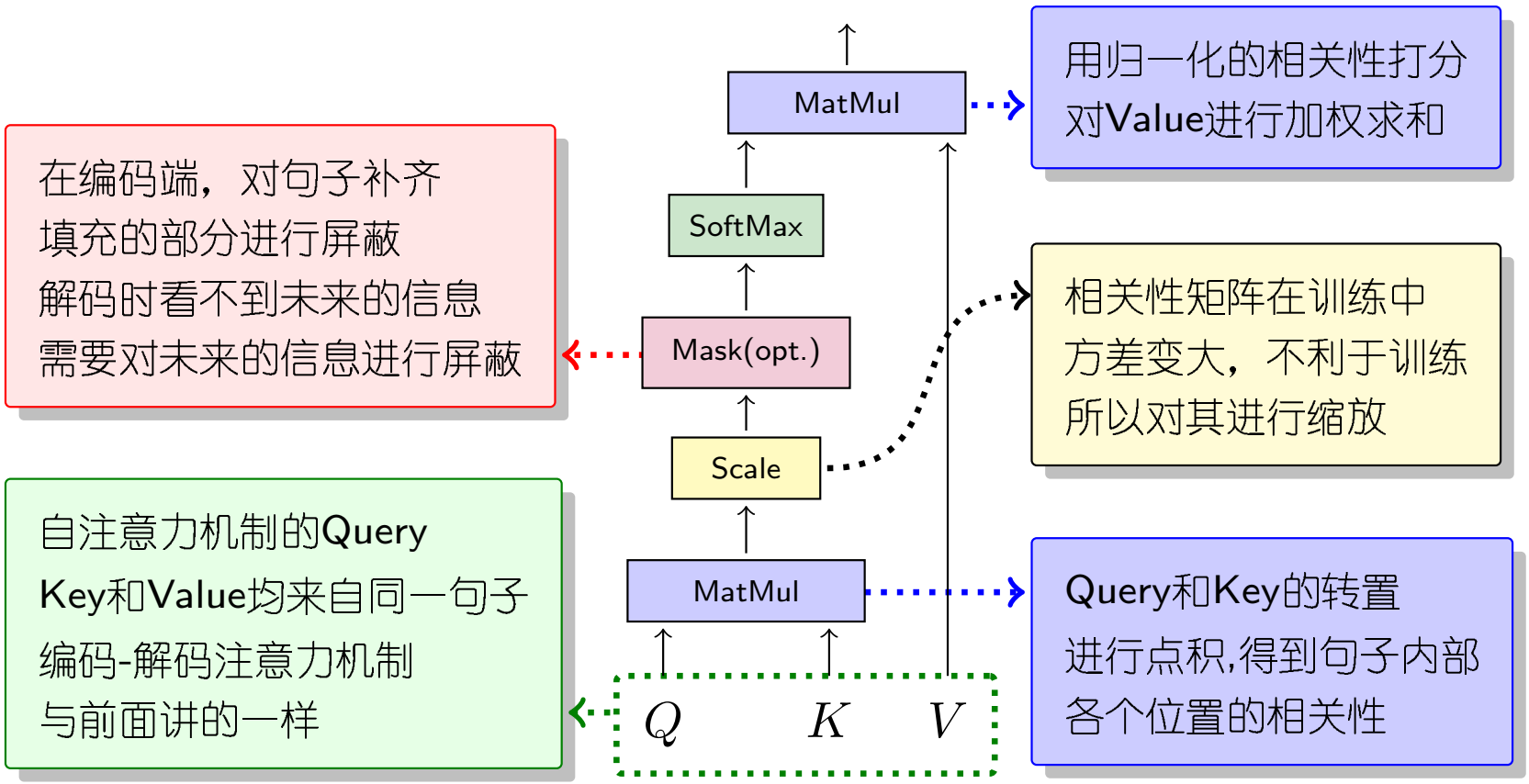
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}} + Mask)V$$



基于点乘的注意力机制

- Transformer使用点乘的自注意力方法来捕获句子内部各个位置之间的相似性：

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



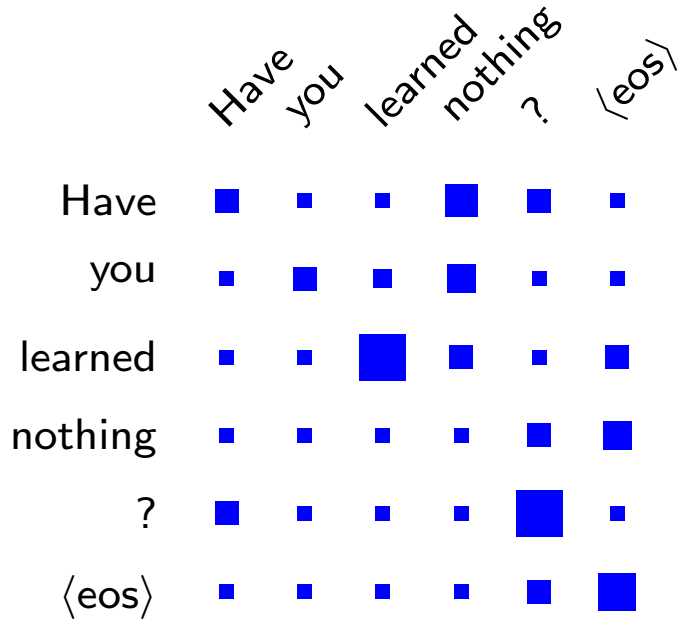
Mask

- 对于源语和目标语的输入，由于需要进行batch处理，有些部分是填充的（Padding），需要用Mask进行屏蔽
- 对于解码器来说，由于在预测的时候是自左向右进行的，为了保持训练解码一致，需要对未来信息进行屏蔽

Mask

- 对于源语和目标语的输入，由于需要进行batch处理，有些部分是填充的（Padding），需要用Mask进行屏蔽
- 对于解码器来说，由于在预测的时候是自左向右进行的，为了保持训练解码一致，需要对未来信息进行屏蔽

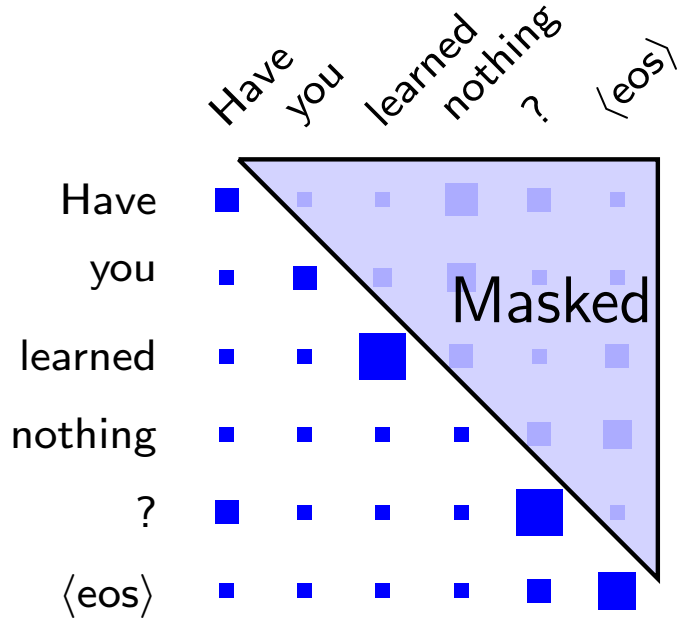
$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} + Mask\right)$$



Mask

- 对于源语和目标语的输入，由于需要进行batch处理，有些部分是填充的（Padding），需要用Mask进行屏蔽
- 对于解码器来说，由于在预测的时候是自左向右进行的，为了保持训练解码一致，需要对未来信息进行屏蔽

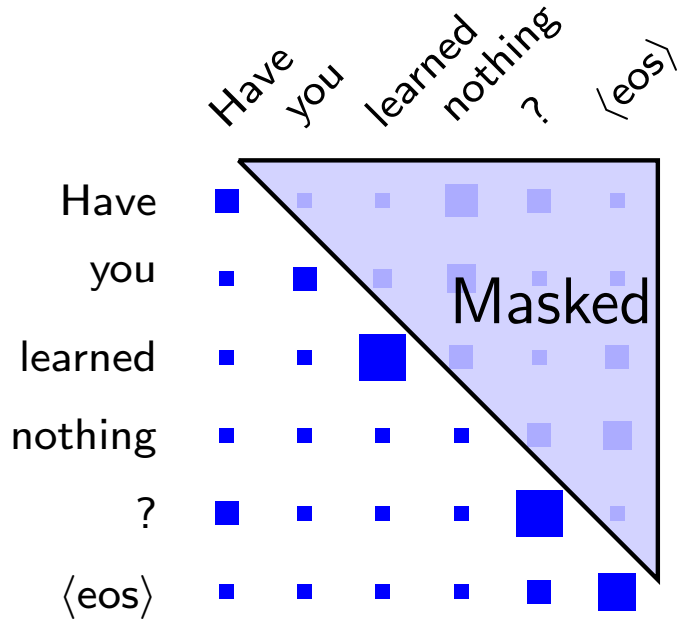
$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} + Mask\right)$$

