

# 语言模型与递归神经网络 (C19)

信息科学与技术学院

胡俊峰



# 回顾小结一下上次课的内容:

- 1、网络连接的本质是?
- 2、卷积算子的本质是?
- 3、为什么说卷积算子是共享参数的?




# 递归神经网络 (RNN)

- 序列编码问题的提出
- RNN模型基本架构
- RNN模型实现名字分类（例子）
- 递归神经网络实现序列模型编码



# 语言模型 到 序列编码

Language Model: 给定一个词语序列, 预测序列的概率 $P(w_1, \dots, w_m)$

一元语言模型:  $P(t_1 t_2 t_3) = P(t_1)P(t_2 | t_1)P(t_3 | t_1 t_2)$        $P_{\text{uni}}(t_1 t_2 t_3) = P(t_1)P(t_2)P(t_3).$   
  
序列的embedding?

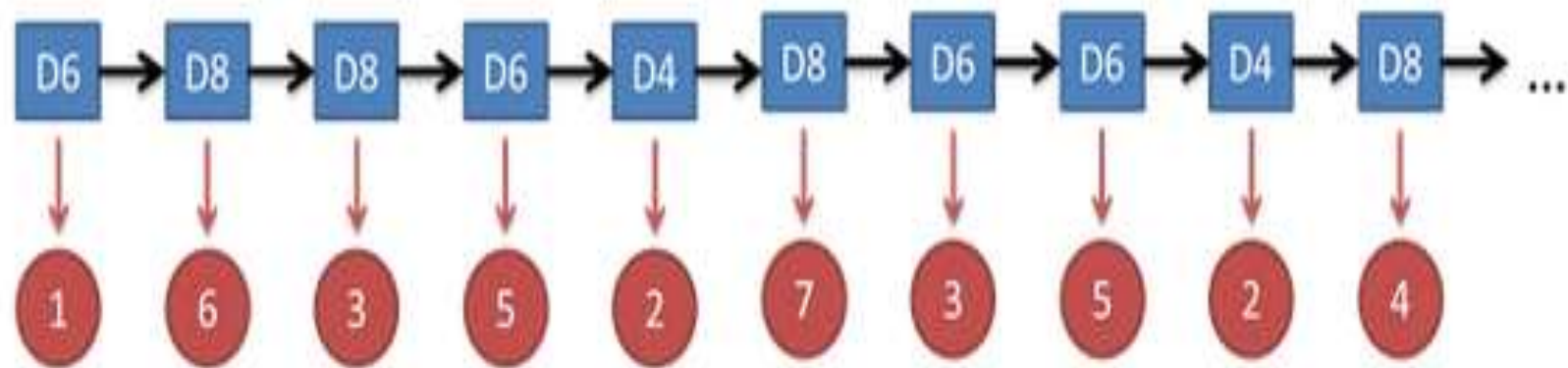
n-gram语言模型:  $P(w_1, \dots, w_m) = \prod_{i=1}^m P(w_i | w_1, \dots, w_{i-1}) \approx \prod_{i=1}^m P(w_i | w_{i-(n-1)}, \dots, w_{i-1})$

神经网络语言模型: 预测概率分布

- Feedforward NNLM
- Recurrent NNLM

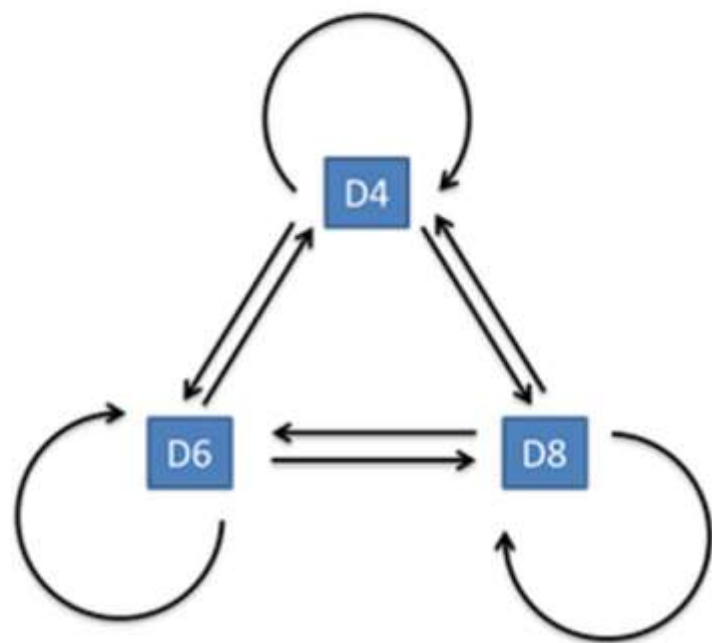
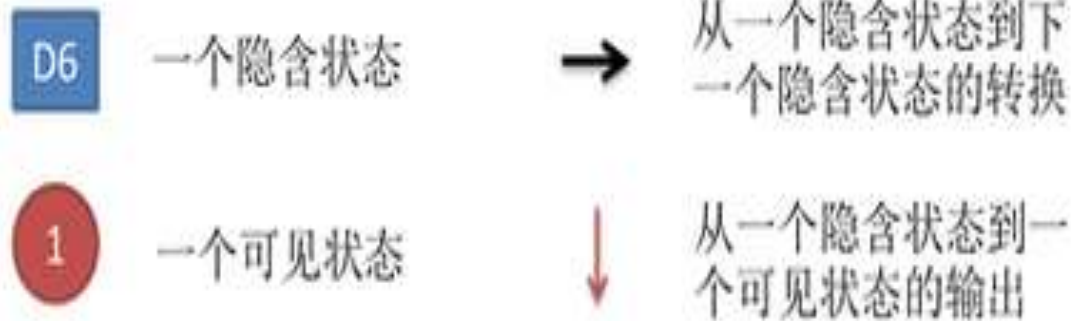


隐马尔可夫模型示意图



隐含状态转换关系示意图

图例说明:



# HMM最基础的应用 —— 序列生成

- 已知隐状态的初始概率； 转换概率矩阵； 发射概率矩阵
- 生成序列
- 流程： 随机一个初始状态
- 迭代转移： 转移矩阵相乘， 发射矩阵相乘（得到发射概率分布）， 采样输出一个out



# HMM的三种问题

- 1) 知道隐含状态数量，以及状态间的转换概率矩阵 $A$ ，隐状态对结果的发射概率矩阵 $B$ ，观察到结果状态链，问系统生成这个结果的概率（评价）。
- 2) 知道隐含状态数量，以及状态间的转换概率矩阵 $A$ ，隐状态对结果的发射概率矩阵 $B$ ，观察到结果状态链，想知道最大概率的隐含状态链
  - 如：观察云层情况，预测天气变化。
  - 观察词序列，预测词性序列（词性标注）
  - 观察字序列，预测分词标记。
- 3) 知道隐含状态数量，以及每个隐状态对结果的发射概率矩阵，观察到很多结果状态链，想反推出每个状态之间的转换概率矩阵



# HMM假设

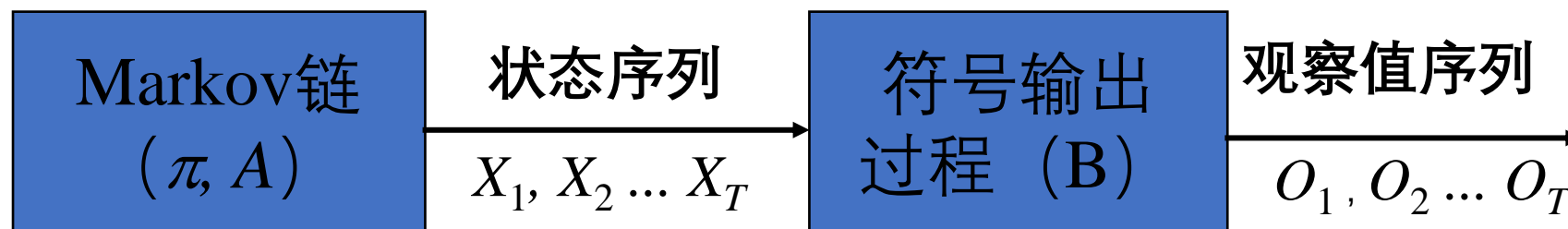
- 一个随机过程，有一个观察序列  $O=O_1, O_2 \dots O_T$ ，该过程隐含着一个状态序列  $X=X_1, X_2 \dots X_T$
- 假设
  - Markov假设
    - 假设1：有限历史假设：  $P(X_i/X_1, X_2 \dots X_{i-1}) = P(X_i/X_{i-1})$
    - 假设2：时间不动性假设
  - 输出条件独立性假设
    - 输出仅与当前状态有关
    - $P(O_1, O_2 \dots O_T / X_1, X_2 \dots X_T) = \prod_t P(O_t / X_t)$





# HMM模型 - 图示

- 两个随机过程



HMM的组成示意图

# HMM模型的符号表示

- 模型表示

- 五元组( $S, V, \pi, A, B$ )

- 符号表

- $S$ : 状态集合,  $\{s_1, \dots, s_N\}$ 。
      - $V$ : 输出字母表,  $\{v_1, \dots, v_M\}$

- 模型参数

- $\pi$ : 初始状态概率。  $\pi = \{\pi_i\}; \quad i \in S$
      - $A$ : 状态转移概率。  $A = \{a_{ij}\}; \quad i, j \in S$
      - $B$ : 符号输出概率。  $B = \{b_{jk}\}; \quad j \in S, k \in V$

- 序列

- 状态序列:  $X = X_1, X_2 \dots X_T \quad X_t \in S$
    - 输出序列:  $O = O_1, O_2 \dots O_T \quad O_t \in V$



# HMM过程描述

$t = 1;$

初始状态概率分布为 $\pi$ 。从状态 $s_i$ 开始的概率为 $\pi_i$ ;

Forever do

从状态 $s_i$  向状态 $s_j$ 转移,并输出观察符号 $O_t = k$ 。

其中, 状态转移概率为 $a_{ij}$ 。符号输出概率为  $b_{jk}$

$t = t+1$

End



# HMM模型 - 图示

状态序列

$X_1$

$X_2$

$X_{T-1}$

$X_{T-1}$

状态空间

$s_{1,1}$

$s_{1,2}$

$s_{1,3}$

$s_{1,4}$

$s_{2,1}$

$s_{2,2}$

$s_{2,3}$

$s_{2,4}$

$s_{T-1,1}$

$s_{T-1,2}$

$s_{T-1,3}$

$s_{T-1,4}$

$s_{T,1}$

$s_{T,2}$

$s_{T,3}$

$s_{T,4}$

观察序列

$O_1$

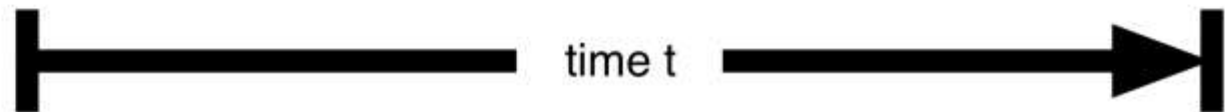
$O_2$

.....

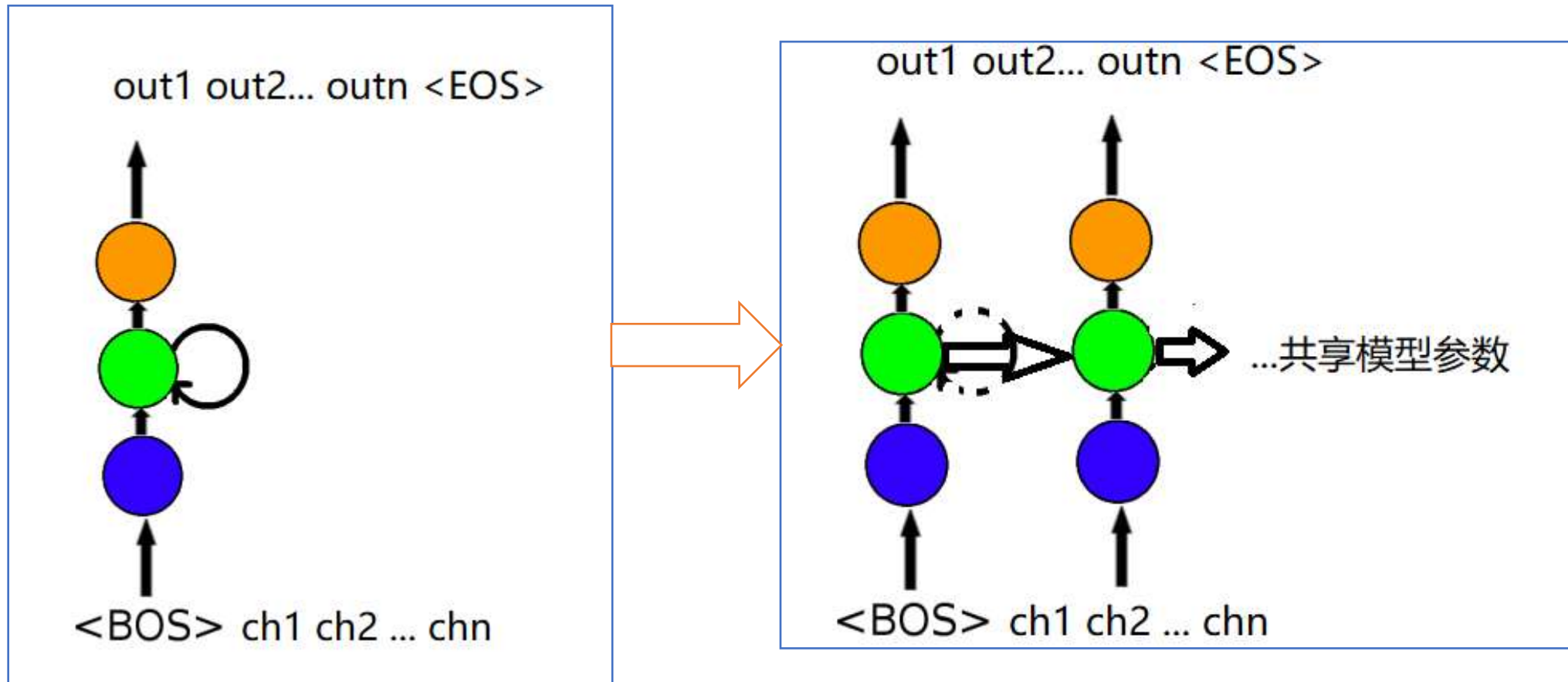
$O_{T-1}$

$O_T$

时间



RNN基本模型： $f_i(f_{i-1}, \text{input}_i)$  (f的输出为隐状态空间向量)



