

# 神经网络基础

车万翔、郭江、崔一鸣

社会计算与信息检索研究中心  
哈尔滨工业大学

## 感知器 (Perceptron)

- 最简单也是最早出现的机器学习模型
- 灵感直接来源于生产生活的实践

## 面试评分

- $s = x_1 + x_2 + x_3 + x_4$ 
  - 如果  $s \geq t$  则录用, 否则不录用
- 考虑面试官的经验 (权重)
  - $s = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$

## 感知器模型

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{else} \end{cases}, \text{ 其中 } b = -t$$



## □ 线性回归 (Linear Regression)

□  $y = \sum_i w_i x_i + b$

## □ Logistic回归 (Logistic Regression)

### □ Logistic函数

$$y = \frac{L}{1 + e^{-k(z-z_0)}}$$

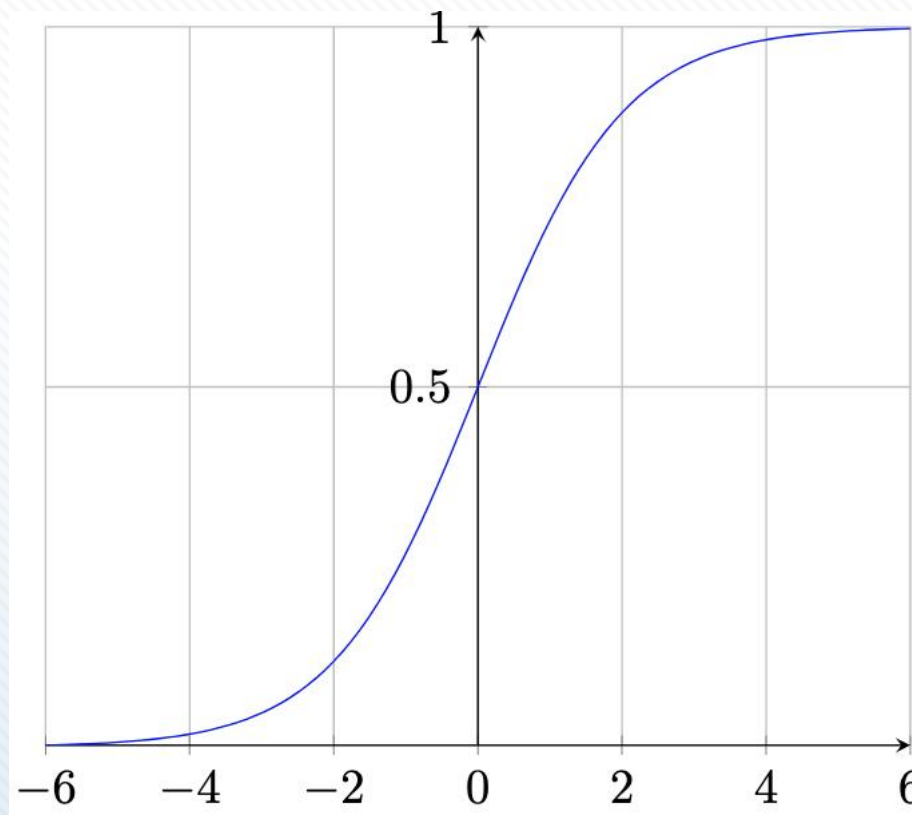
□ 设  $z = \sum_i w_i x_i + b$

### □ Sigmoid函数 (Logistic函数的特例)

$$y = \frac{1}{1 + e^{-z}}$$

### □ 处理二元分类问题, $y$ 为输出的**概率**

□ 垃圾邮件过滤、褒贬识别



Sigmoid函数示意图

□ 使用Softmax回归处理多元分类

□ Softmax函数

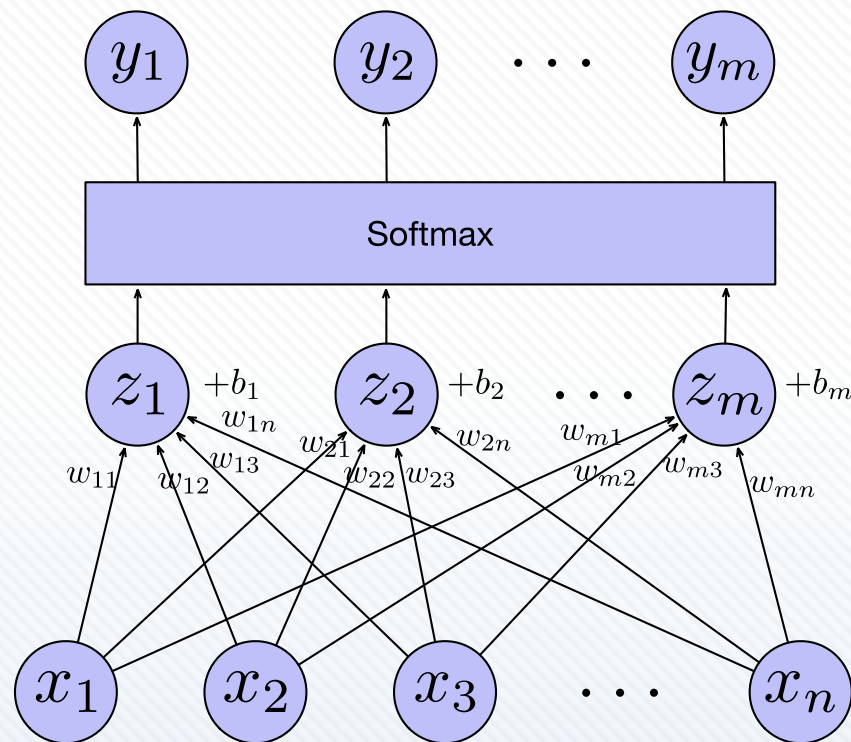
$$y_i = \text{Softmax}(z)_i = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + \dots + e^{z_m}}$$

□ 其中 $y_i$ 为第 $i$ 个类别的概率,  $z_i = \sum_j w_{ij}x_k + b$ ,  
 $w_{ij}$ 为第 $i$ 个类别所对应的第 $j$ 个输入的权重

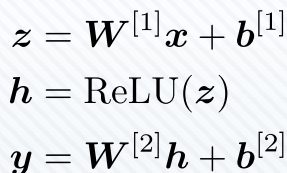
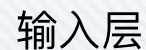
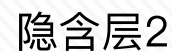
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \text{Softmax} \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \dots + w_{1n}x_n + b_1 \\ w_{21}x_1 + w_{22}x_2 + \dots + w_{2n}x_n + b_2 \\ \vdots \\ w_{m1}x_1 + w_{m2}x_2 + \dots + w_{mn}x_n + b_m \end{pmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \text{Softmax} \left( \begin{bmatrix} w_{11}, w_{12}, \dots, w_{1n} \\ w_{21}, w_{22}, \dots, w_{2n} \\ \vdots \\ w_{m1}, w_{m2}, \dots, w_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \right)$$

$$y = \text{Softmax}(Wx + b)$$



- 堆叠多层感知器（线性回归+非线性激活函数）
- 可解决线性不可分问题



## □引入PyTorch模块

```
□>>> from torch import nn
```

## □线性层

```
>>> linear = nn.Linear(32, 2) # 输入32维, 输出2维
>>> inputs = torch.rand(3, 32) # 创建一个形状为(3, 32)的随机张量, 3为批次大小
>>> outputs = linear(inputs) # 输出张量形状为(3, 2)
>>> print(outputs)
tensor([[ 0.5387, -0.4537],
        [ 0.2181, -0.3745],
        [ 0.3704, -0.8121]], grad_fn=<AddmmBackward>)
```

## □激活函数

```
□>>> from torch.nn import functional as F
```

```
>>> activation = F.sigmoid(outputs)
>>> print(activation)
tensor([[0.6315, 0.3885],
        [0.5543, 0.4075],
        [0.5916, 0.3074]], grad_fn=<SigmoidBackward>)
>>> activation = F.softmax(outputs, dim=1)
# 沿着第2维进行Softmax运算, 即对每批次中的各样例分别进行Softmax运算
>>> print(activation)
tensor([[0.7296, 0.2704],
        [0.6440, 0.3560],
        [0.7654, 0.2346]], grad_fn=<SoftmaxBackward>)
>>> activation = F.relu(outputs)
>>> print(activation)
tensor([[0.5387, 0.0000],
        [0.2181, 0.0000],
        [0.3704, 0.0000]], grad_fn=<ReluBackward0>)
```