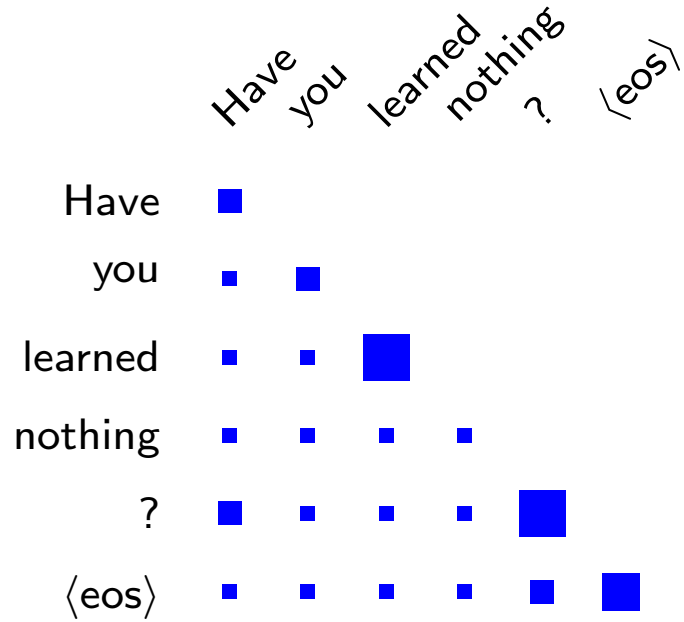


Mask

- 对于源语和目标语的输入，由于需要进行batch处理，有些部分是填充的（Padding），需要用Mask进行屏蔽
- 对于解码器来说，由于在预测的时候是自左向右进行的，为了保持训练解码一致，需要对未来信息进行屏蔽

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} + Mask\right)$$



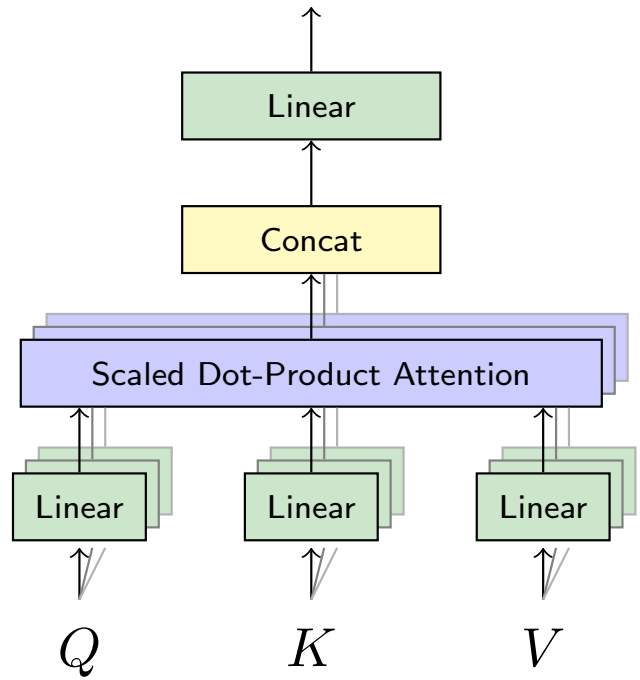


多头自注意力模型

- Transformer首次提出了多头注意力机制，将输入的Query、Key、Value沿着隐层维度切分为h个子集，分别进行注意力操作，取得了很好的效果

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



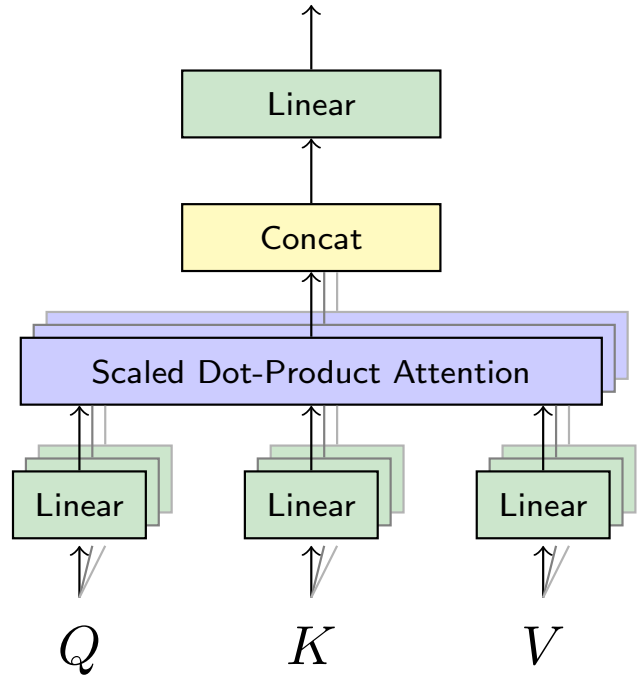
多头自注意力模型

- Transformer首次提出了多头注意力机制，将输入的Query、Key、Value沿着隐层维度切分为h个子集，分别进行注意力操作，取得了很好的效果

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

将输入沿着隐层维度分为h个子集
首先对不同的子集分别进行线性变换
然后执行h次基于点乘的注意力操作
最后将注意力操作的输出连接到一起
h通常设置为8



多头自注意力模型

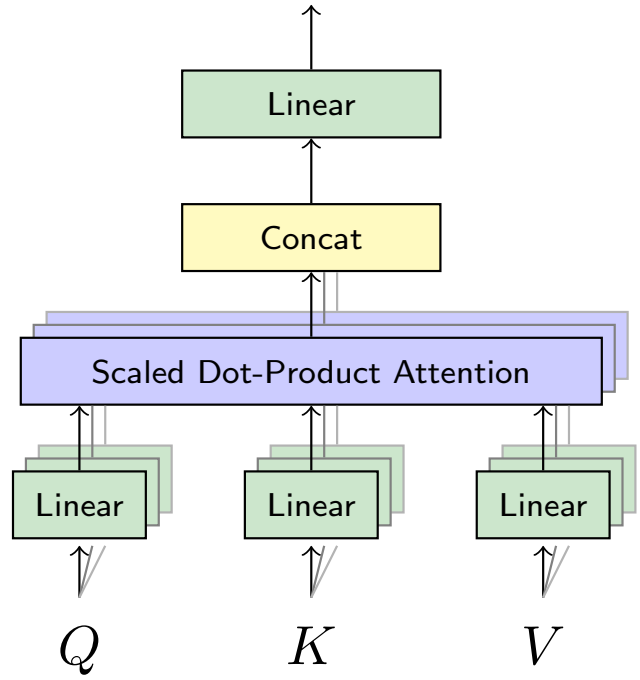
- Transformer首次提出了多头注意力机制，将输入的Query、Key、Value沿着隐层维度切分为h个子集，分别进行注意力操作，取得了很好的效果

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

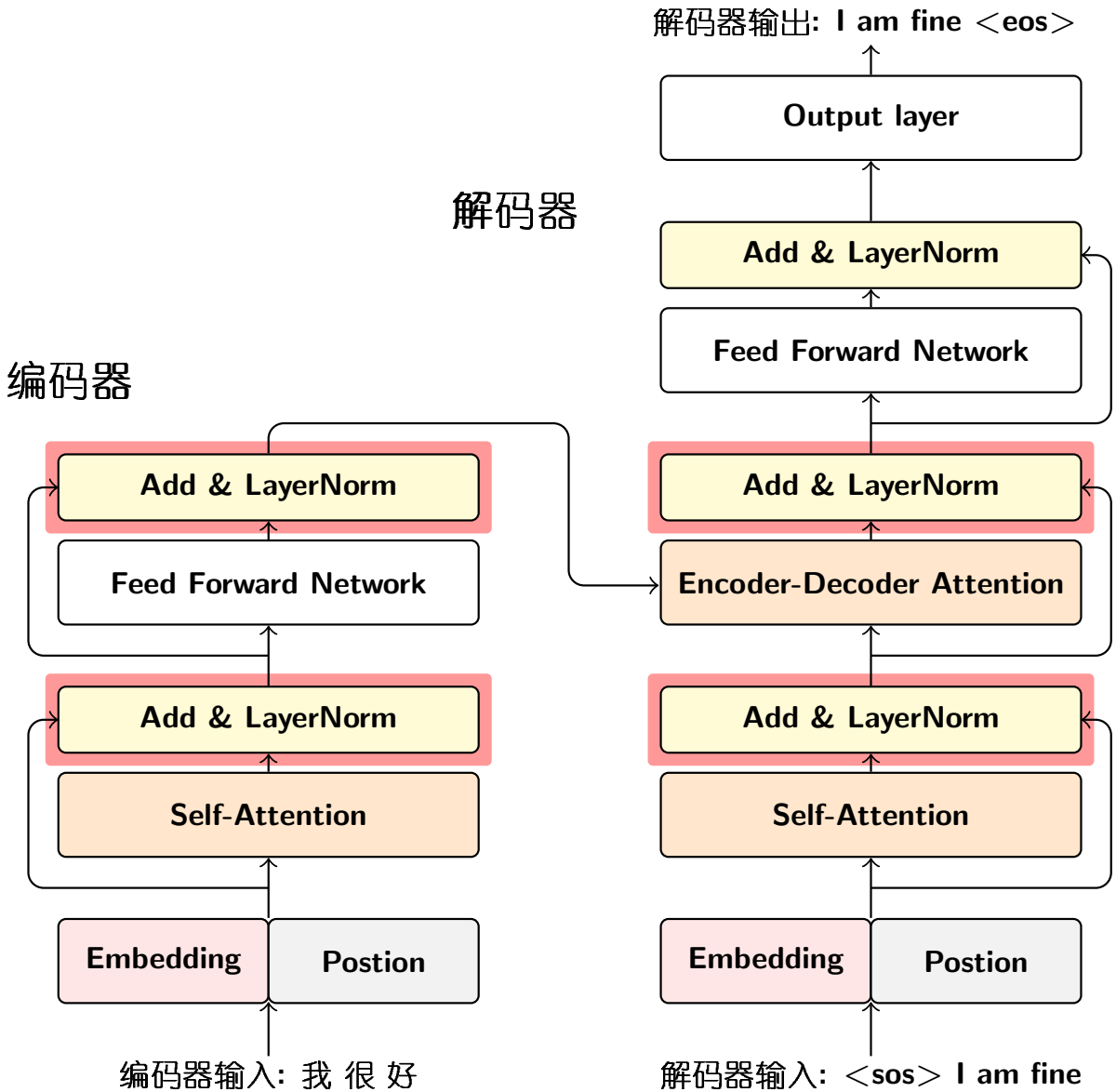
将输入沿着隐层维度分为h个子集
首先对不同的子集分别进行线性变换
然后执行h次基于点乘的注意力操作
最后将注意力操作的输出连接到一起
h通常设置为8