# RNN模型 VS k阶序列回归模型

- 定长区域输入，全局方案回归
- 累进输入，全局记忆回归

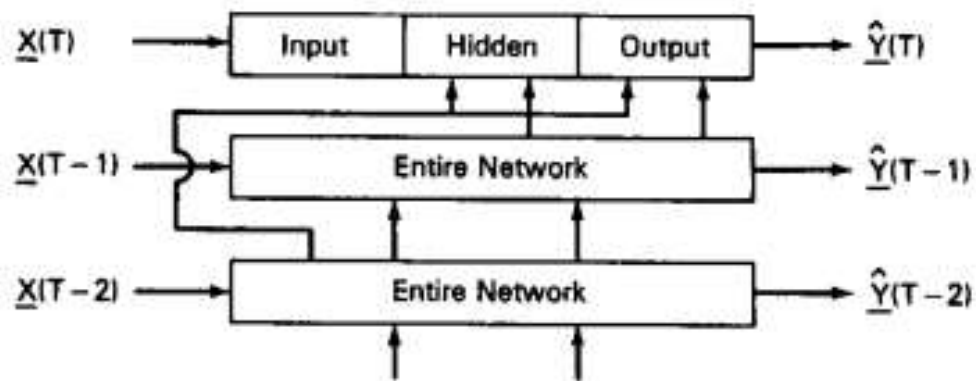


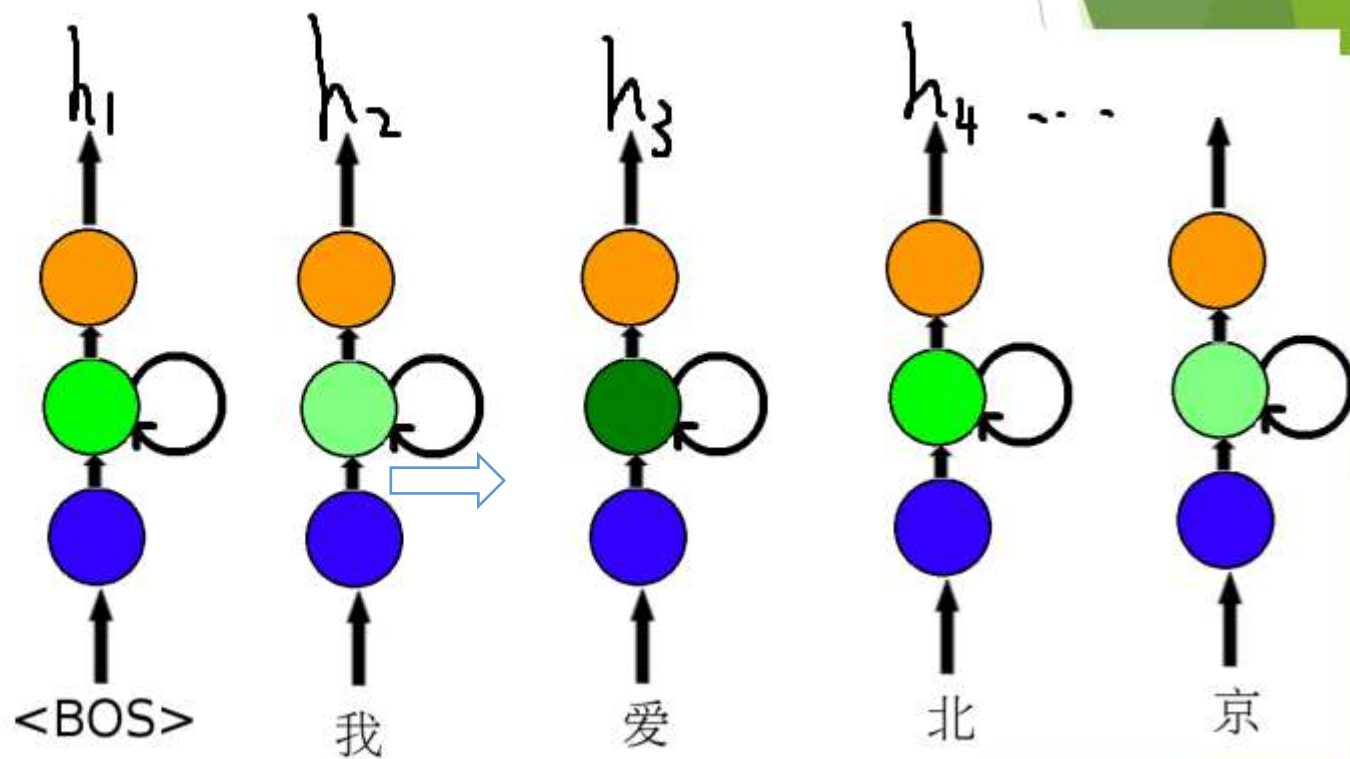
Fig. 5. Generalized network design with time lags.

序列映射可以对照HMM中最优隐状态序列来理解  
训练过程可以参照回归模型来理解

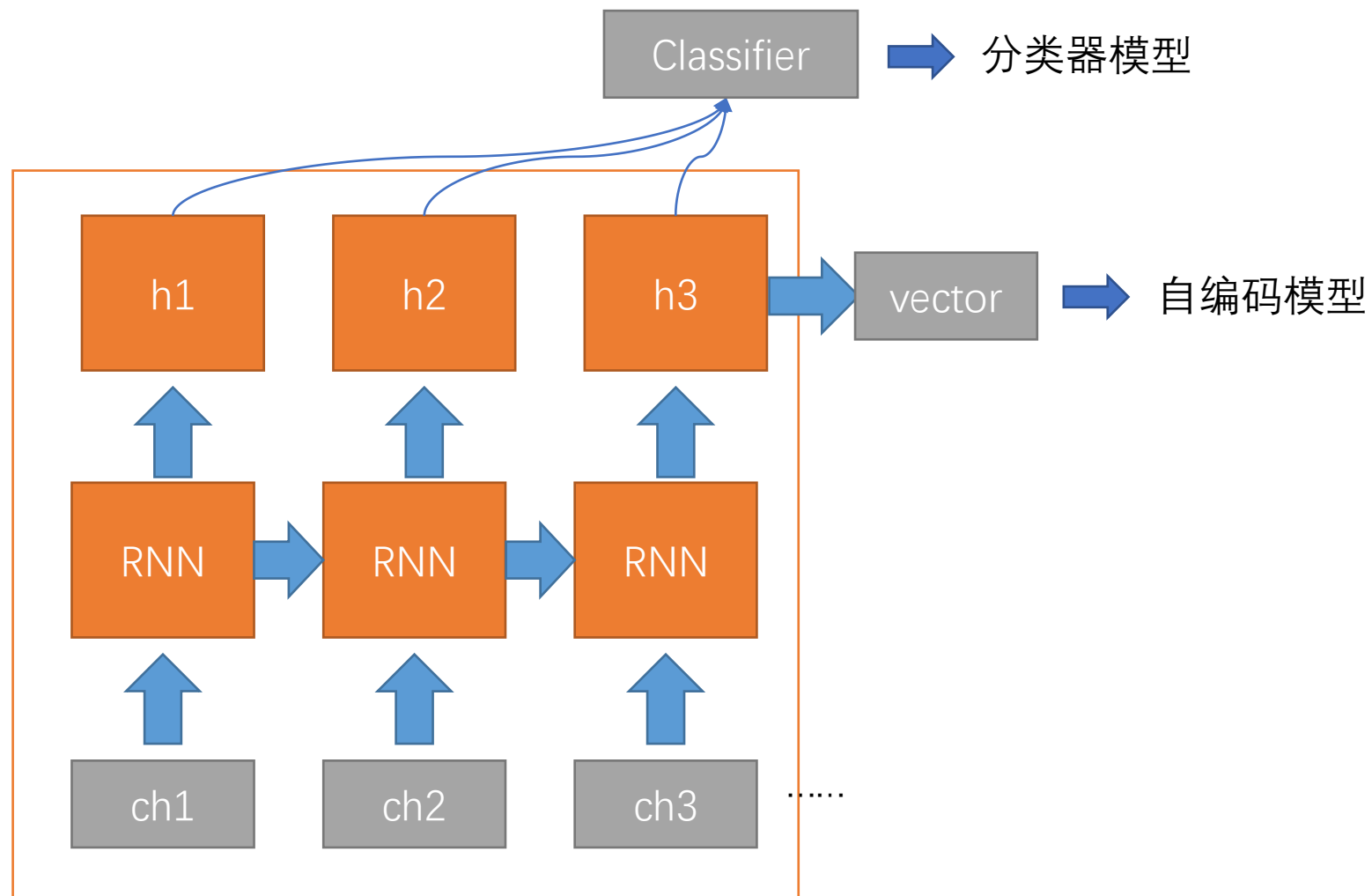
## ► Recurrent Neural Network 循环神经网络

### ► 用法

#### ► 序列映射



# RNN序列编码问题建模方案：





# 例子：RNN实现名字分类：训练数据集

- 18个文件（国家），英文拼写模式的姓名，训练集不平衡

Arabic	2017/3/12 14:46	文本文档	13 KB
Chinese	2017/3/12 14:46	文本文档	2 KB
Czech	2017/3/12 14:46	文本文档	4 KB
Dutch	2017/3/12 14:46	文本文档	3 KB
English	2017/3/12 14:46	文本文档	27 KB
French	2017/3/12 14:46	文本文档	3 KB
German	2017/3/12 14:46	文本文档	6 KB
Greek	2017/3/12 14:46	文本文档	2 KB
Irish	2017/3/12 14:46	文本文档	2 KB
Italian	2017/3/12 14:46	文本文档	6 KB
Japanese	2017/3/12 14:46	文本文档	8 KB
Korean	2017/3/12 14:46	文本文档	1 KB
Polish	2017/3/12 14:46	文本文档	2 KB
Portuguese	2017/3/12 14:46	文本文档	1 KB
Russian	2017/3/12 14:46	文本文档	84 KB
Scottish	2017/3/12 14:46	文本文档	1 KB
Spanish	2017/3/12 14:46	文本文档	3 KB
Vietnamese	2017/3/12 14:46	文本文档	1 KB



# RNN在pytorch中实现方案:

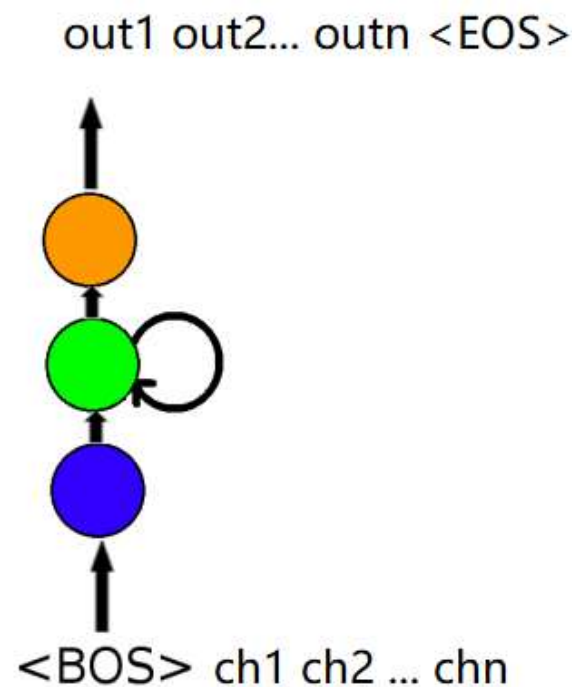
- 定义网络结构
- 设置样本集 (分类问题)
- 采用序列循环模式进行训练

```
rnn.zero_grad() # 每行清零一次

# 序列循环模式。
# 这里中间的output没有被利用
for i in range(line_tensor.size()[0]): # 循环所有字符
    output, hidden = rnn(line_tensor[i], hidden) # hidden被递归输入

loss = criterion(output, category_tensor) # 计算loss
loss.backward() # 反向梯度计算

# Add parameters' gradients to their values, multiplied by learning rate
for p in rnn.parameters():
    p.data.add_(p.grad.data, alpha=-learning_rate) # 梯度下降更新所有网络参数
```



## 模型定义： 模型基本架构

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size # 隐状态空间维度

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size) #  $f_i(f_{i-1}, input_i)$ 
        self.i2o = nn.Linear(input_size + hidden_size, output_size) #  $g_i(f_{i-1}, input_i)$ 
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1) # 拼接input, hidden 57+128
        hidden = self.i2h(combined) # to hidden
        output = self.i2o(combined)
        output = self.softmax(output) # 非线性激活函数 -> 分类器
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

## 模型输入输出数据方案：状态空间定义

```
def findFiles(path): return glob.glob(path)

# #abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ .,:;'
all_letters = string.ascii_letters + " .,:;"
n_letters = len(all_letters) # 57

# Turn a Unicode string to plain ASCII(减小输入状态空间维数)
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# print(unicodeToAscii('Ślusàrski'))

category_lines = {} # Build the category_lines dictionary,
all_categories = [] # a list of names per language
```

标准化s，如果所有字符都非标记字符，  
如果是预定的letters集合中，转为ASCII



## 读入数据-生成训练数据集

```
category_lines = {}    # Build the category_lines dictionary,
all_categories = []    # a list of names per language

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines] # 返回一个按行的文件内容的列表

for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category) # 取文件名作为类别名
    lines = readLines(filename)
    category_lines[category] = lines # 类名-按行内容的列表

n_categories = len(all_categories)
print(all_categories )
```

57

```
['Arabic', 'Chinese', 'Czech', 'Dutch', 'English', 'French', 'German', 'Greek', 'Irish', 'Italian', 'Japanese', 'Korean', 'Polish', 'Portuguese', 'Russian', 'Scottish', 'Spanish', 'Vietnamese']
```

```
print(category_lines['Chinese'][:9])
```

```
['Ang', 'AuYong', 'Bai', 'Ban', 'Bao', 'Bei', 'Bian', 'Bui', 'Cai']
```

## 名字（字符）序列 → 输入向量

```
# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# Just for demonstration, turn a letter into a <1 x n_letters> Tensor
def letterToTensor(letter):
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1 # 转成one hot输入向量
    return tensor

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters) # 初始化全零
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1 # 字符对应位置设1
    return tensor

print(letterToTensor('J'))

print(lineToTensor('Jones').size())
```

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0.]])
```



# 模型预测：输入字符串-输出类的对数概率

```
input = lineToTensor('Albert')
hidden = torch.zeros(1, n_hidden) # 隐状态初始设为0

output, next_hidden = rnn(input[0], hidden) # 第一轮forward结果
print(output)
```

```
tensor([[ -2.9699, -2.8553, -2.7270, -2.9729, -2.5957, -2.7799, -2.5413, -3.8779,
          -2.9261, -3.4373, -3.3391, -2.2084, -3.1340, -3.4764, -2.9974, -2.9137,
          -3.1045, -2.4782]], grad_fn=<LogSoftmaxBackward>)
```

$$= \log(e^{(x_i - M)}) - \log\left(\sum_j^n e^{(x_j - M)}\right) = (x_i - M) - \log\left(\sum_j^n e^{(x_j - M)}\right)$$

```
def categoryFromOutput(output):
    top_n, top_i = output.topk(1) # 返回output的topk 及下标
    category_i = top_i[0].item() # top1 下标
    return all_categories[category_i], category_i

print(categoryFromOutput(output))
```

<https://www.zhihu.com/question/358069078>

('Korean', 11)

# 训练模块定义

```
learning_rate = 0.005 # If you set this too high, it might explode. If too low, it might not learn

def train(category_tensor, line_tensor):

    hidden = rnn.initHidden()

    rnn.zero_grad() # 每行清零一次

    # 序列循环模式。
    # 这里中间的output没有被利用
    for i in range(line_tensor.size()[0]): # 循环所有字符
        output, hidden = rnn(line_tensor[i], hidden) # hidden被递归输入

    loss = criterion(output, category_tensor) # 计算loss
    loss.backward() # 反向梯度计算

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate) # 梯度下降更新所有网络参数

    return output, loss.item()
```



# 模型参数：

```
: paras = list(rnn.parameters())  
for num, para in enumerate(paras):  
    print('para:', num)  
    print(para.shape)
```

```
para: 0  
torch.Size([128, 185])  
para: 1  
torch.Size([128])  
para: 2  
torch.Size([18, 185])  
para: 3  
torch.Size([18])
```



# 训练参数设定:

```
import time
import math

n_iters = 100000    # 10万轮
print_every = 5000  # 5000次check一下
plot_every = 1000

# Keep track of losses for plotting
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)    # 下取整, 转成分钟
    s -= m * 60
    return '%dm %ds' % (m, s)    # **分**秒

start = time.time()
```



# 模型训练

```
for iter in range(1, n_iters + 1): # 每行一轮
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss # 每1000轮loss累加

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else 'X (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100, timeSince(start), loss, line,
                                                guess, guess_i, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every) # 最近1000次 loss平均, append
        current_loss = 0                             # 清零
```

```
5000 5% (0m 6s) 2.8874 Farina / Japanese X (Italian)
10000 10% (0m 11s) 0.7383 Bekoryukov / Russian ✓
15000 15% (0m 18s) 1.4846 Thai / Chinese X (Vietnamese)
20000 20% (0m 23s) 2.6085 Cerny / German X (Czech)
25000 25% (0m 29s) 0.6633 Pefanis / Greek ✓
```

## 模型定义： 模型基本架构

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size # 隐状态空间维度

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size) #  $f_i(f_{i-1}, input_i)$ 
        self.i2o = nn.Linear(input_size + hidden_size, output_size) #  $g_i(f_{i-1}, input_i)$ 
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1) # 拼接input, hidden 57+128
        hidden = self.i2h(combined) # to hidden
        output = self.i2o(combined)
        output = self.softmax(output) # 非线性激活函数 -> 分类器
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

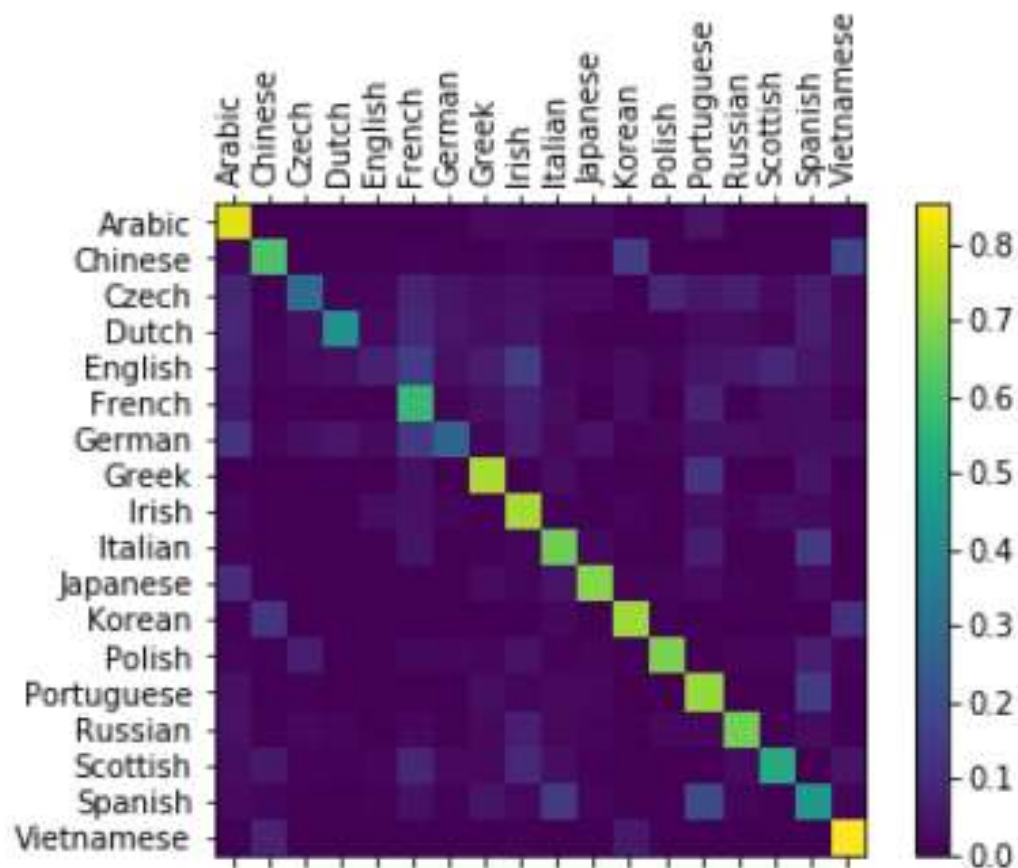
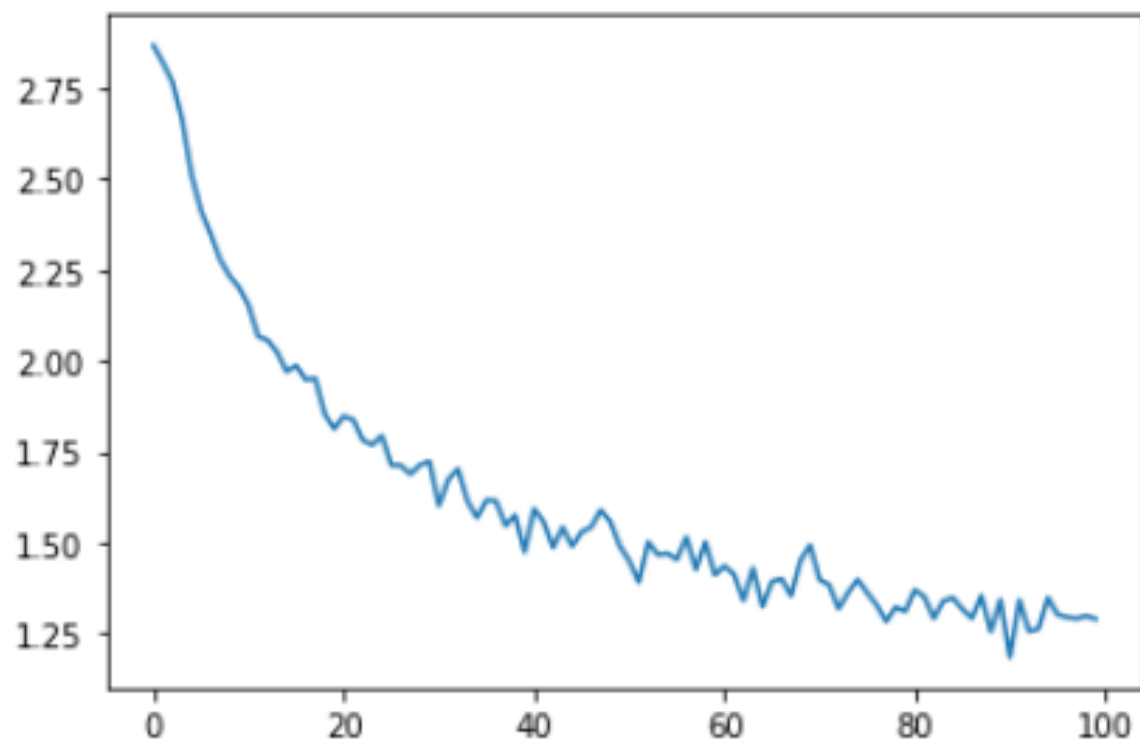
85000 85% (1m 40s) 0.5301 Ku / Korean ✓  
90000 90% (1m 45s) 0.6712 Sokoloff / Polish ✓  
95000 95% (1m 51s) 1.0642 Pochitalin / Russian ✓  
100000 100% (1m 57s) 0.6546 Riagain / Irish ✓

## 训练结果观察