这样做是希望模型在不同的子空间中学习到更丰富的信息



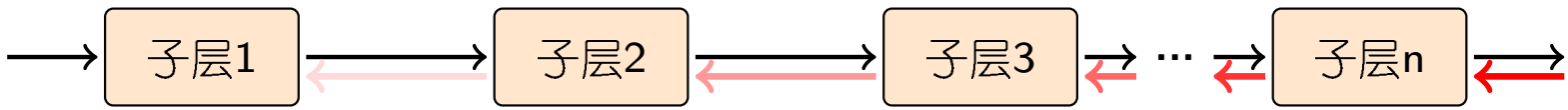
Transformer

- Transformer 残差和层正则化



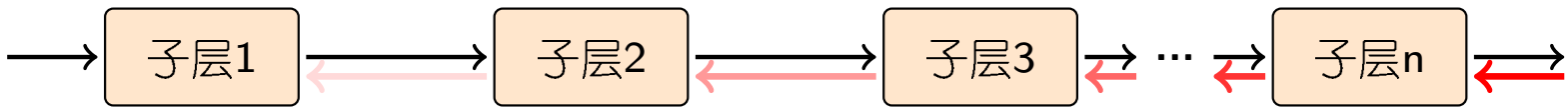
残差&层正则化

- 在Transformer中，编码器、解码器分别由6层网络组成，每层网络又包含多个子层（自注意力网络、前馈神经网络）。Transformer实际上是一个很深的网络结构，在训练过程中容易出现梯度消失的情况



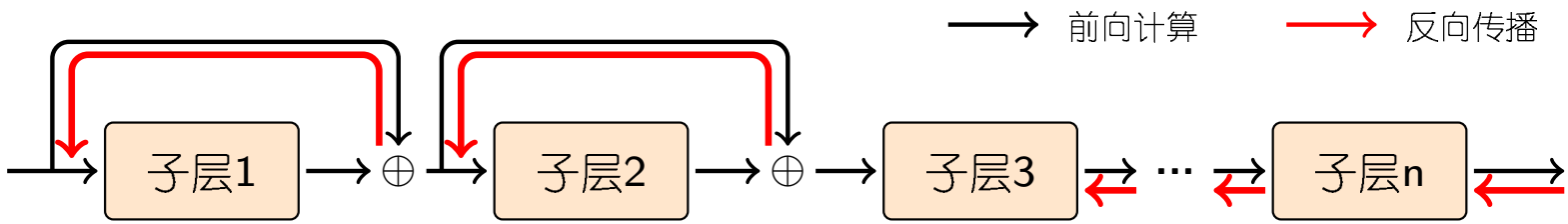
残差&层正则化

- 在Transformer中，编码器、解码器分别由6层网络组成，每层网络又包含多个子层（自注意力网络、前馈神经网络）。Transformer实际上是一个很深的网络结构，在训练过程中容易出现梯度消失的情况



- 在这里引入了在图像领域用来训练深层网络的技术，**残差网络**来避免上述问题

$$x_{l+1} = x_l + \mathcal{F}(x_l)$$



残差&层正则化(续)

- 在Transformer的训练过程中，由于引入了残差操作，将前面所有层的输出加到一起。这样会导致高层的参数分布不断变大，造成训练过程不稳定、训练时间较长。 为了避免这种情况，在每层中加入了层正则化操作
  - 使用均值和方差对样本进行平移缩放，将数据规范化为均值为0，方差为1的标准分布

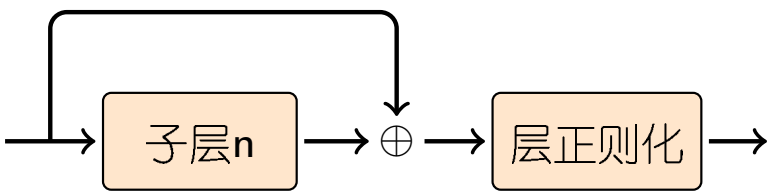
$$\mathrm{LN}(x) = g \cdot \frac{x - \mu}{\sigma} + b$$

残差&层正则化(续)

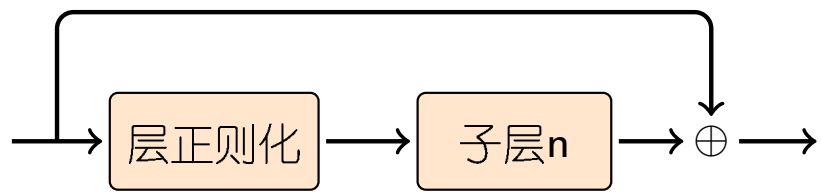
- 在Transformer的训练过程中，由于引入了残差操作，将前面所有层的输出加到一起。这样会导致高层的参数分布不断变大，造成训练过程不稳定、训练时间较长。 为了避免这种情况，在每层中加入了层正则化操作
  - 使用均值和方差对样本进行平移缩放，将数据规范化为均值为0，方差为1的标准分布

$$\mathrm{LN}(x) = g \cdot \frac{x - \mu}{\sigma} + b$$

- 在Transformer中经常使用的层正则化操作有两种，分别是后正则化和前正则化



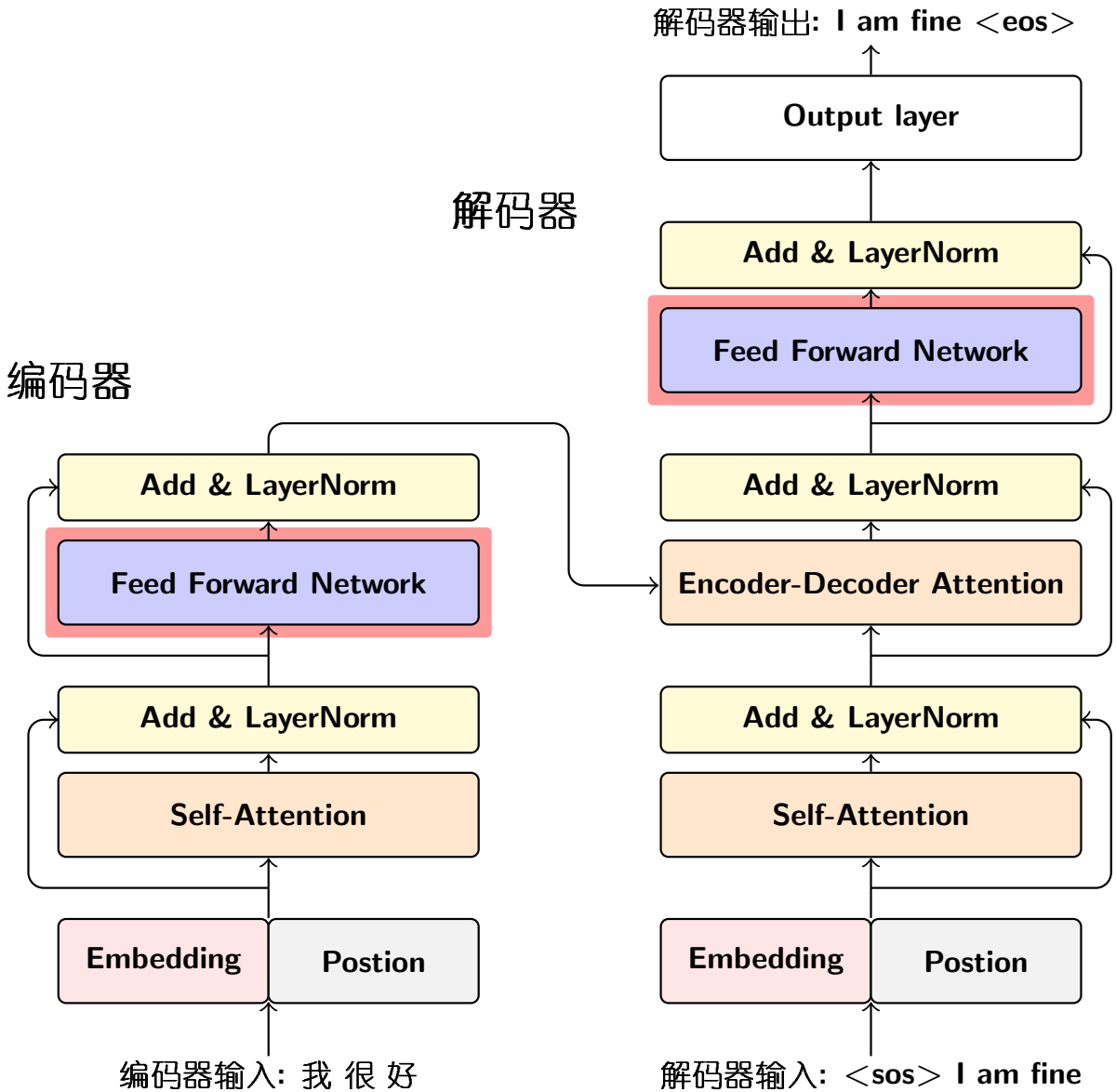
后正则化



前正则化

Transformer

- Transformer 前馈全连接网络



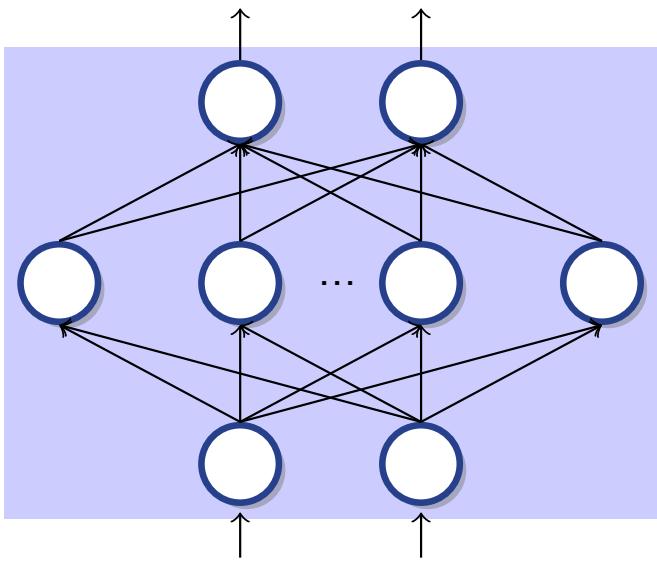
前馈全连接网络

- 在每层中，除了注意力操作，还包含了一个全连接的前馈神经网络，网络中包含两次线性变换和一次非线性变换(ReLU激活函数)，每层的前馈神经网络参数不共享

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

- 通常情况下，注意力部分的隐层维度为512，FFN部分的隐层维度设置为2048

全连接网络的作用主要体现在
将经过注意力操作之后的表示
映射到更大的网络空间中
提升了网络模型的表示能力
实验证明，去掉全连接网络
会对模型的性能造成影响



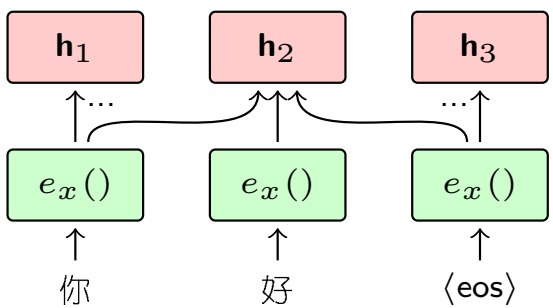
训练

- Transformer一个主要的改进就是可以进行并行化训练。由于之前的RNN是基于时序进行训练，只有在前一时刻训练结束才能进行当前时刻的训练，而Transformer将任意时刻的输入信息之间的距离拉近为1，因此可以并行化训练，提高训练效率

训练

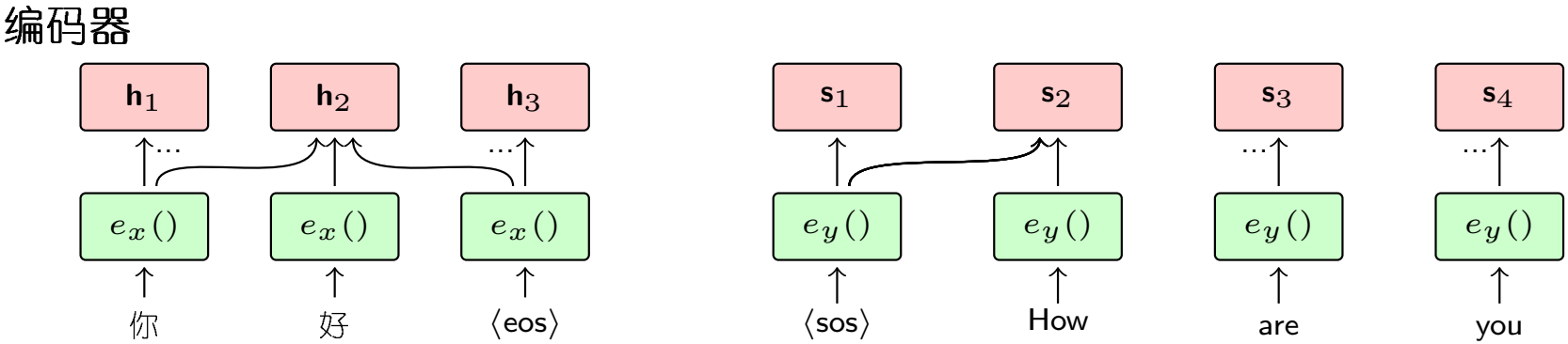
- Transformer一个主要的改进就是可以进行并行化训练。由于之前的RNN是基于时序进行训练，只有在前一时刻训练结束才能进行当前时刻的训练，而Transformer将任意时刻的输入信息之间的距离拉近为1，因此可以并行化训练，提高训练效率

编码器



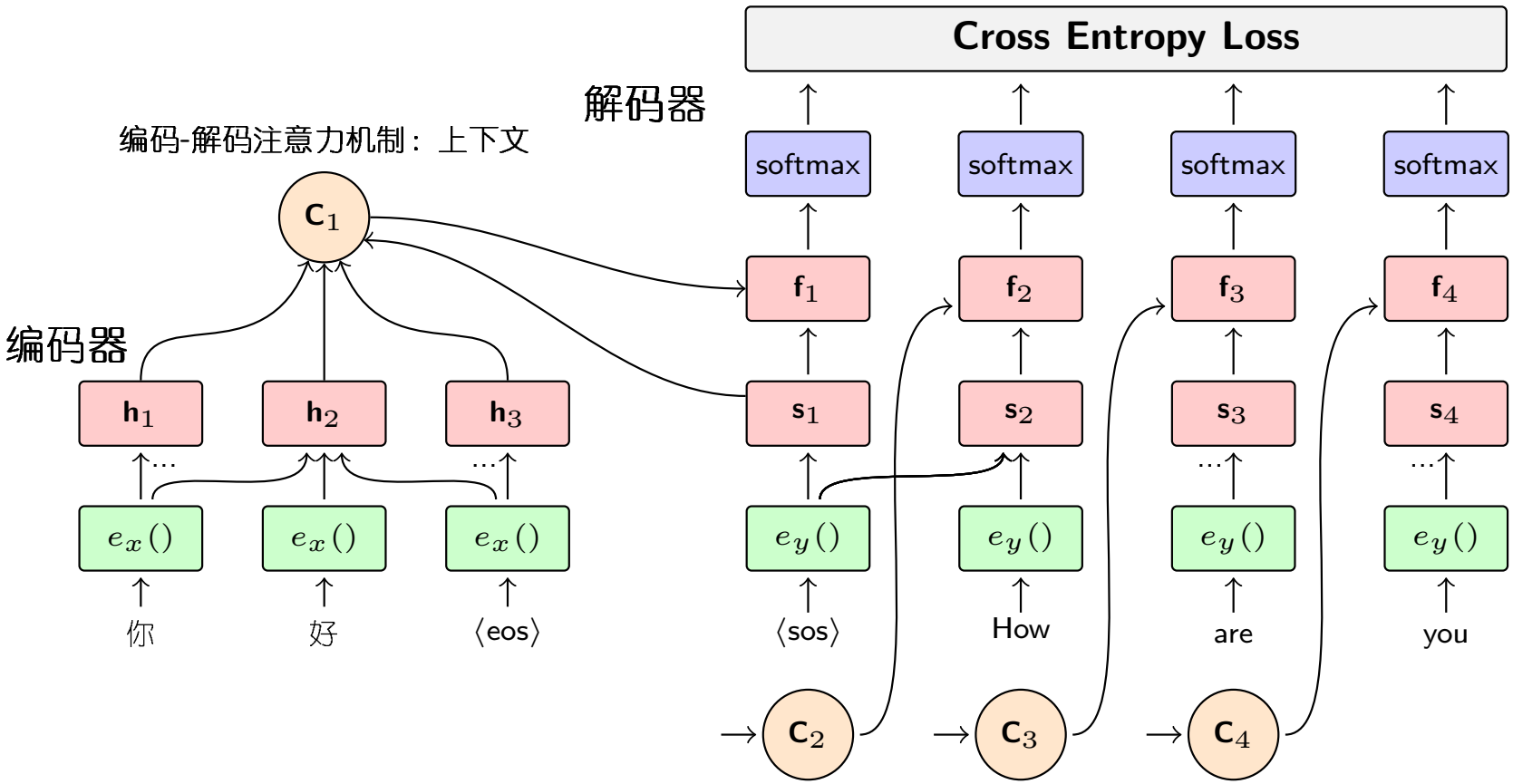
训练

- Transformer一个主要的改进就是可以进行并行化训练。由于之前的RNN是基于时序进行训练，只有在前一时刻训练结束才能进行当前时刻的训练，而Transformer将任意时刻的输入信息之间的距离拉近为1，因此可以并行化训练，提高训练效率