```
import matplotlib.pyplot as plt  
import matplotlib.ticker as ticker
```

```
plt.figure()  
plt.plot(all_losses)
```

[<matplotlib.lines.Line2D at 0x2035b6bad88>]



# 小结一下：

- 序列编码的每一个输出都是前驱序列的编码结果
- 序列编码本质上是一个时序过程，理论上应该有滑窗限制
  - 在语言类问题的处理中常常借用语言的自然边界，如标点
- RNN序列编码是一个1阶模型仿真n阶模型，必然弱于真n阶模型的编码能力

Torchtext的安装问题：要与pytorch版本匹配。



# 序列编码与生成 (Seq2seq) 模型

- 问题提出
- 模型基本思想与框架
- 模型实现
- Attention机制



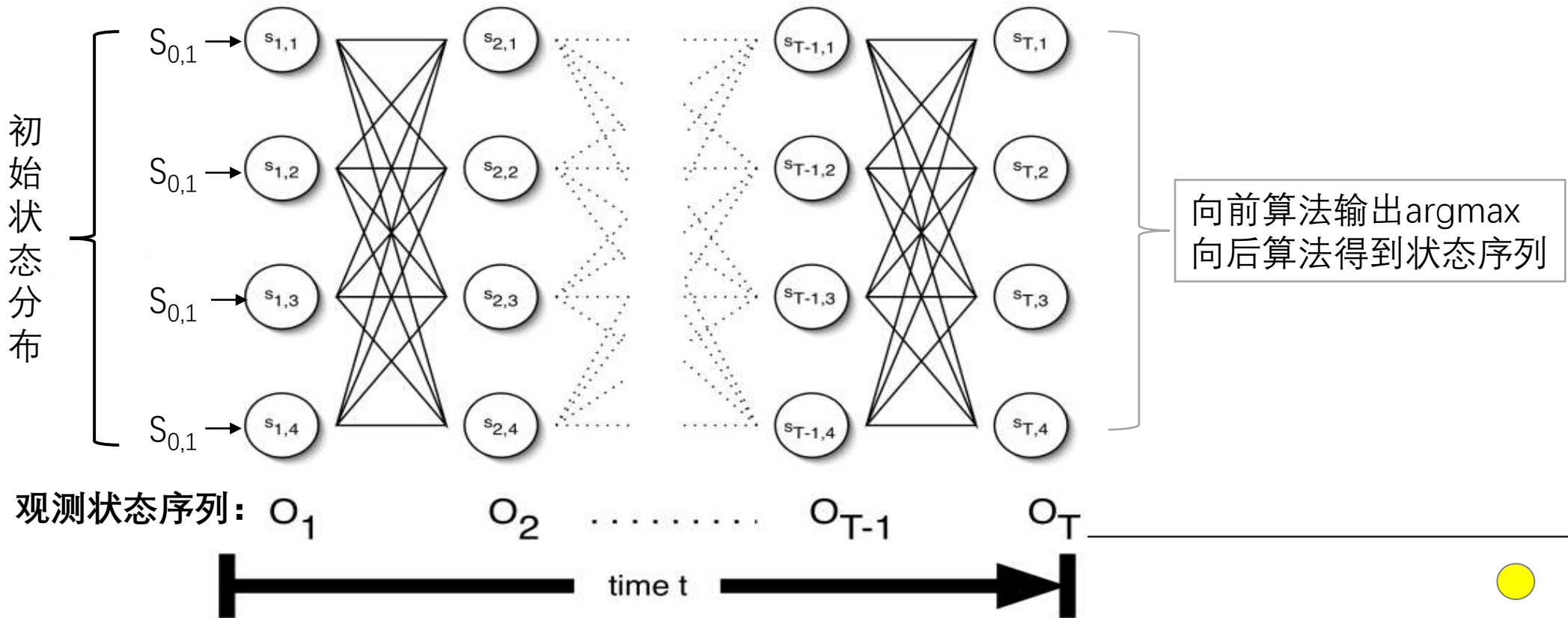
# HMM: 已知模型参数和观察序列, 预测隐状态结果

状态转移矩阵

↓

隐状态解码算法中的向前算法（求最大值）： $p_{t+1}(j) = \max_{1 \leq i \leq n} [p_t(i) a_{ij}] b_{j, o_{t+1}}$

← 发射概率矩阵

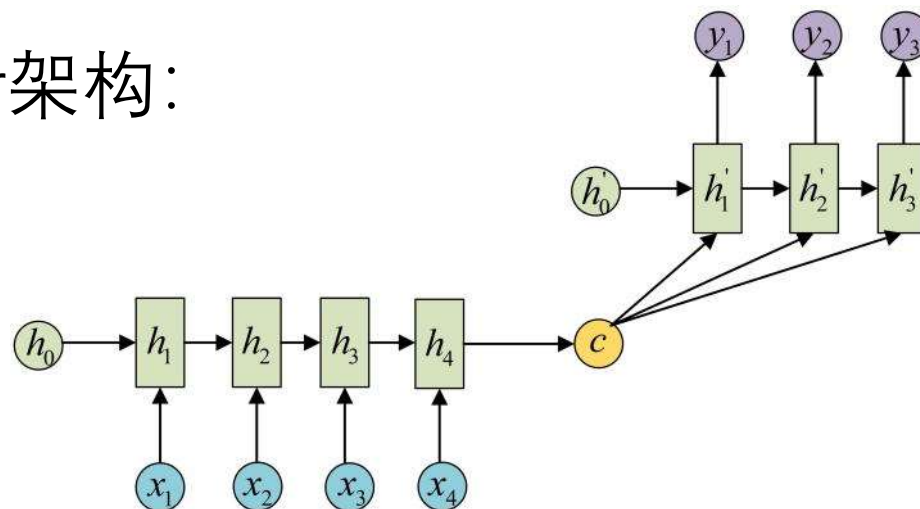




# Encoder-Decoder架构: 处理Seq2Seq任务

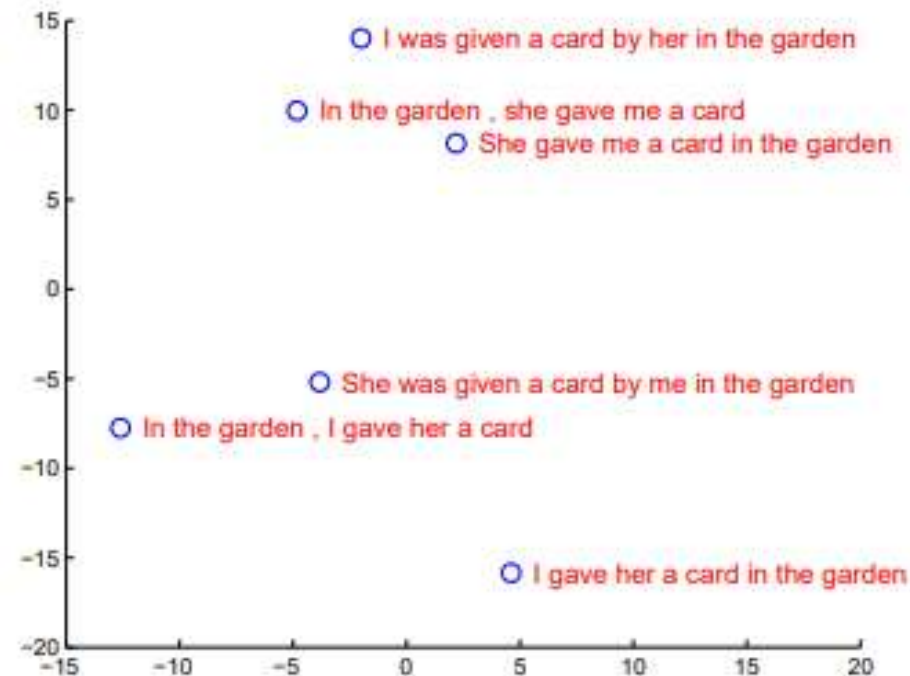
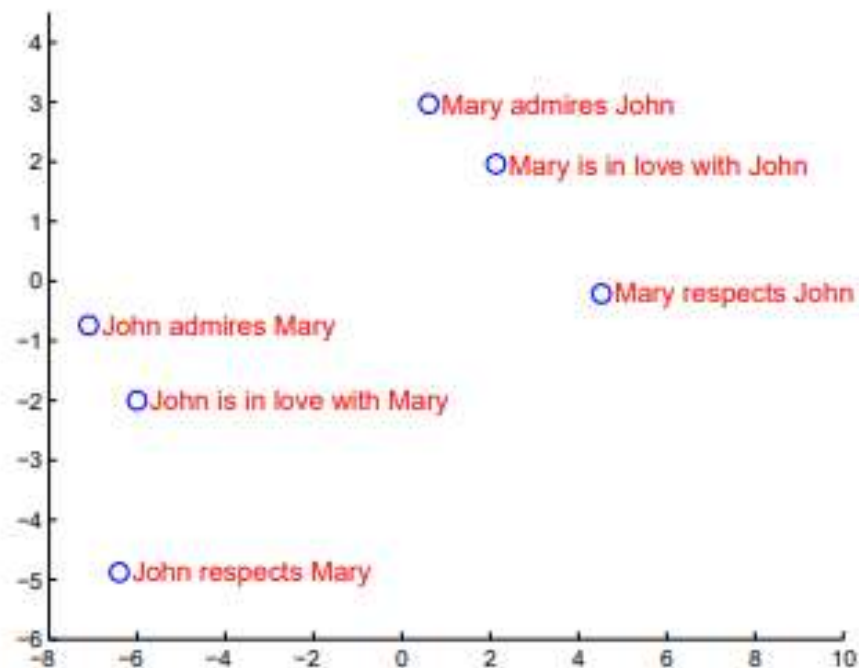
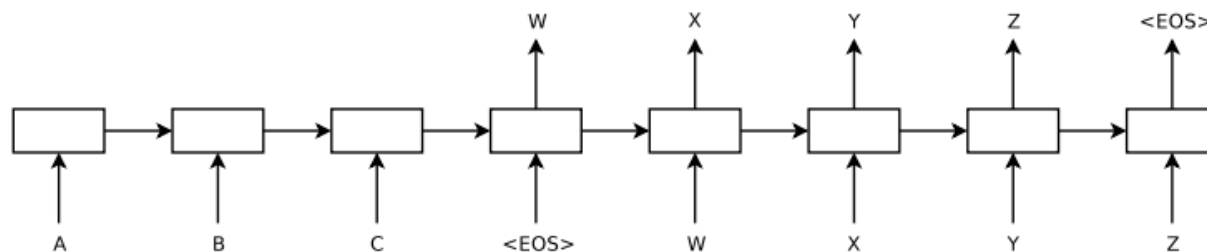
- RNN, LSTM: 上世纪80 ~ 90年代被提出, 2014年因Encoder-Decoder模型架构的提出而大火
  - Bengio论文: <https://arxiv.org/pdf/1406.1078.pdf>
  - 解决了一大批语音 $\leftrightarrow$ 文本、机器翻译等任务, 催生出了许多相关应用

- Encoder-Decoder架构:



# 句子的向量表示与生成 seq2seq

## Sequence to Sequence Learning with Neural Networks

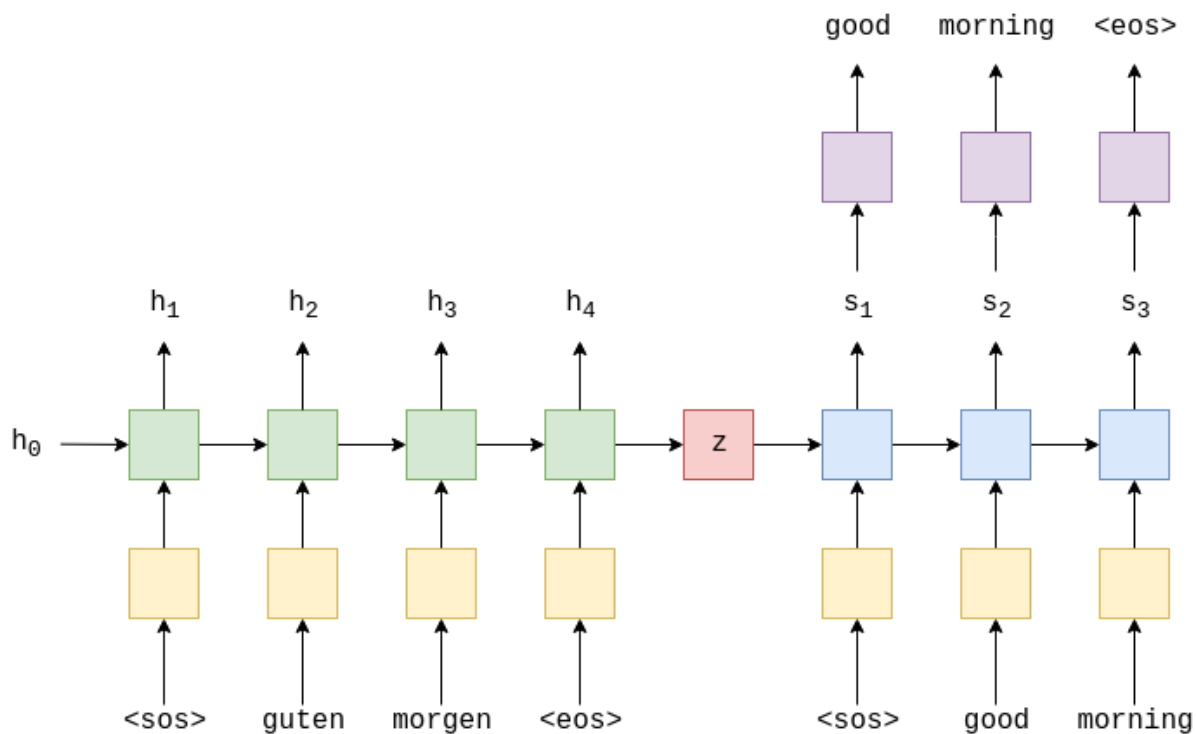


# 模型基本框架

$$h_t = \text{EncoderRNN}(e(x_t), h_{t-1})$$

$$s_t = \text{DecoderRNN}(d(y_t), s_{t-1})$$

$$\hat{y}_t = f(s_t)$$



# LSTM:

## 简介

### 通用LSTM的结构

$$\bar{z}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z$$

$$\mathbf{z}^t = g(\bar{z}^t)$$

$$\bar{\mathbf{i}}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i$$

$$\mathbf{i}^t = \sigma(\bar{\mathbf{i}}^t)$$

$$\bar{\mathbf{f}}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f$$

$$\mathbf{f}^t = \sigma(\bar{\mathbf{f}}^t)$$

$$\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$$

$$\bar{\mathbf{o}}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o$$

$$\mathbf{o}^t = \sigma(\bar{\mathbf{o}}^t)$$

$$\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t$$

block input

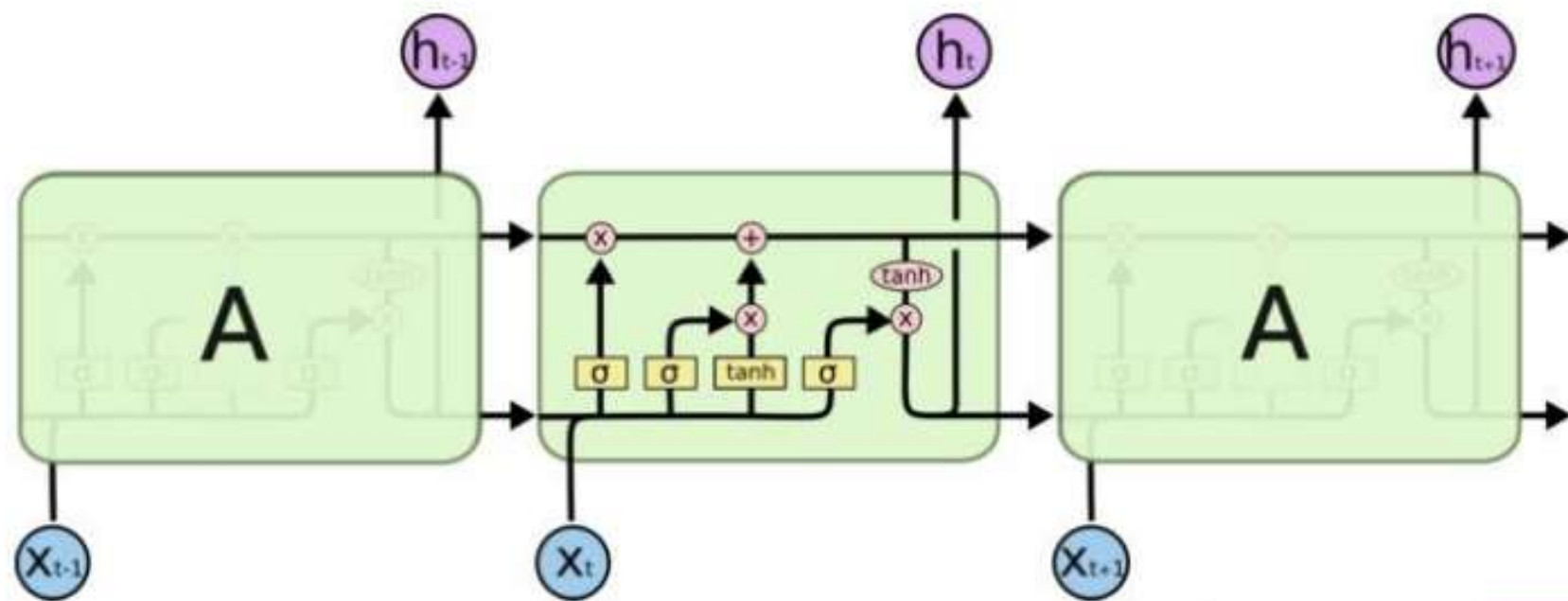
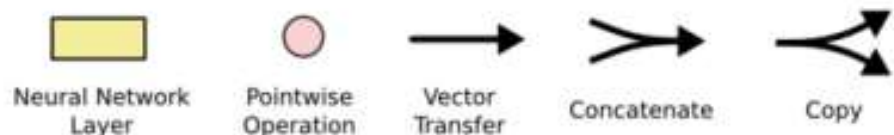
input gate

forget gate

cell

output gate

block output



0

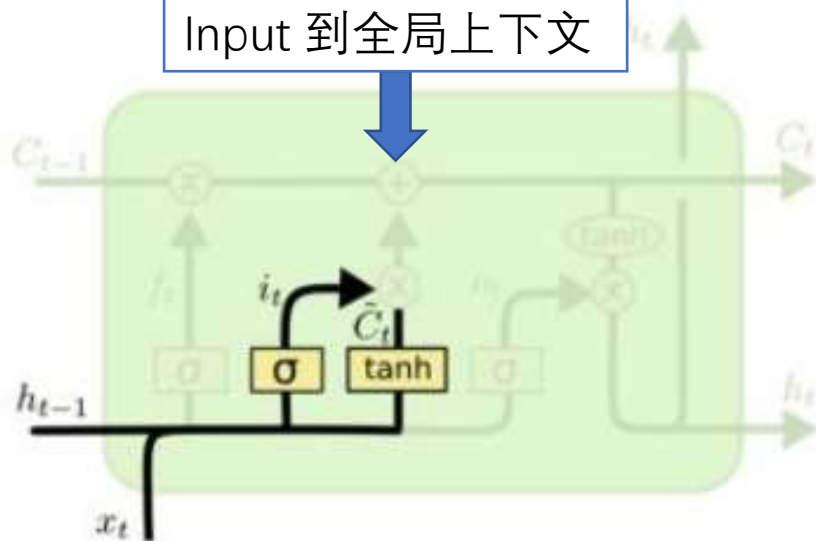
## 简介

### ► LSTM结构

#### ► Block input

- 代表输入信息
- 结合上一时间的输出和当前输入
- RNN中的原始结构

Input 到全局上下文



常规RNN

$$\bar{z}^t = W_z x^t + R_z y^{t-1}$$

$$z^t = g(\bar{z}^t)$$

*block input*

$$\bar{i}^t = W_i x^t + R_i y^{t-1}$$

$$i^t = \sigma(\bar{i}^t)$$

*input gate*

$$\bar{f}^t = W_f x^t + R_f y^{t-1}$$

$$f^t = \sigma(\bar{f}^t)$$

*forget gate*

$$c^t = z^t \odot i^t + c^{t-1} \odot f^t$$

*cell*

$$\bar{o}^t = W_o x^t + R_o y^{t-1}$$

$$o^t = \sigma(\bar{o}^t)$$

*output gate*

$$y^t = h(c^t) \odot o^t$$

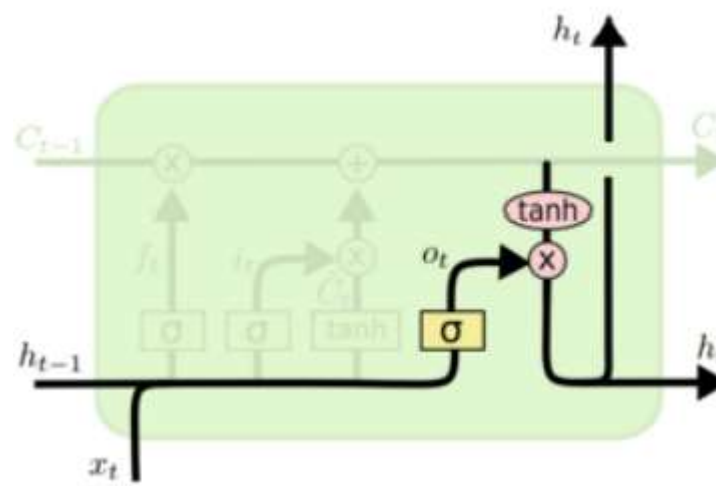
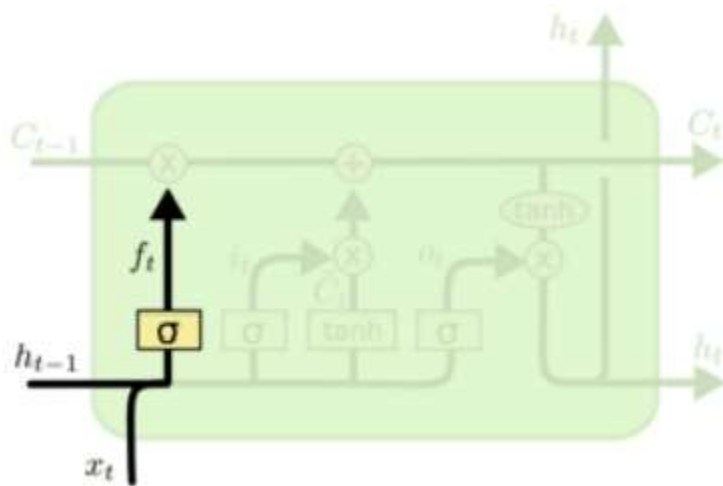
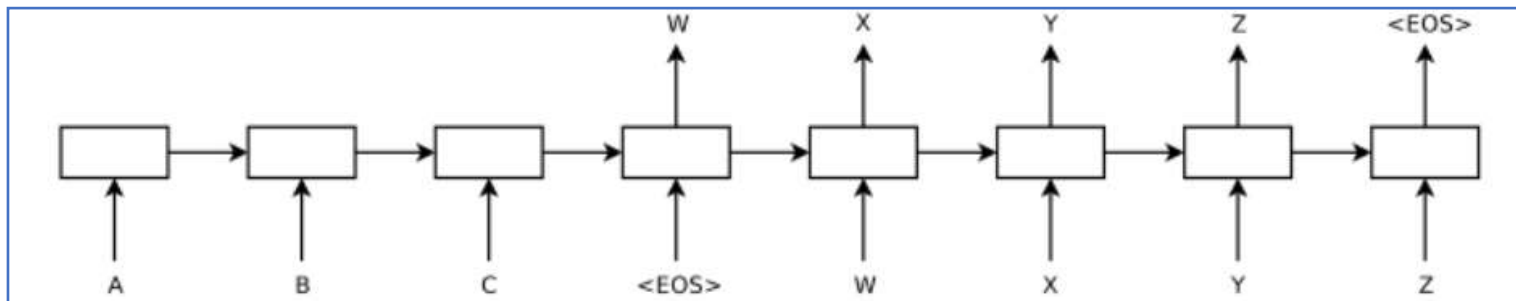
*block output*

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t])$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t])$$

# 加入遗忘机制以及全局上下文的RNN架构

- 隐状态 $h_i$ 代表了到时间 $t_i$ 为止的序列embedding



$$\bar{z}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1}$$

$$\mathbf{z}^t = g(\bar{z}^t)$$

$$\bar{\mathbf{i}}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1}$$

$$\mathbf{i}^t = \sigma(\bar{\mathbf{i}}^t)$$

$$\bar{\mathbf{f}}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1}$$

$$\mathbf{f}^t = \sigma(\bar{\mathbf{f}}^t)$$

$$\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$$

$$\bar{\mathbf{o}}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1}$$

$$\mathbf{o}^t = \sigma(\bar{\mathbf{o}}^t)$$

$$\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t$$

block input

input gate

forget gate

cell

output gate

block output

Forget gate

output gate



# LSTM编码过程： 数据准备：

```
def tokenize_de(text):  
    """  
    Tokenizes German text from a string into a list of strings (tokens) and reverses it  
    """  
    return [tok.text for tok in spacy_de.tokenizer(text)][::-1]  
  
def tokenize_en(text):  
    """  
    Tokenizes English text from a string into a list of strings (tokens)  
    """  
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

```
SRC = Field(tokenize = tokenize_de,  
            init_token = '<sos>',  
            eos_token = '<eos>',  
            lower = True)
```

```
TRG = Field(tokenize = tokenize_en,  
            init_token = '<sos>',  
            eos_token = '<eos>',  
            lower = True)
```

```
SRC.build_vocab(train_data, min_freq = 2)  
TRG.build_vocab(train_data, min_freq = 2)
```



# LSTM编码过程:

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(input_dim, emb_dim) # token embedding

        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)

        self.dropout = nn.Dropout(dropout)


    def forward(self, src):
        #src = [src len, batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        outputs, (hidden, cell) = self.rnn(embedded)

        return hidden, cell
```



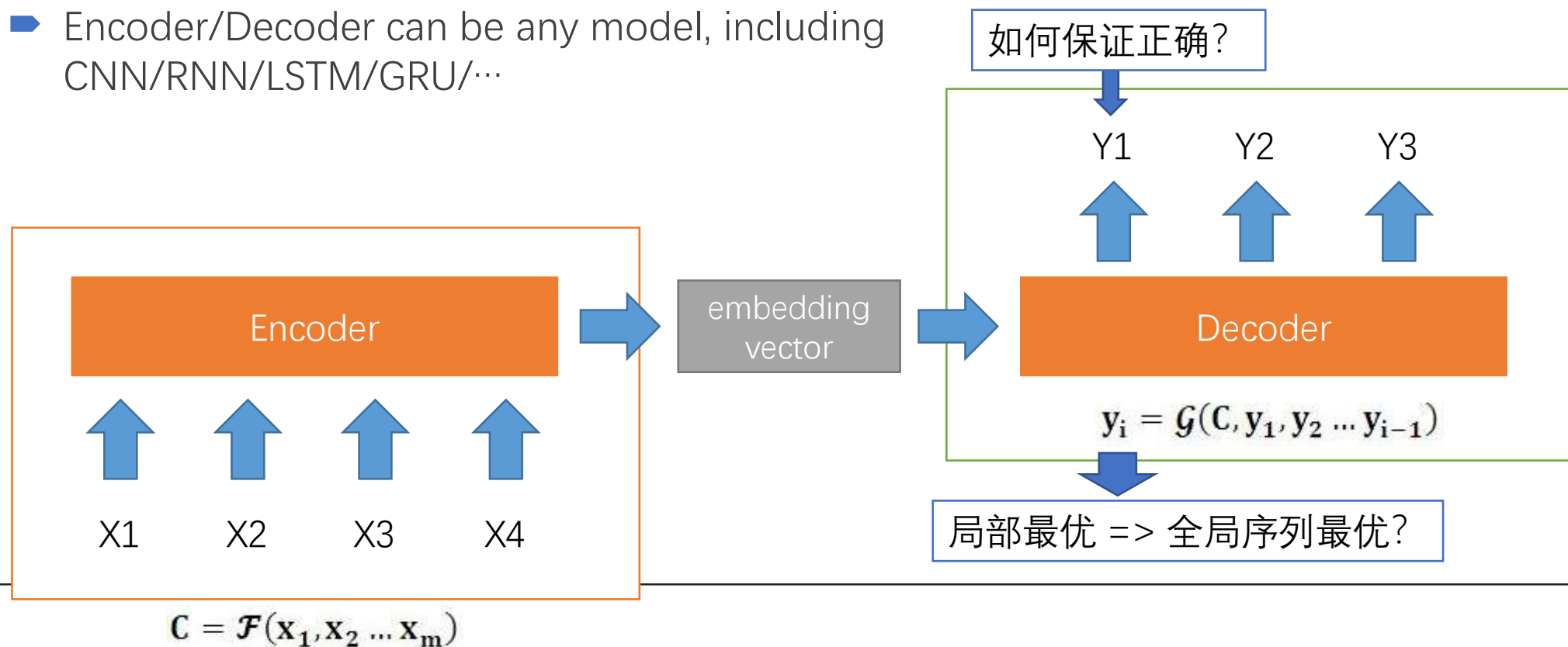
self.embedding = nn.Embedding(input\_size, hidden\_size)  
self.gru = nn.GRU(hidden\_size, hidden\_size)



# 解码过程基本架构:

- 解决seq2seq效率问题的一种方法
- 定长输入 → 定长输出, 多余的用空字符补齐
- Encoder/Decoder can be any model, including CNN/RNN/LSTM/GRU/...