训练

- Transformer一个主要的改进就是可以进行并行化训练。由于之前的RNN是基于时序进行训练，只有在前一时刻训练结束才能进行当前时刻的训练，而Transformer将任意时刻的输入信息之间的距离拉近为1，因此可以并行化训练，提高训练效率

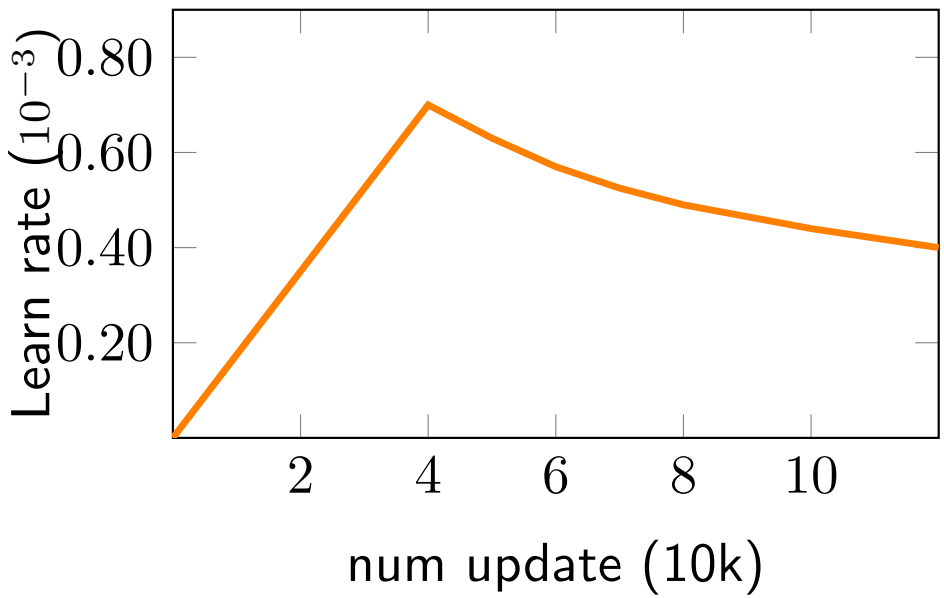


优化器

- **优化器**：使用Adam优化器，β_1=0.9，β_2=0.98，$\epsilon = 10^{-9}$
- **学习率**：关于学习率的设置，引入了warmup策略，在训练初期，学习率从一个较小的初始值逐渐增大，当到达一定的步数，学习率再逐渐减小

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step^{-0.5}, step \cdot \text{warmup\_steps}^{-1.5})$$

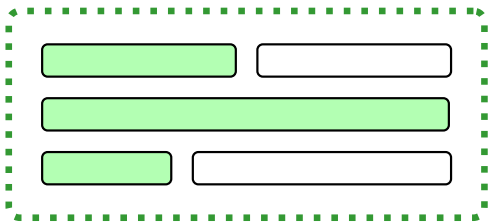
这样做可以减缓在训练初期的不稳定现象，保持分布平稳，通常warmup_steps通常设置为4000



训练配置

- **Mini Batch训练**：选择批量的数据作为训练样本，来计算损失函数，提高训练效率
  - Mini Batch大小通常设置为2048/4096(token数)
  - 通常对句子长度进行排序，选取长度相近的句子组成一个batch，可以减少padding数量，提高训练效率

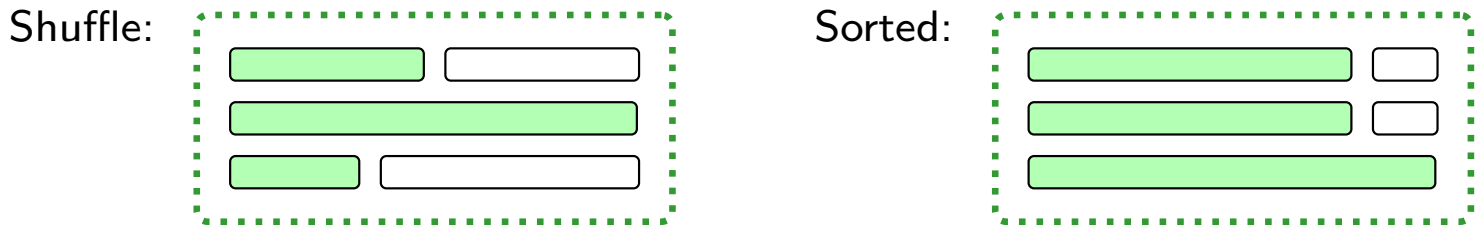
Shuffle:



Sorted:

训练配置

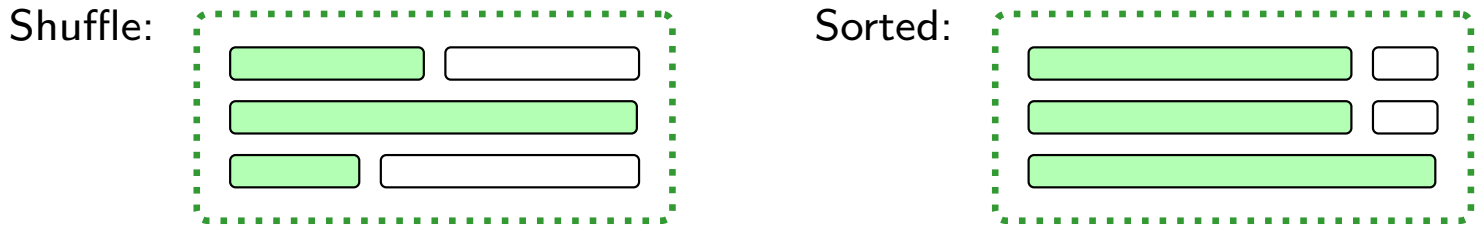
- **Mini Batch训练**：选择批量的数据作为训练样本，来计算损失函数，提高训练效率
  - Mini Batch大小通常设置为2048/4096(token数)
  - 通常对句子长度进行排序，选取长度相近的句子组成一个batch，可以减少padding数量，提高训练效率



- **Dropout** ：为了防止网络训练过拟合，加入了Dropout操作。在四个地方用到了Dropout，词嵌入和位置编码、残差连接、注意力操作和前馈神经网络。Drop率通常设置为0.1

训练配置

- **Mini Batch训练**：选择批量的数据作为训练样本，来计算损失函数，提高训练效率
  - Mini Batch大小通常设置为2048/4096(token数)
  - 通常对句子长度进行排序，选取长度相近的句子组成一个batch，可以减少padding数量，提高训练效率



- **Dropout** ：为了防止网络训练过拟合，加入了Dropout操作。在四个地方用到了Dropout，词嵌入和位置编码、残差连接、注意力操作和前馈神经网络。Drop率通常设置为0.1
- **标签平滑**：学习一个较平滑的的目标，可以提升泛化能力，防止过拟合 :)

训练配置(续)

- **Transformer Base**：标准的Transformer结构，解码器编码器均包含6层，隐层维度为512，前馈神经网络维度为2048，多头注意力机制为8头，Dropout设为0.1
- **Transformer Big**：为了提升网络的表示能力，在Base的基础上增大隐层维度至1024，前馈神经网络的维度变为4096，多头注意力机制为16头，Dropout设为0.3
- **Transformer Deep**：加深编码器网络层数可以进一步提升网络的性能，但简单堆叠网络层数会出现梯度消失问题，导致训练无法收敛。需要使用DLCL、正则化前作等方法来训练更深的网络。

#	BLEU		params
	EN-DE	EN-FR	
Transformer Base	27.3	38.1	65×10^6
Transformer Big	28.4	41.8	213×10^6
Transformer Deep(48层)	30.2	43.1	194×10^6