```
if use_teacher_forcing:  
    # Teacher forcing: Feed the target as the next input  
    for di in range(target_length):  
        decoder_output, decoder_hidden, decoder_attention = decoder(  
            decoder_input, decoder_hidden, encoder_outputs)  
        loss += criterion(decoder_output, target_tensor[di])  
        decoder_input = target_tensor[di] # Teacher forcing
```



# 解码过程:

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```



# The problem of RNN in Encoder-decoder

- ➡ Decoder中的输入的每个embedding vector都是相同的
- ➡ 这意味着
  - ➡ 一个向量难以包含全部序列信息
  - ➡ 先前输入的信息会逐级衰减
  - ➡ 难以根据context对齐



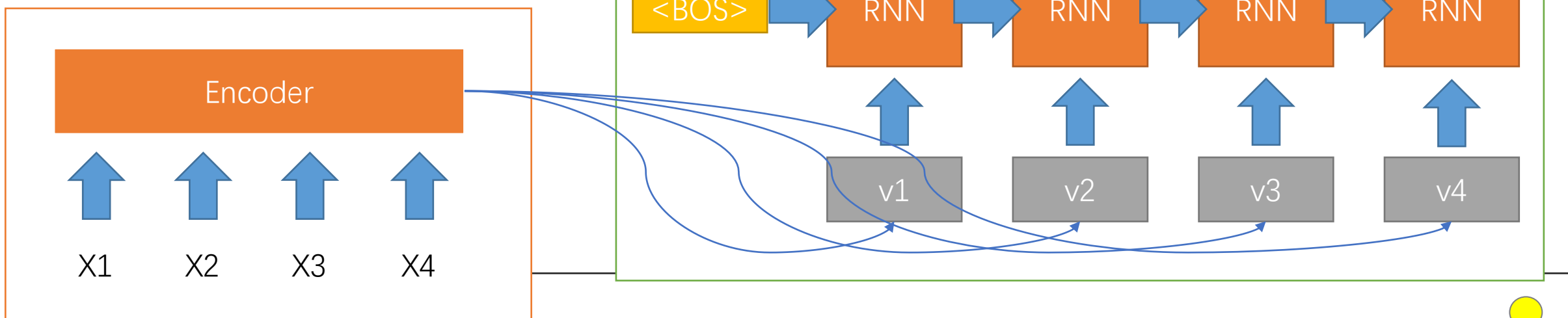
# Attention Model

- 增加了一个注意力范围，强调接下来输出内容应该关注哪一部分

$$y_1 = f1(C_1)$$

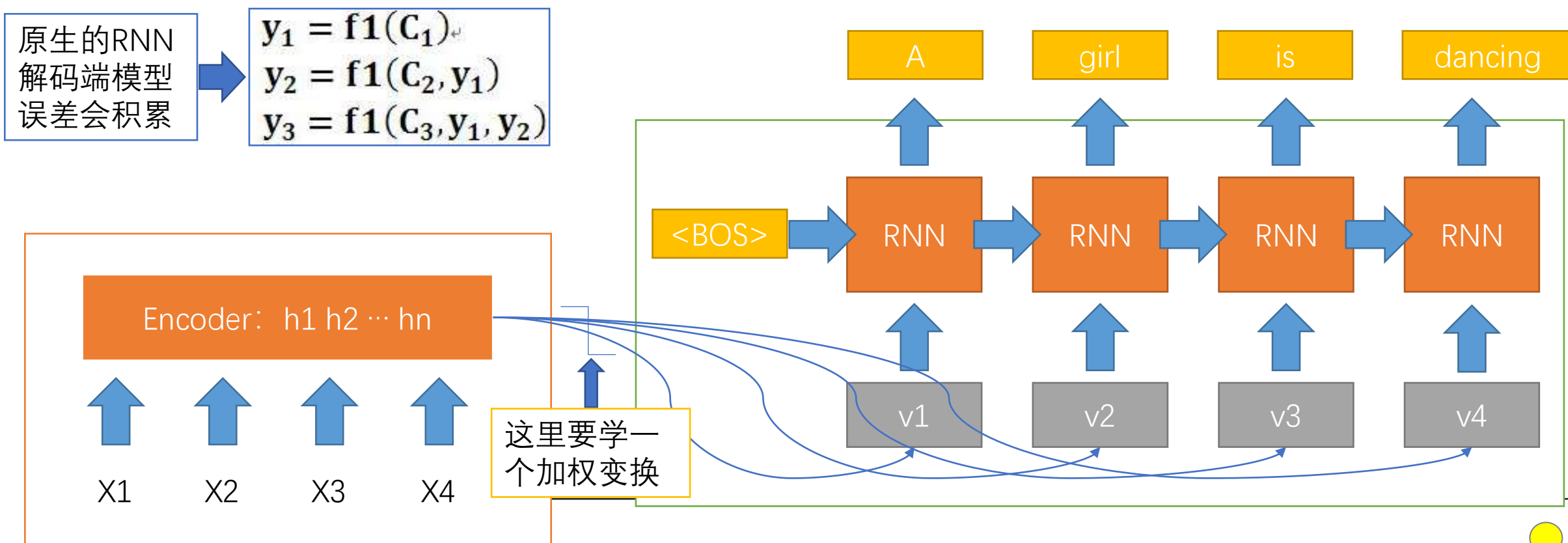
$$y_2 = f1(C_2, y_1)$$

$$y_3 = f1(C_3, y_1, y_2)$$



# seq2seq中常规的Attention方案：

- 增加了一个注意力范围，强调接下来输出内容应该关注哪一部分。本质上属于一种词典学习。更多的参数、更多的信息输入。



# 生成结果评测： BLEU score

- BLEU 考察输出语句和参考语句之间单词串的重叠程度。考虑系统输出语句中单词的 1 元组至 N 元组是否有在参考语句中出现，以此计算出 N 个精度（precision），并以下式计算 BLEU 分数。

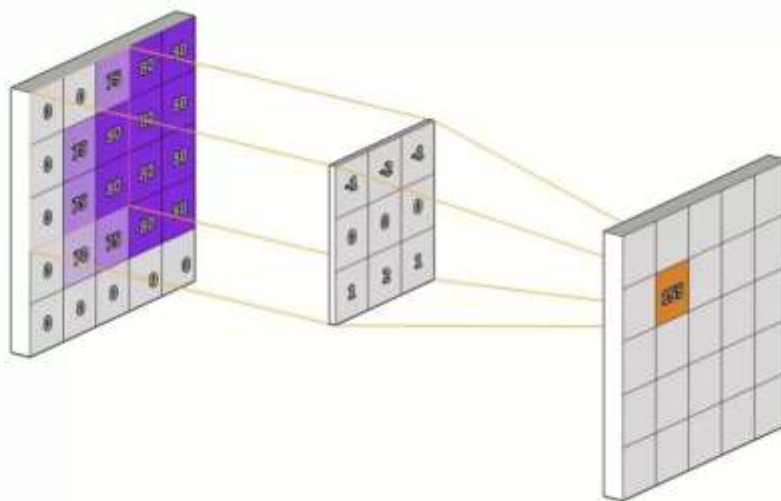
$$BLEU = BP \times \exp \sum_{n=1}^N w_n \log Precision_n$$

$$BP = \begin{cases} e^{1-\frac{r}{c}} & \text{if } c \leq r \\ 1 & \text{if } c > r \end{cases}$$



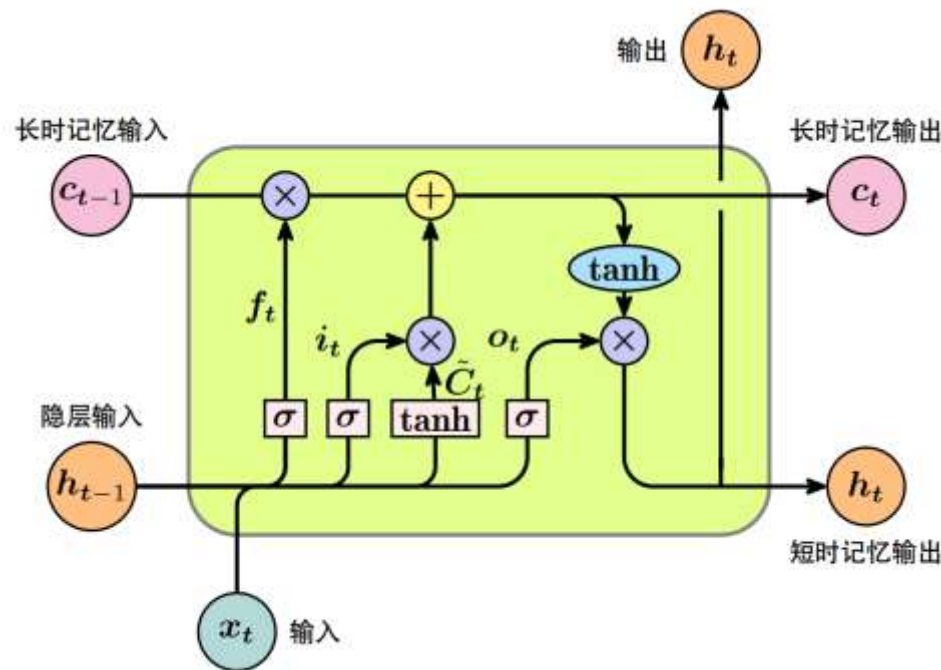
# CNN,

- CNN: 通过多通道卷积核独立地提取特征，通过层深不断集成不同特征
- 优点：便于并行化、可独立抽取多维度局部特征
- 缺点：只能处理局部特征；不具备建模位置信息、时序依赖关系的能力



# RNN, LSTM, Seq2Seq

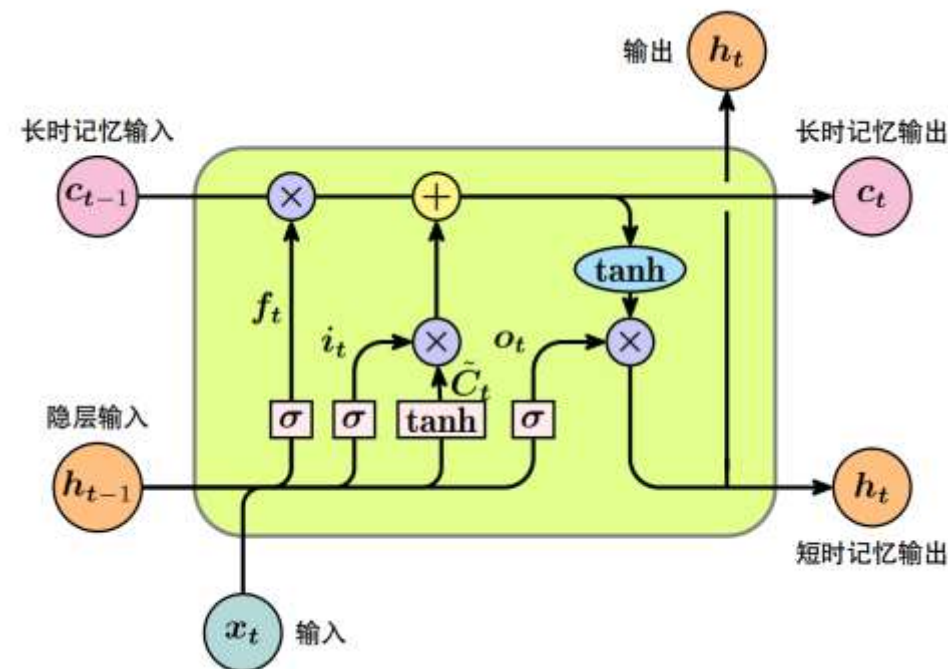
- LSTM: 通过长短期记忆+门控设计, 解决梯度消失/爆炸问题, 可处理更长的序列
  - 20->100 ✓
  - 100->1000 ?





# LSTM处理序列数据的缺点

- 即使改善了长期依赖问题，处理超长序列（>500）依然很困难
- 计算时的时间开销很大，不便于并行化计算



# Transformer结构介绍

# Transformer所解决的问题

- 依然沿用了Encoder-Decoder的思路，融合了CNN/RNN的长处
  - CNN: 并行化、抽取独立子空间的特征（多头注意力机制）；增添了位置编码
  - RNN: 长序列建模、序列依赖关系（自注意力机制）
  - LSTM: Long-Term Memory中类似残差连接的机制防止梯度消失（残差连接、FFN）
- 解决长距离依赖问题
  - 词和词两两之间的相关性被全部保留