推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！

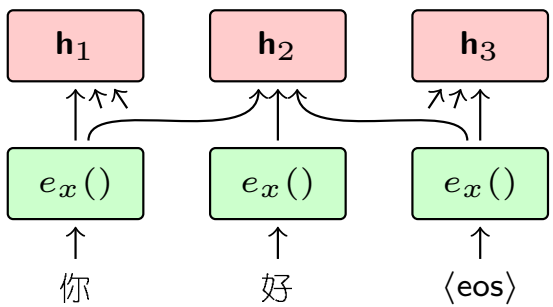
推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network

推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network

编码器

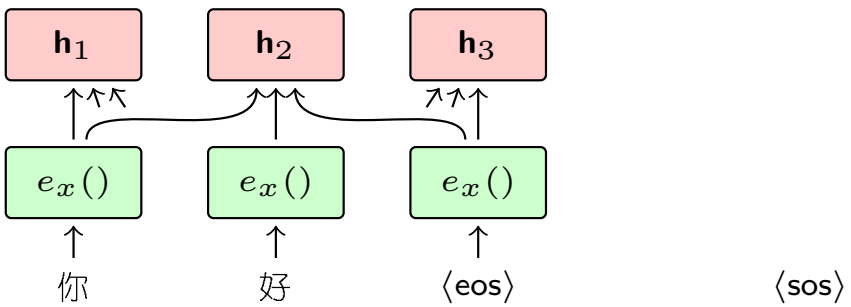


推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network

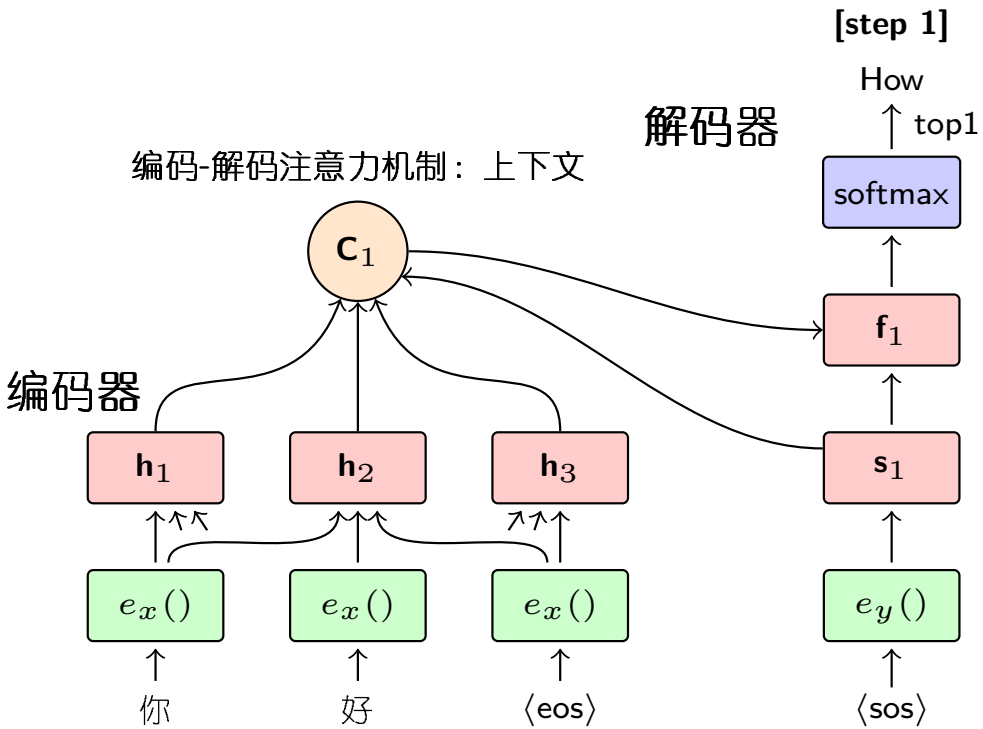
[step 1]

编码器



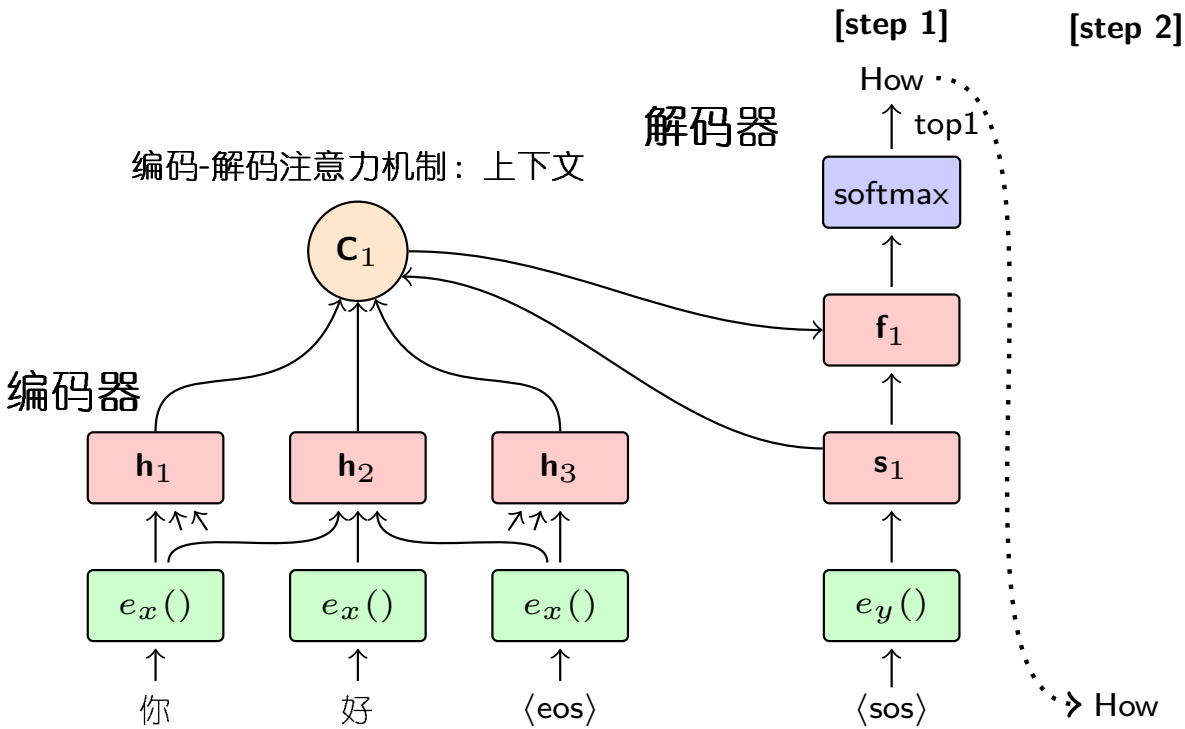
推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network



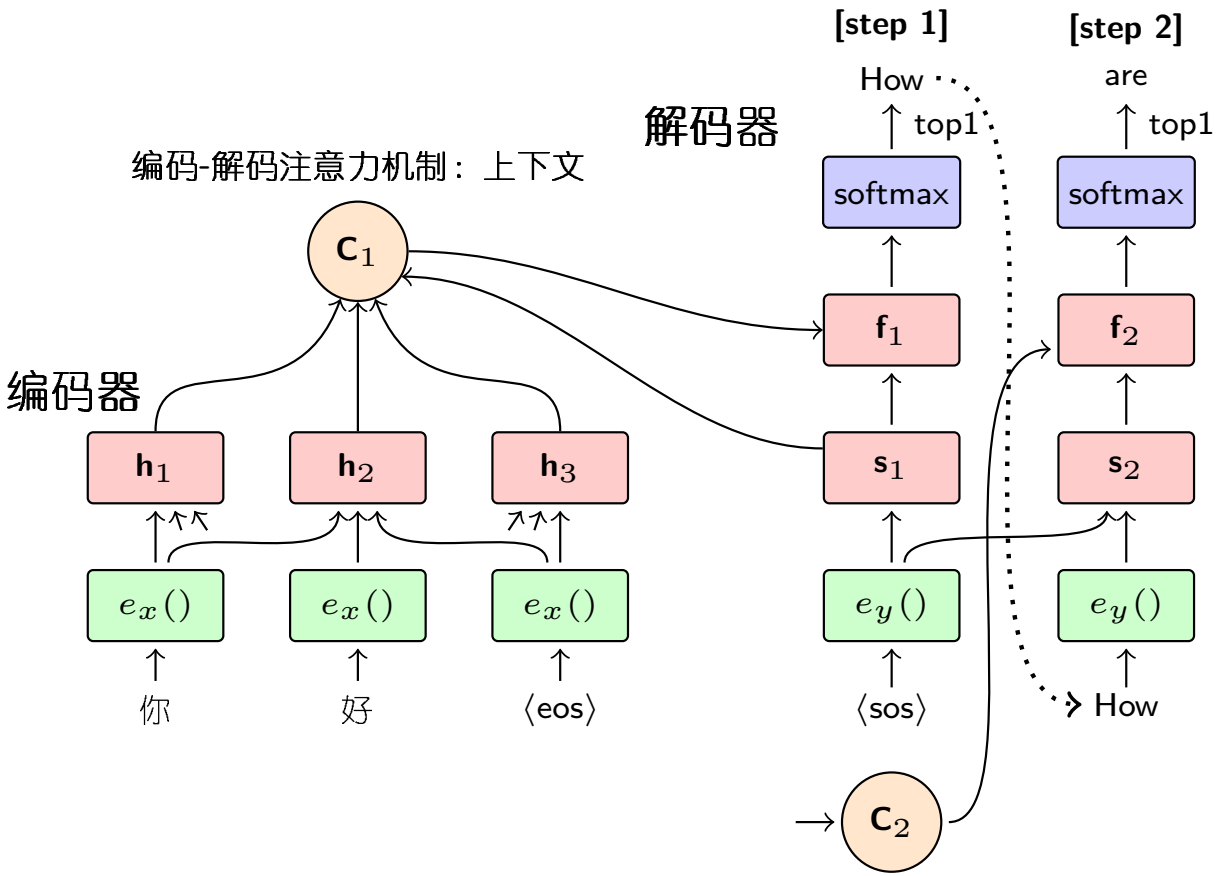
推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network



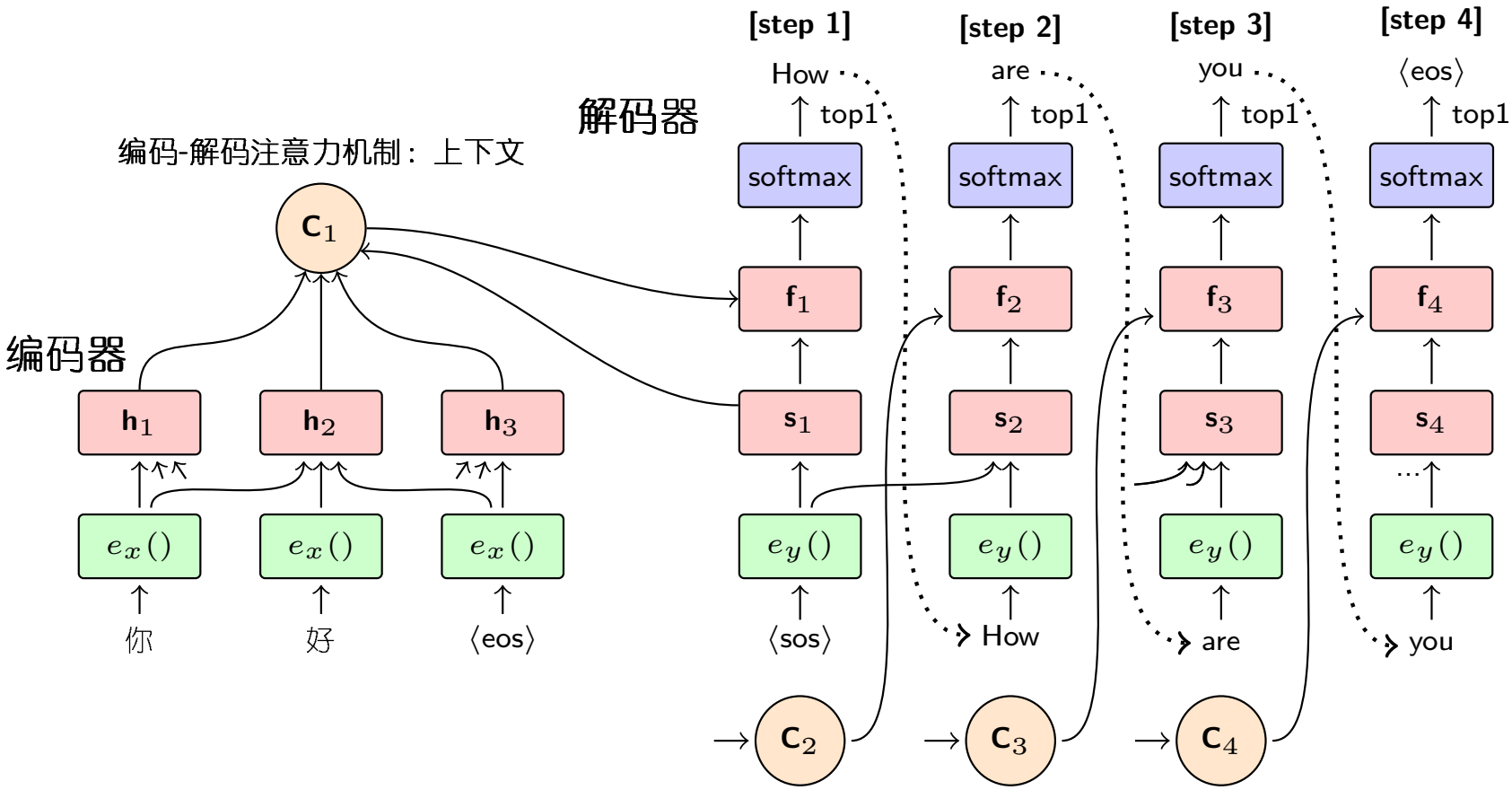
推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network



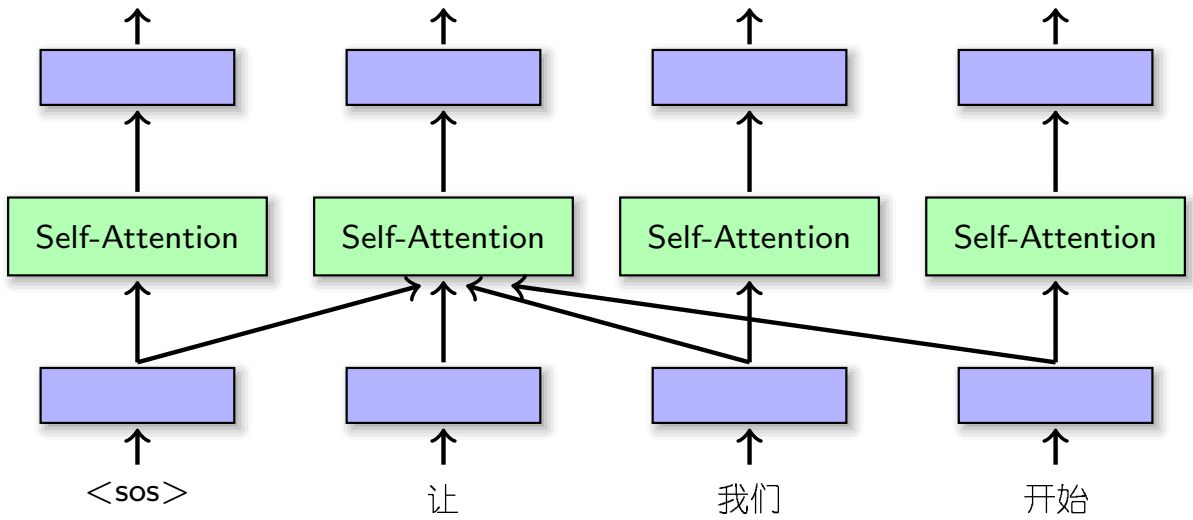
推断

- 由于自回归性，Transformer在推断阶段无法进行并行化操作，导致推断速度非常慢！
- 加速手段：低精度、Cache(缓存需要重复计算的变量)、Average Attention Network、Shared Attention Network



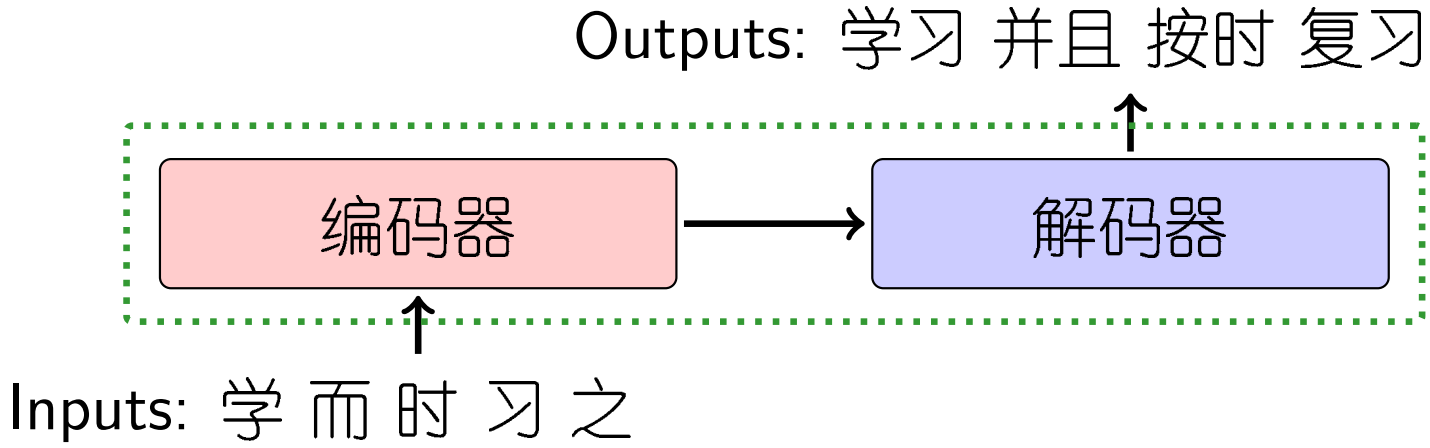
有趣的应用

一些有趣的神经机器翻译应用



NMT应用

- 神经机器翻译翻译系统除了满足日常翻译需求，还有很多其他有意思的应用！
  - 例如文言文翻译，将翻译系统中的源语、目标语，换成文言文和相应的现代文译文进行训练，就可以获得一个古文翻译系统