- 全连接式地计算词两两之间的相关性，无需通过隐藏层传递
- 便于并行化计算
  - 可以增加训练数据、增大模型规模，从而不断刷新上限（如GPT-3 175B）

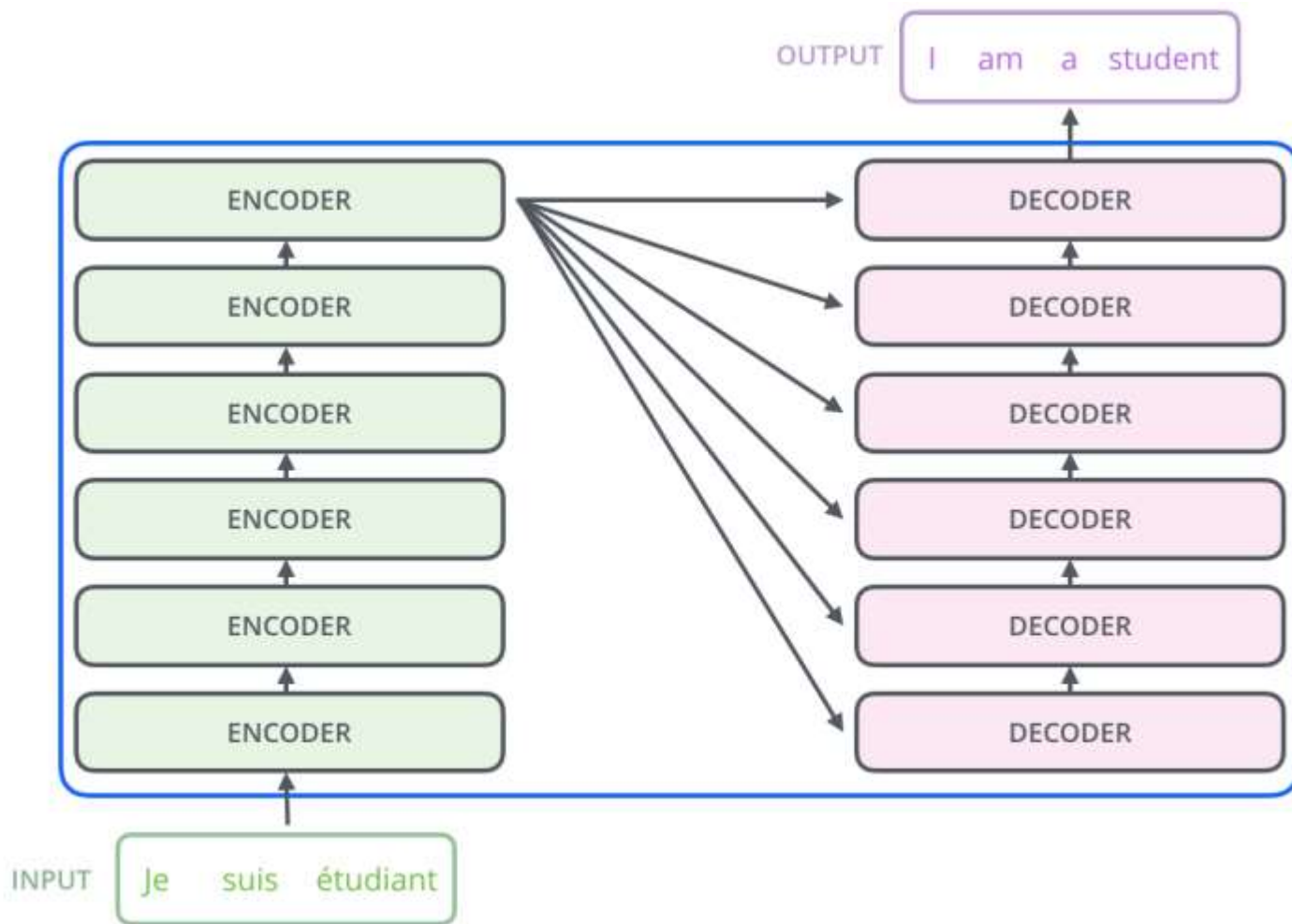


# Background

- Attention is all you need论文链接: <https://arxiv.org/abs/1706.03762/>
- 背景1: 使用卷积网络来减少时序计算量 (Sequential Computation), 增大GPU的并行计算效率
  - 缺点: 很难对长距离依赖进行建模
  - 优点: 多个输出通道; 对应Transformer中的Multi-head Attention
- 背景2: Self-Attention, memory network等相关工作
- Transformer是第一个只依赖于SA来做Encoder-Decoder架构的模型



# 自顶向下——总体概览



依然是Encoder-Decoder架构，  
其中Encoder和Decoder分别由N  
个相同的block组成

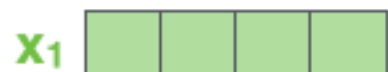
# 关键点

- 模型结构
  - 编码器：Self Attention、Feed-Forward Network、残差连接
  - 解码器：Self Attention、Encoder-Decoder Attention、Feed-Forward Network、残差连接
- 从输入到输出的完整数据流
  - 编码计算过程（Self-Attention、FFN、残差连接）
  - 解码计算过程（Self-Attention、Cross-Attention、one-token per step）
- 重要概念
  - “多头”注意力（Multi-Head Attention）
  - 位置编码（Positional Encoding）

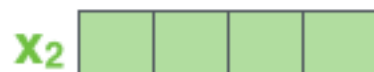


# 输入

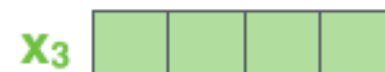
- 词向量：将一个词编码成d维的向量表示
- 句子表示形式：一个由词向量构成的数组，即 $N \times d$ 的二维矩阵
  - 示例：图中词向量维度 $d=4$ ，句子长度 $N=3$



Je

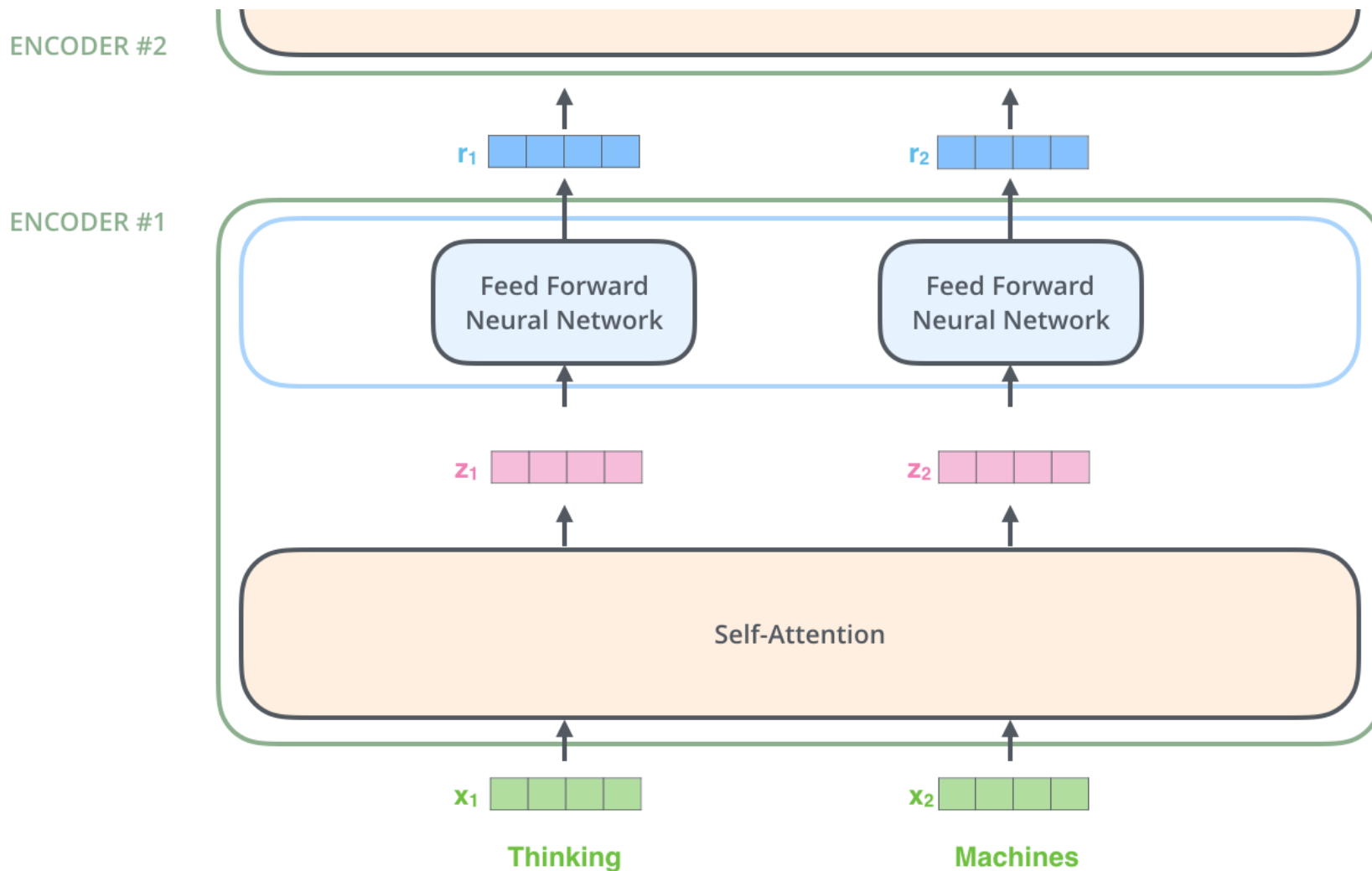


suis



étudiant

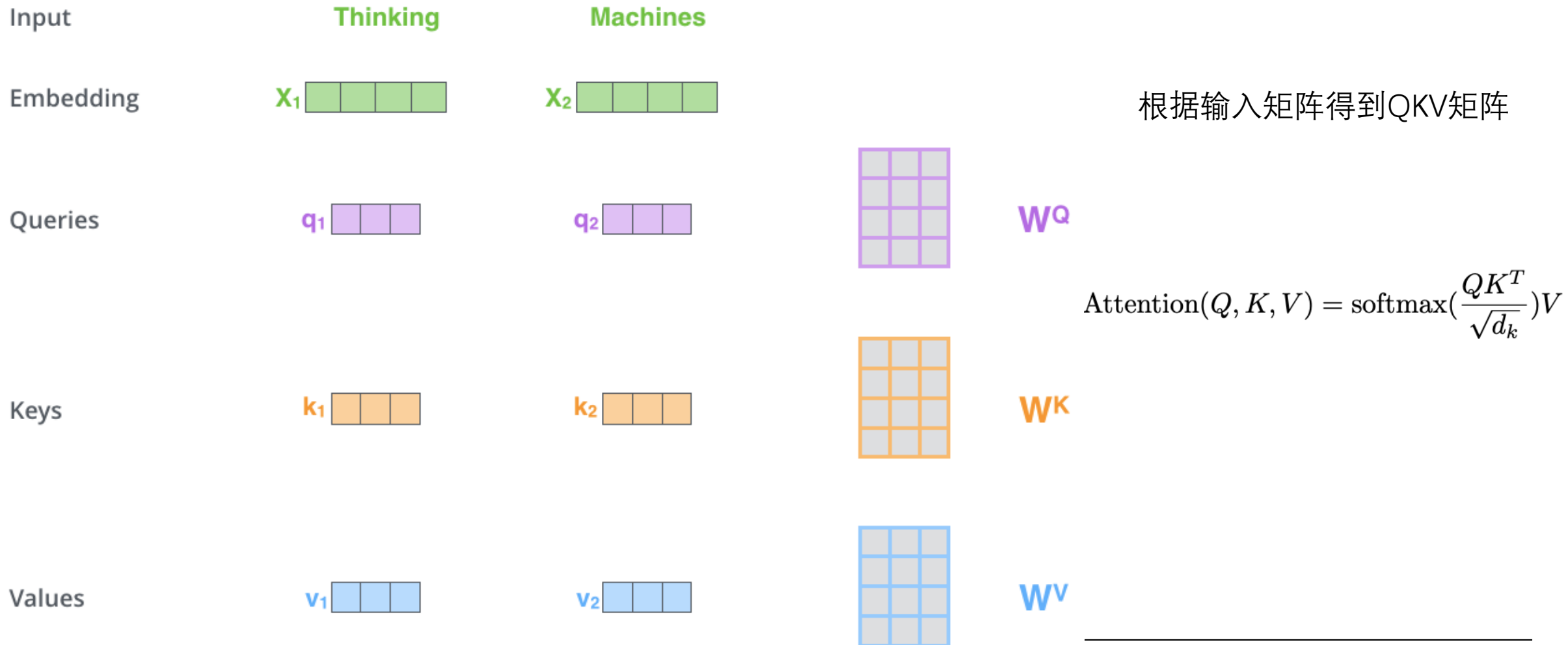
# 编码过程



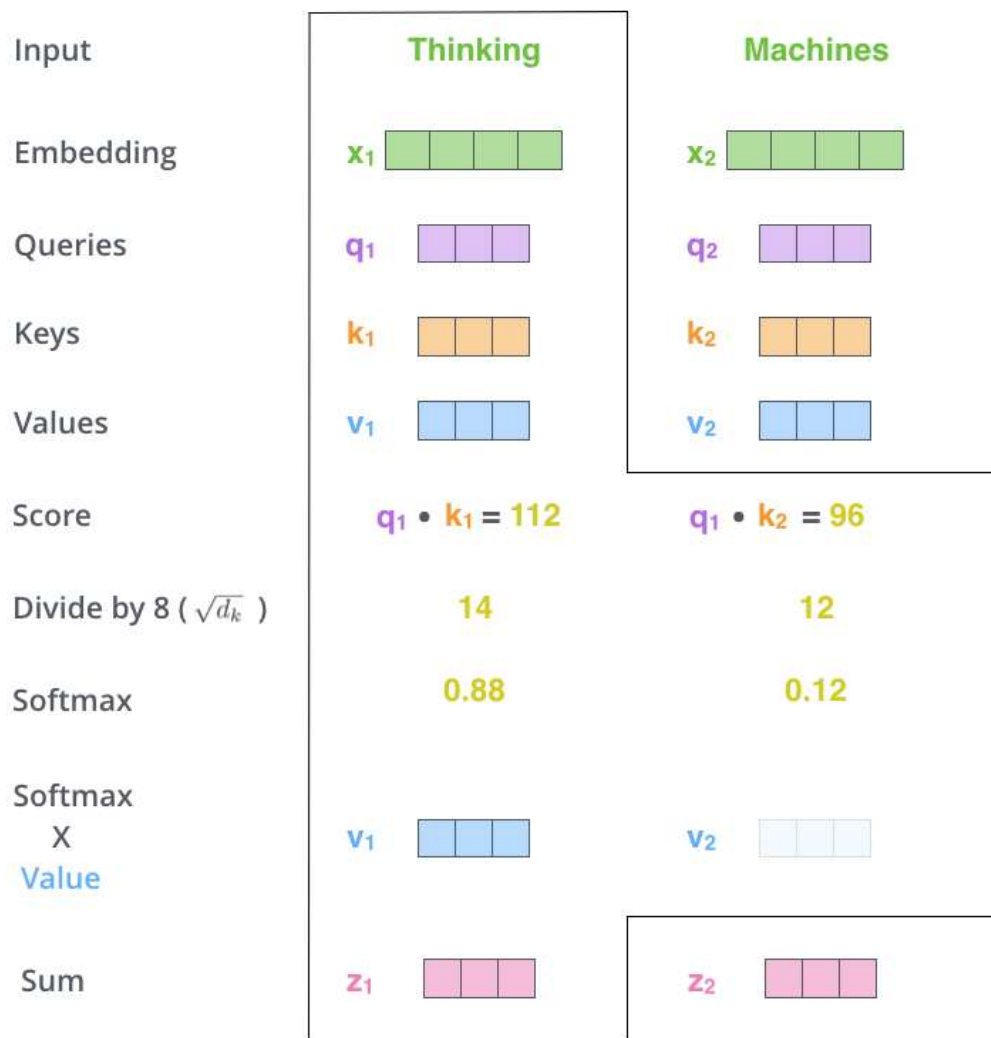
每个encoder block的地位和作用流程是完全相同的



# Self-Attention详解



# Self-Attention详解



根据输入得到Query和Key向量，经Softmax计算出当前词的权重，利用权重对所有Value向量加权求和，得到当前位置下一层的表示

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

关于注意力机制的一个不错的解读文章：  
<https://zhuanlan.zhihu.com/p/37601161>

# Self-Attention——矩阵运算

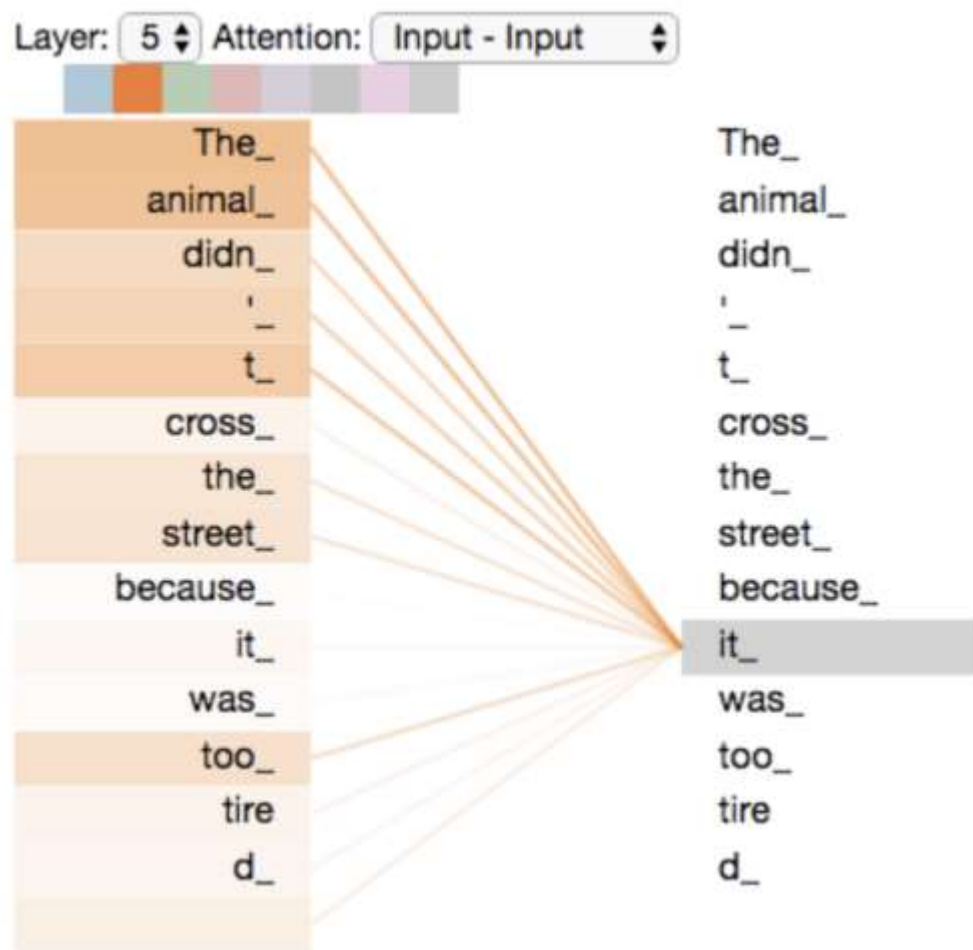
$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{Q}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{Q} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{K}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{K} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