- 需要考虑的问题：
  - 古文短，现代文长，过翻译或者欠翻译对性能影响很大，如何对长度进行更精确的建模
  - 不同时代语言差异性大，如何进行自动适应和风格迁移

NMT应用

- 古文翻译实例

古文：侍卫步军都指挥使、彰信节度使李继勋营于寿州城南，唐刘仁赡伺继勋无备，出兵击之，杀士卒数百人，焚其攻具。

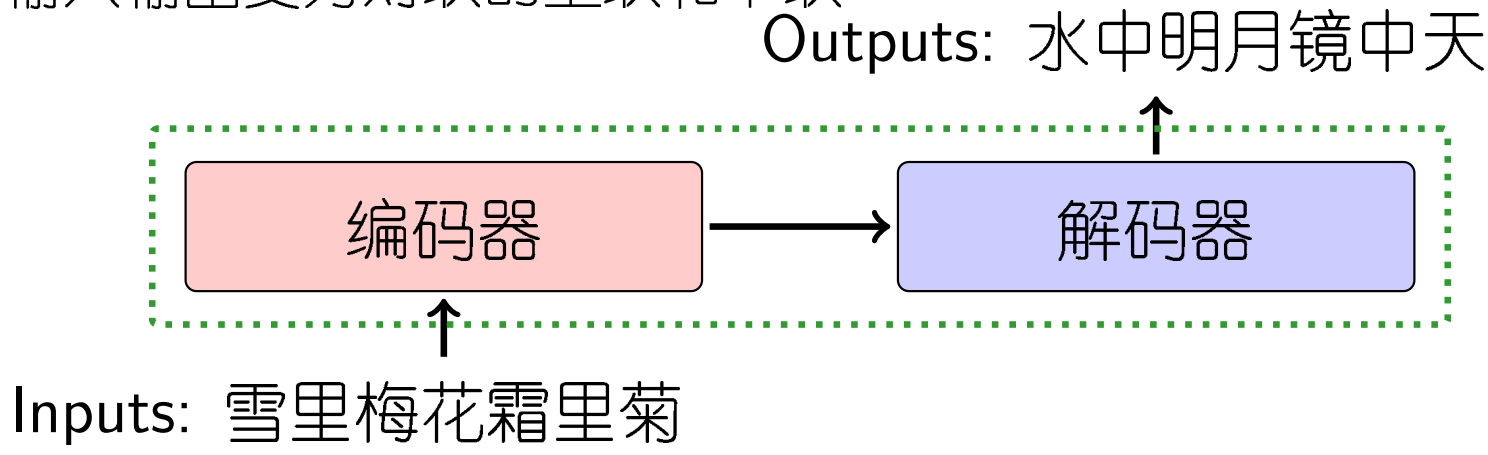
现代文：侍卫步军都指挥使、彰信节度使李继勋在寿州城南扎营，唐刘仁赡窥伺李继勋没有防备，出兵攻打他，杀死士兵几百人，烧毁李继勋的攻城器

古文：其后人稍稍识之，多延至其家，使为弟子论学。

现代文：后来的人渐渐认识他，多把他请到家里，让他为弟子讲授学问。

NMT应用

- 神经机器翻译翻译系统除了满足日常翻译需求，还有很多其他有意思的应用！
  - 除了古文翻译，对联也可以用机器翻译系统生成，只需将输入输出变为对联的上联和下联



- 需要考虑的问题：
  - 对联的上下联有较严格的对应要求，包括长度、押韵、词义的对应等
  - 横批生成难度比较大，横批是对内容的高度概括，但是数据非常缺乏

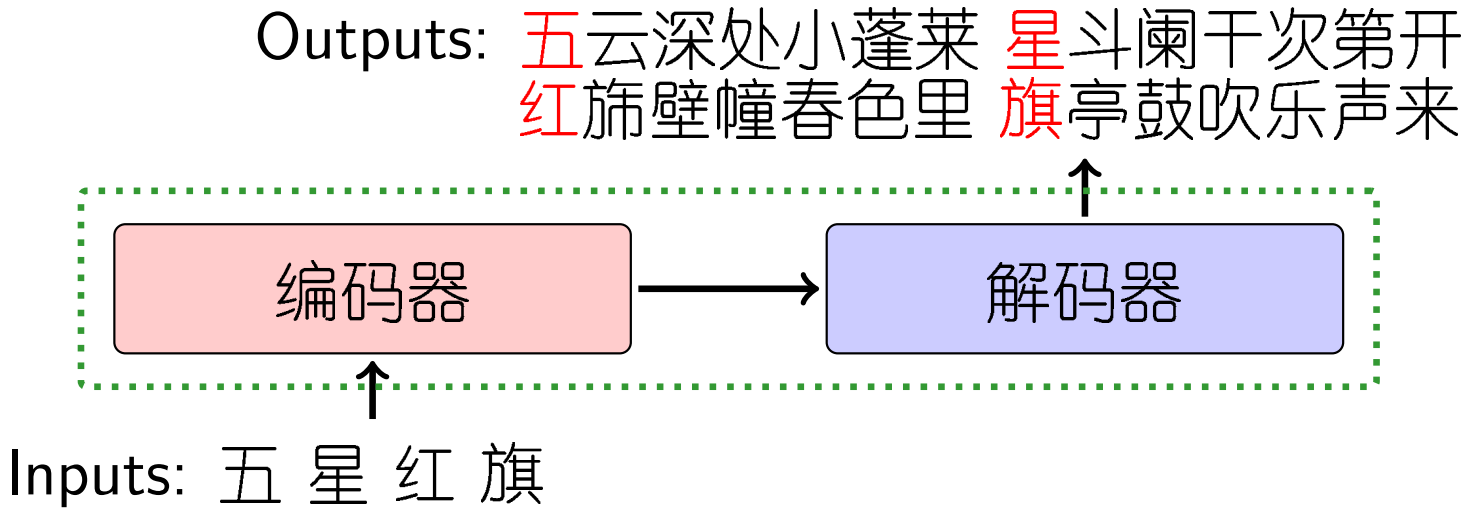
NMT应用

- 对联实例

上联：翠竹千支歌盛世	下联：红梅万点报新春
上联：一帆风顺年年好	下联：万事如意步步高
上联：佳节迎春春生笑脸	下联：新年纳福富华满堂
上联：腊梅吐芳迎红日	下联：绿柳展枝舞春风
上联：雪兆丰年从岭翠	下联：春回大地满园红

NMT应用

- 神经机器翻译翻译系统除了满足日常翻译需求，还有很多其他有意思的应用！
  - 还可以用机器翻译系统来写诗。如藏头诗，给定诗句的第一个字，生成一首完整的诗。还可以根据意境生成诗句



- 需要考虑的问题：
  - 古诗的的书写有对仗要求
  - 意境和字面背后的意思如何体现
  - 藏头诗需要有约束条件的生成

我们赶上了好时代 ...

- 神经机器翻译的火爆这几年有目共睹，好事情！！！
  - https://arxiv.org上搜索neural machine translation
  - ACL、EMNLP等顶会神经机器翻译论文数量近些年几乎呈线性增长
  - 神经机器翻译系统在各大比赛中霸榜，开源机器翻译满天飞，大厂秀肌肉，小作坊刷存在感

我们赶上了好时代 ...

- 神经机器翻译的火爆这几年有目共睹，好事情！！！
  - https://arxiv.org上搜索neural machine translation
  - ACL、EMNLP等顶会神经机器翻译论文数量近些年几乎呈线性增长
  - 神经机器翻译系统在各大比赛中霸榜，开源机器翻译满天飞，大厂秀肌肉，小作坊刷存在感
- 这里只介绍了最基本的概念，NMT的内容远不止这些
  - 各种专题：解码、压缩、先验知识、低资源翻译、无指导方法、篇章级翻译等等等等
  - 推荐一个survey，有些基础的可以参考一下，很全面
    "Neural Machine Translation: A Review" by Felix Stahlberg
    https://arxiv.org/abs/1912.02047
  - 如何搭建一个优秀的NMT系统？- 有许多技巧
    下一章介绍
  - 回忆一下第一章介绍的NMT开源系统，可以试试

一些开源NMT系统

- Tensor2Tensor
  - Google Brain开发，基于静态图实现
  - 先定义、后运行、速度快、可优化，但是代码中的错误难以发现
  - https://github.com/tensorflow/tensor2tensor
- Fairseq
  - Facebook开发，基于动态图实现
  - 灵活，debug方便，更适合自然语言处理
  - https://github.com/pytorch/fairseq
- NiuTrans.NMT
  - 小牛翻译开发，基于动态图实现
  - 简单小巧，易于修改、C语言编写，代码高度优化
  - https://github.com/NiuTrans/NiuTensor
- 其他优秀的开源NMT系统：OpenNMT、THUMT、Sockeye、Marian、Nematus、SGNMT、Neural Monkey...

结束