$$\begin{matrix} \text{X} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{W}^{\text{V}} \\ \begin{array}{|c|c|c|c|} \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \end{matrix} = \begin{matrix} \text{V} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

$$\text{softmax} \left( \frac{\begin{matrix} \text{Q} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{K}^{\text{T}} \\ \begin{array}{|c|c|} \hline & \\ \hline & \\ \hline & \\ \hline \end{array} \end{matrix}}{\sqrt{d_k}} \right) \begin{matrix} \text{V} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$
$$= \begin{matrix} \text{Z} \\ \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline \end{array} \end{matrix}$$

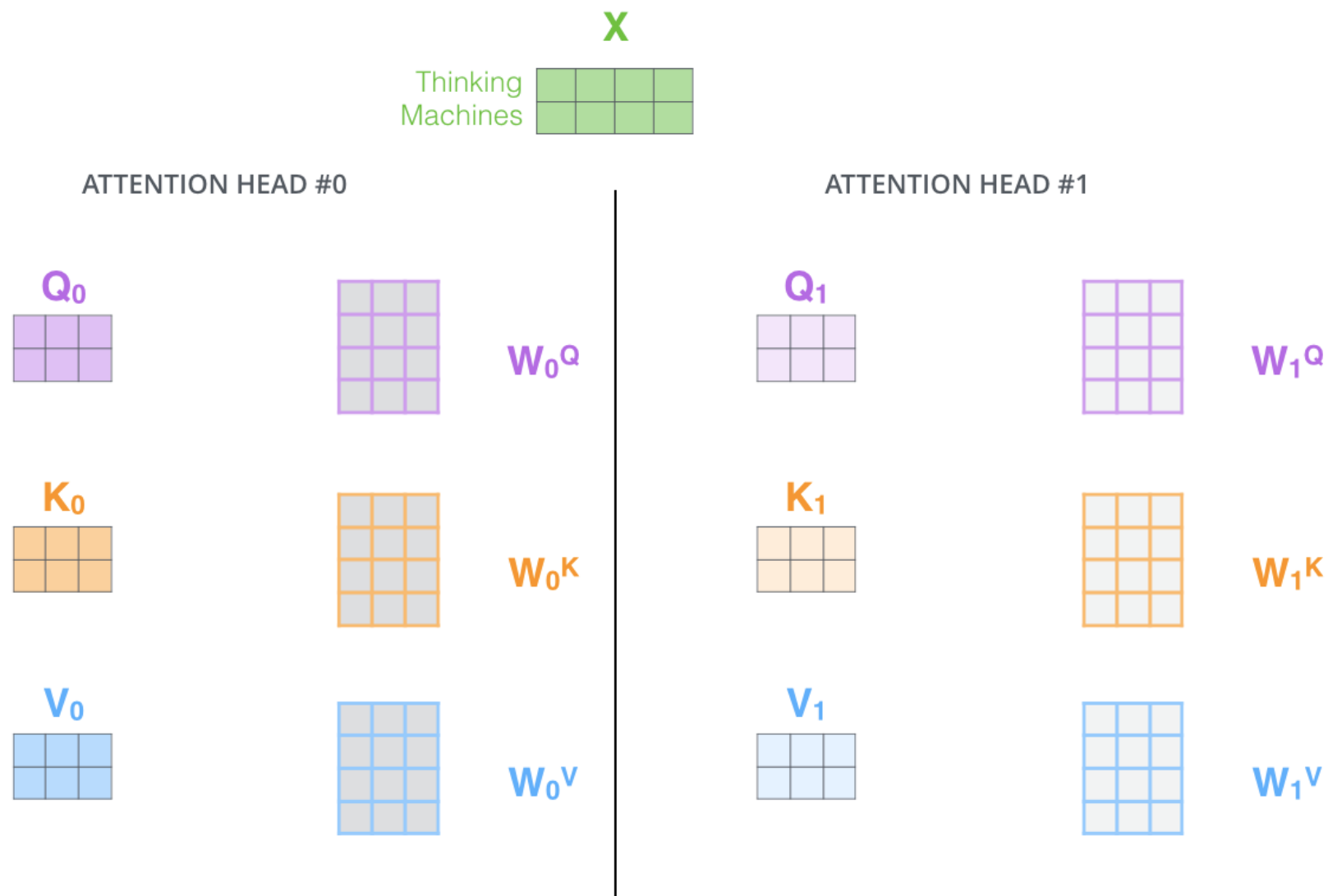
# Self-Attention——可视化展示



左侧代表第k层，右侧代表第k+1层

第k+1层的每个词都融合了前一层所有词的信息，只是权重有大有小

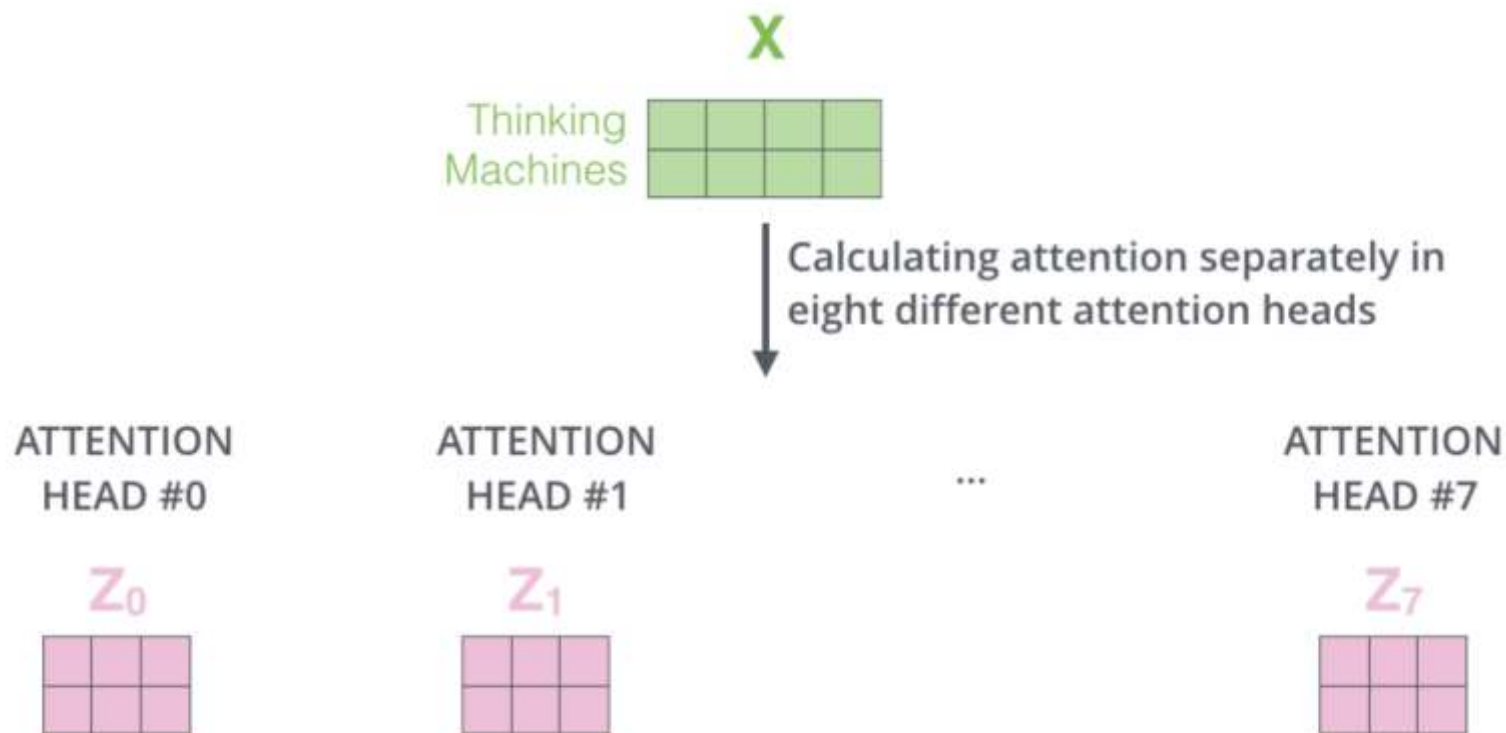
# 更进一步——多头注意力机制



类似CNN的不同channel, Attention Head也可以设置成多个, 并且每个都是独立随机初始化的, 通过训练可以从数据中学到不同方面的特征



# 更进一步——多头注意力机制



分别计算得到每个头的输出

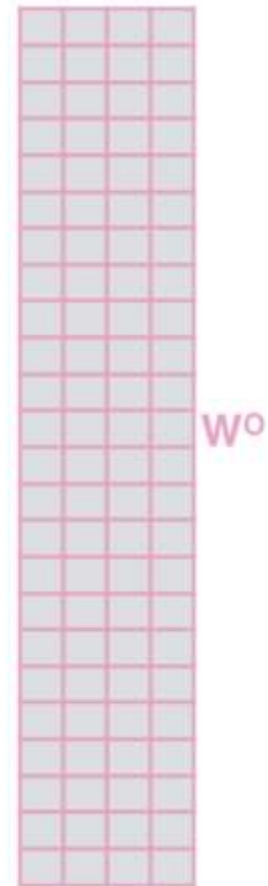
# 更进一步——多头注意力机制

1) Concatenate all the attention heads



2) Multiply with a weight matrix  $W^O$  that was trained jointly with the model

x



3) The result would be the  $Z$  matrix that captures information from all the attention heads. We can send this forward to the FFNN

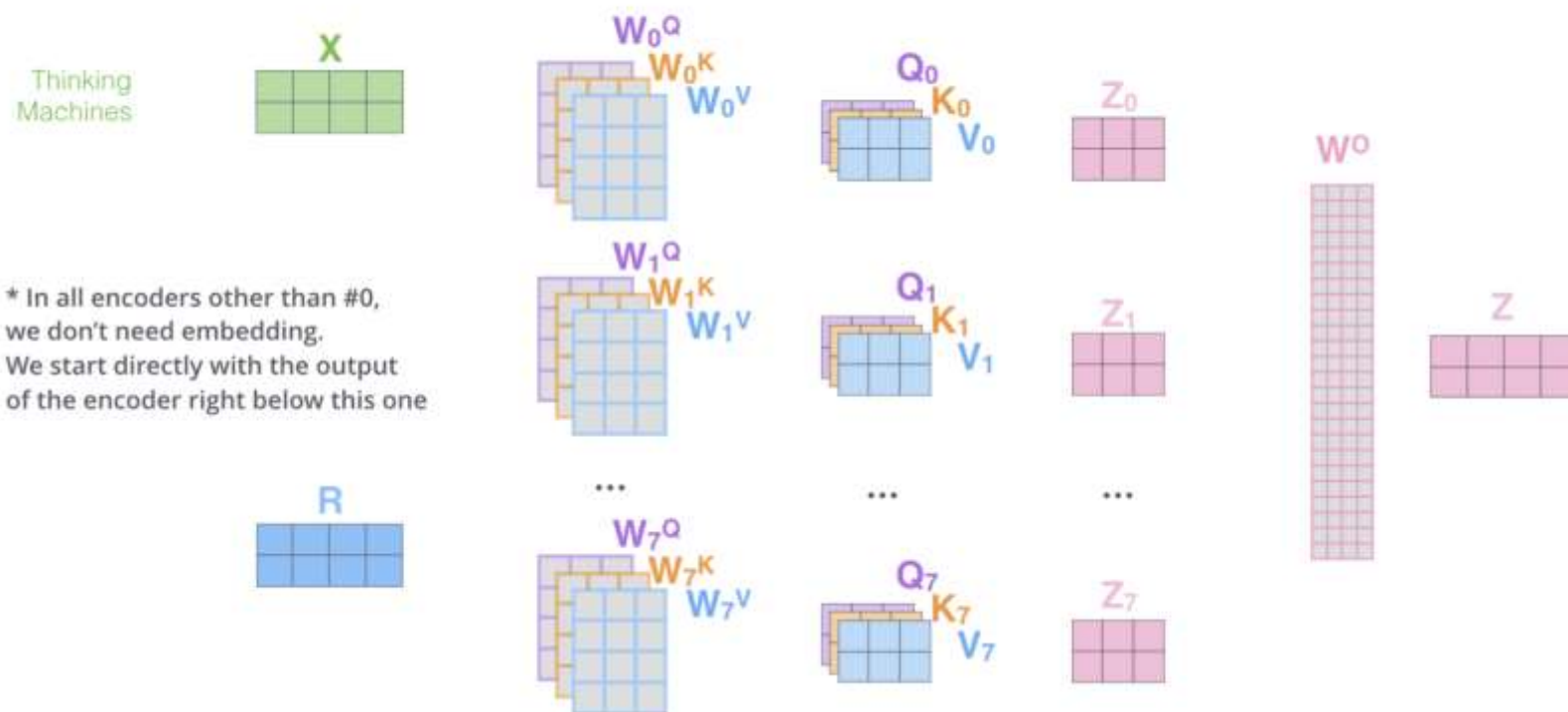


$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

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

# 多头注意力机制——总结

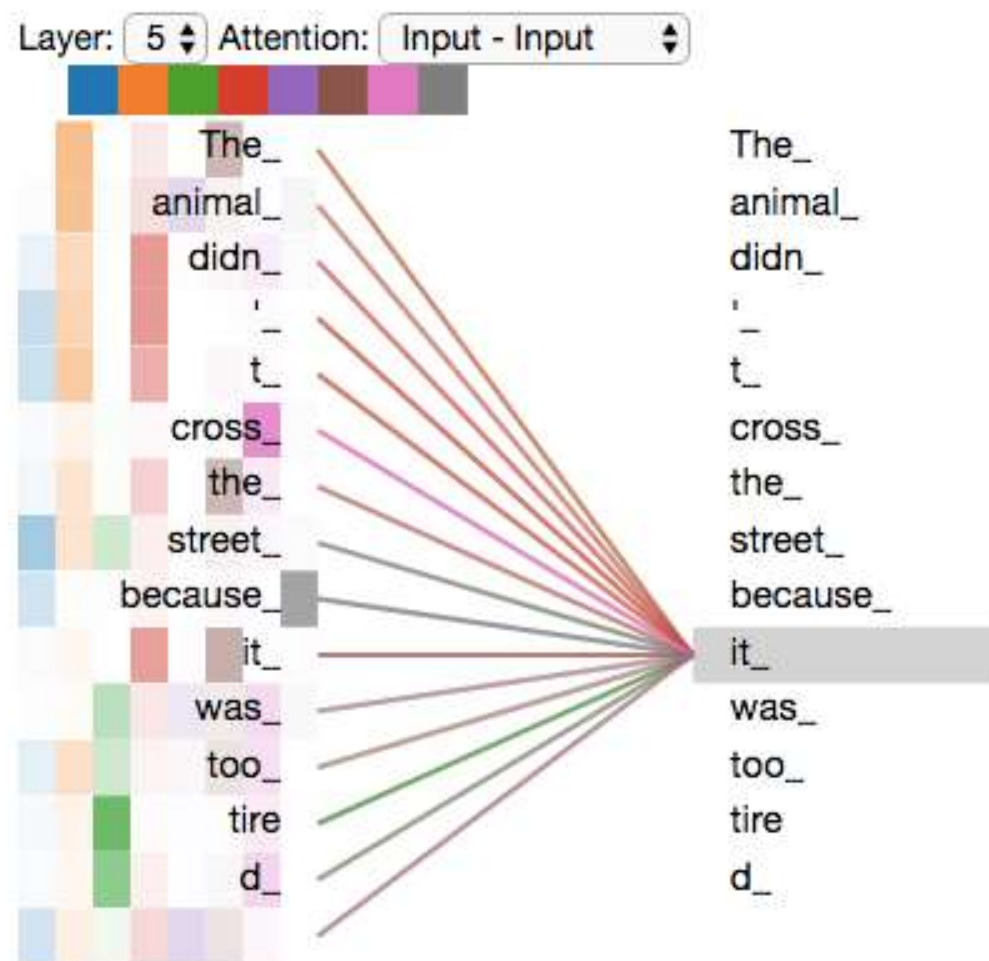
- 1) This is our input sentence\*
- 2) We embed each word\*
- 3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices
- 4) Calculate attention using the resulting  $Q/K/V$  matrices
- 5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer



$Q_0 \sim Q_n$ 相互独立，便于捕获不同子空间的特征

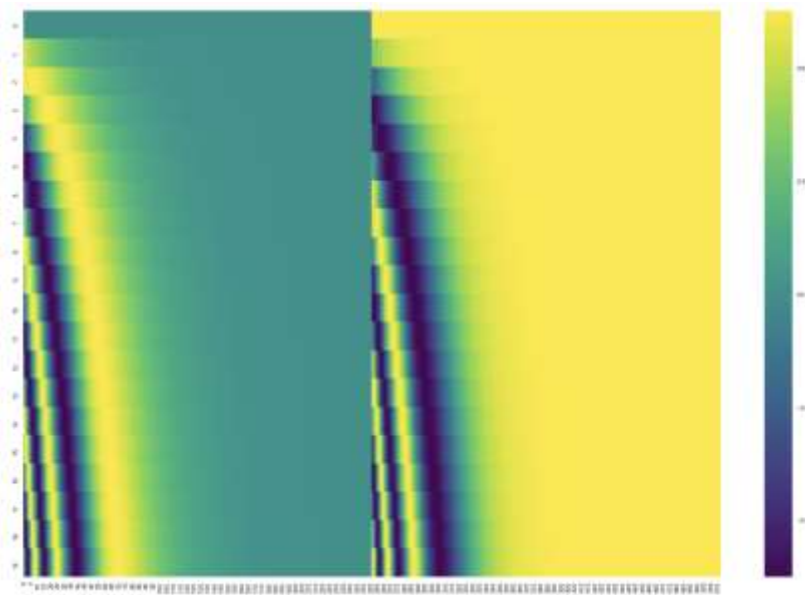
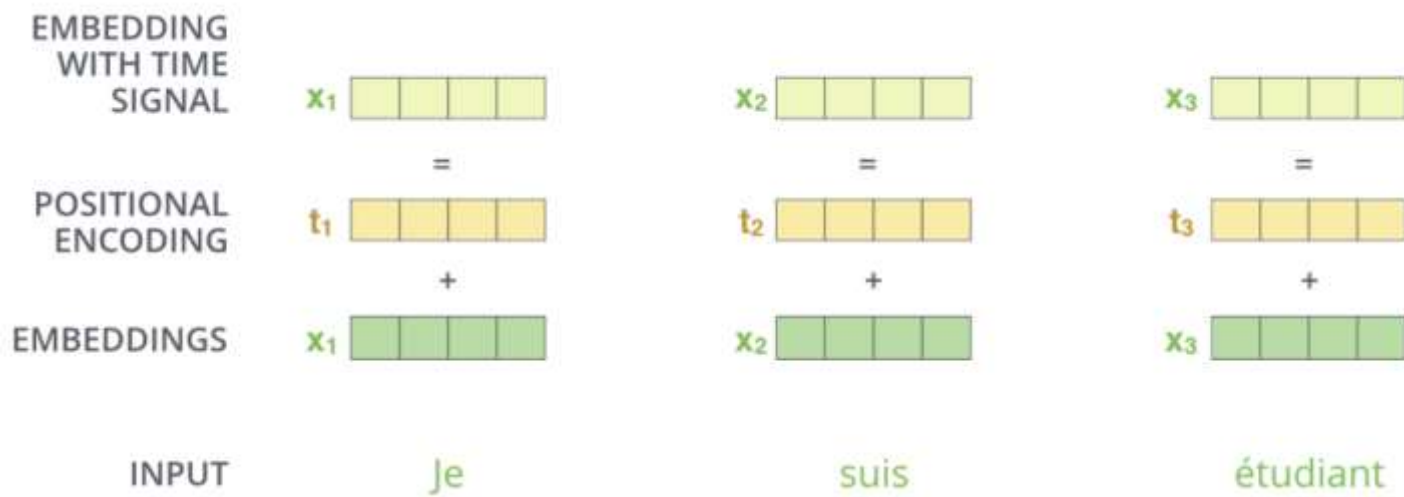


# 多头注意力机制——可视化展示

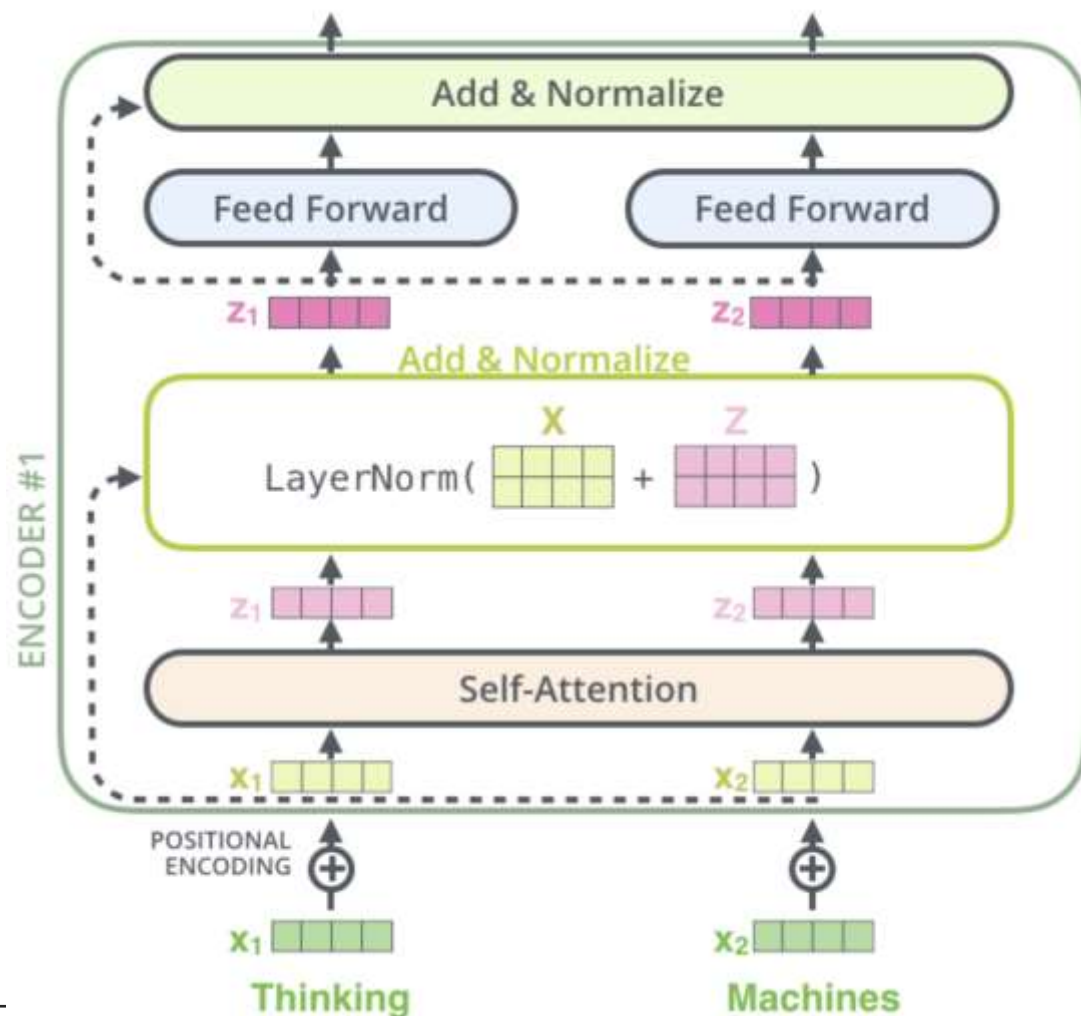


不同颜色代表不同的  
Attention Head

## 位置编码——给单词添加位置信息



# 前馈神经网络与残差连接



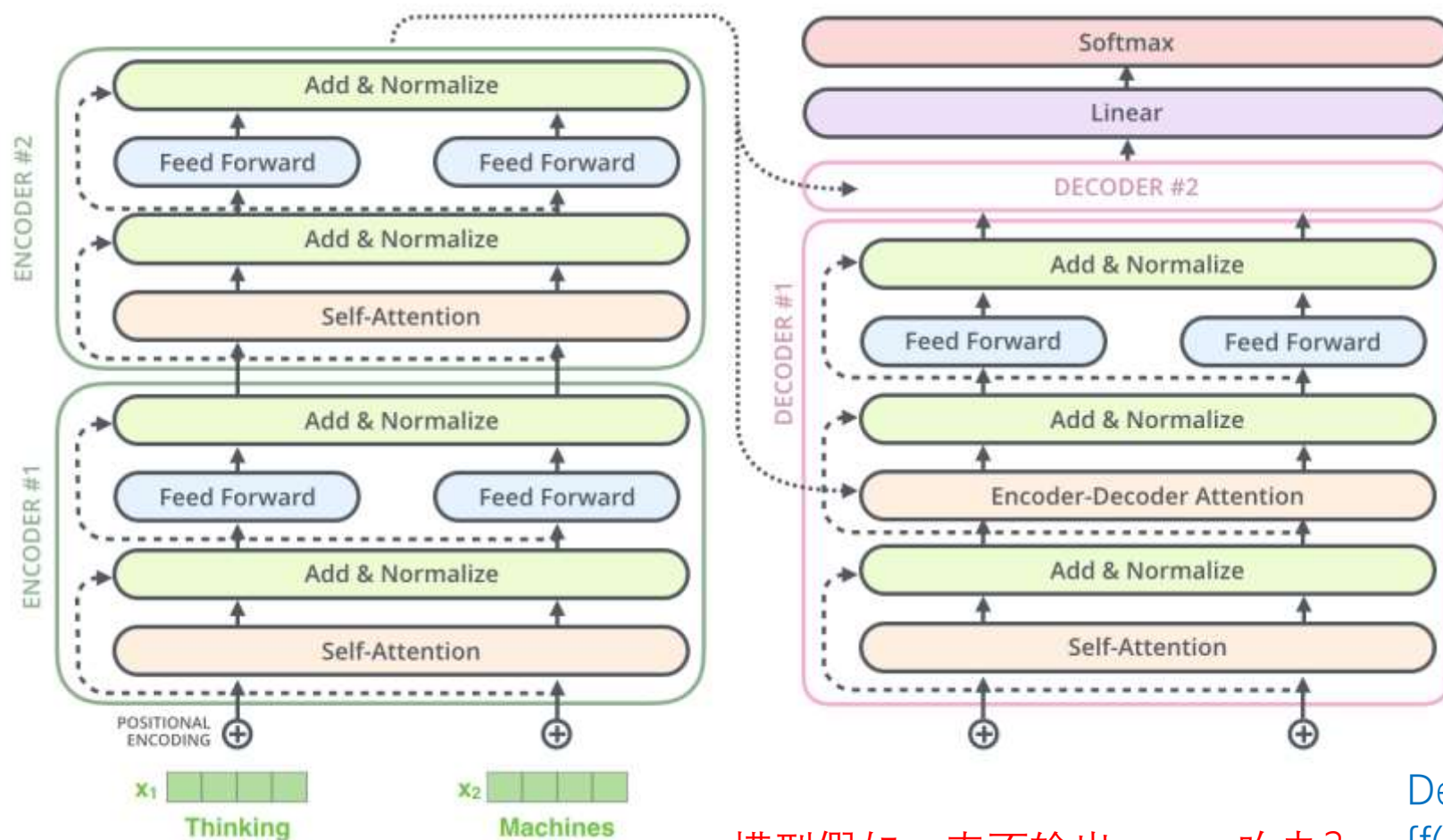
前馈神经网络（Feed-Forward Network）：  
其实就是两个全连接层中间加一个非线性激活函数构成的双层感知机。

主要作用：将Attention后的特征升维再降维（提高模型容量），并且加入了非线性变换增强表示能力

另一种理解：相当于对原位置做了 $1 \times 1$ 的卷积，巩固每一个位置信息的表示

残差连接： $X_2 = X_1 + \text{Encoder\_Layer}(X_1)$   
主要作用：防止过拟合、保持长期记忆、解决梯度消失和梯度爆炸问题

# Decoder结构



依然保有的Self-Attention：计算当前已解码序列的表示

**增加了Encoder-Decoder Attention：**基于编码器得到的上下文向量与解码器当前输出的向量，解码下一时刻的token

编码阶段：整个序列计算一次直接得到下一层的输出

解码阶段：每一时刻只输出1个token，直至出现<eos>结束符号

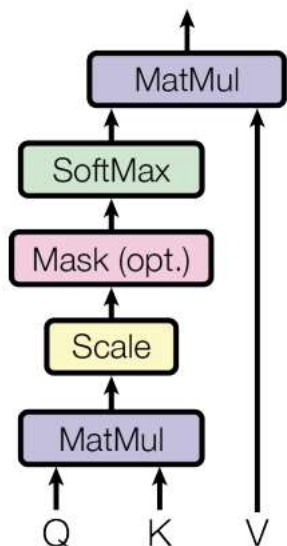
模型假如一直不输出<eos>咋办？

Decoder可以理解为一个映射的集合：  
 $\{f(\text{sos}) = x_1; f(\text{sos}, x_1) = x_2;$   
 $f(\text{sos}, x_1, x_2) = x_3; f(\text{sos}, x_1, x_2, x_3) = \text{eos}\}$

# 模型结构——论文中的原图

- 对于Encoder端的Self-Attention: Q, K, V均相同
- 对于Decoder端的MH-Attention: Q来自Decoder输入,
- K, V来自Encoder的输出。

Scaled Dot-Product Attention



Multi-Head Attention

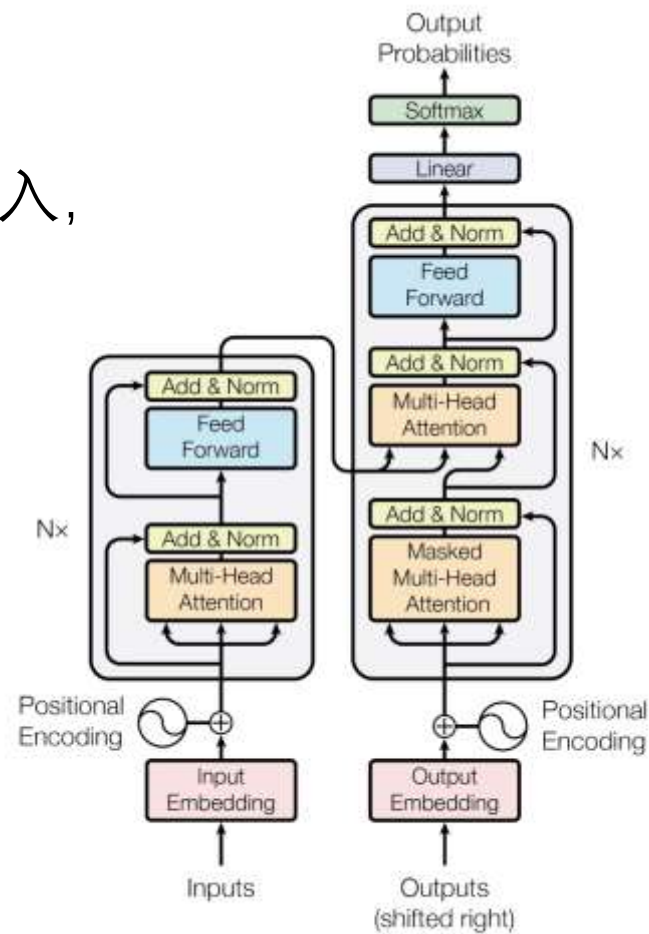
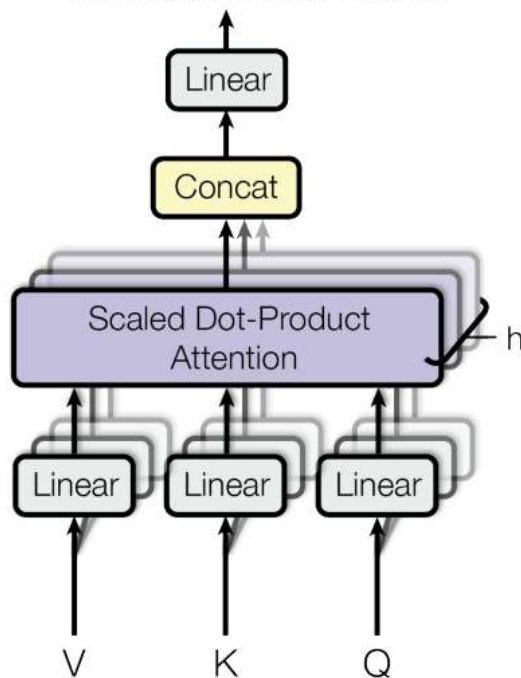


Figure 1: The Transformer - model architecture.



# 变量、公式及关键概念梳理

- Encoder: 包含(N=6)个相同的层, 输入 $(x_1, \dots, x_n)$ , 得 $\bar{z} = (z_1, \dots, z_n)$ .
  - Encoder Self-Attention: Q, K, V都来自输入序列X, 算自己内部的自注意力
- Decoder: 包含(N=6)个相同的层, 通过 $\bar{z} = (z_1, \dots, z_n)$  和  $(y_1, \dots, y_m)$  得到最终结果
  - Decoder Self-Attention: Q, K, V都来自当前已解码出的序列Y, 算output内部的自注意力
  - Decoder Cross-Attention: Q来自Y, K和V来自 $\bar{z}$ , 算交叉注意力
- 自注意力公式:  $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 
  - 分成多个头:  $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$   
where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$
  - 除以sqrt(d\_k): 相当于scale操作, 训练时保持梯度稳定
- 每个Layer中的前向FFN (包含残差连接):  $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$
- 包含位置信息的Positional Encoding