## □ 自定义神经网络实现

```
class MLP(nn.Module):  
    def __init__(self, input_dim, hidden_dim, num_class):  
        super(MLP, self).__init__()  
        # 线性变换: 输入层->隐含层  
        self.linear1 = nn.Linear(input_dim, hidden_dim)  
        # 使用ReLU激活函数  
        self.activate = F.relu  
        # 线性变换: 隐含层->输出层  
        self.linear2 = nn.Linear(hidden_dim, num_class)  
  
    def forward(self, inputs):  
        hidden = self.linear1(inputs)  
        activation = self.activate(hidden)  
        outputs = self.linear2(activation)  
        probs = F.softmax(outputs, dim=1) # 获得每个输入属于某一类别的概率  
        return probs
```

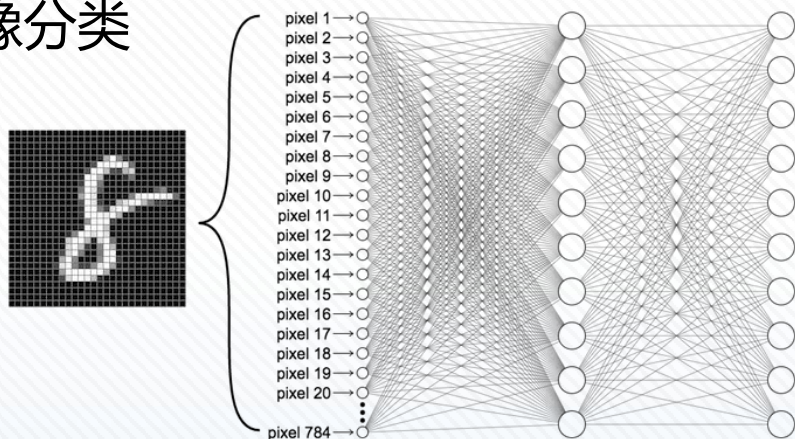
```
mlp = MLP(input_dim=4, hidden_dim=5, num_class=2)  
inputs = torch.rand(3, 4) # 输入形状为(3, 4)的张量, 其中3表示有3个输入, 4表示每个  
                           输入的维度  
probs = mlp(inputs) # 自动调用forward函数  
print(probs) # 输出3个输入对应输出的概率
```

<https://colab.research.google.com/drive/1EMJJf fsmISlqO3KwuG5WIP-lxyNVXTrK?usp=sharing>

colab

## 全连接的多层感知器无法处理输入的偏移情况

### 图像分类



### 情感分类

我喜欢自然语言处理。 vs. 我非常喜欢自然语言处理。

## 解决方案

使用小的全连接层抽取局部特征（又称卷积核或滤波器）

如遍历文本中的N-gram等

使用多个卷积核提取不同种类的特征

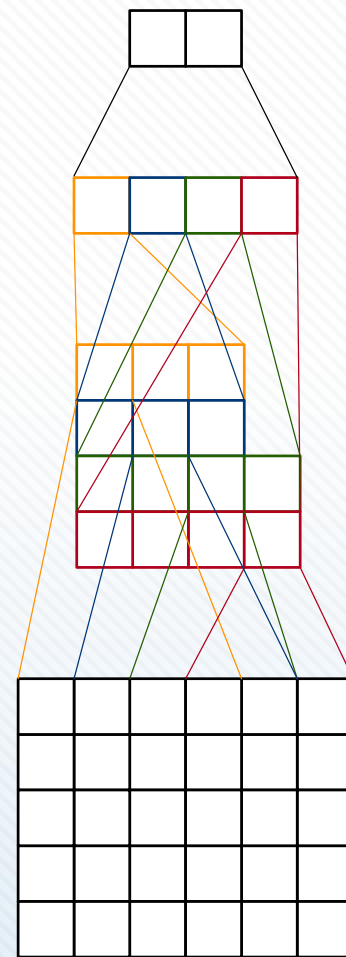
使用池化层将特征进行聚合（最大、平均、求和等）

全连接层

池化层

卷积层

输入层



我 喜 自 语 处 。  
欢 然 言 理

colab

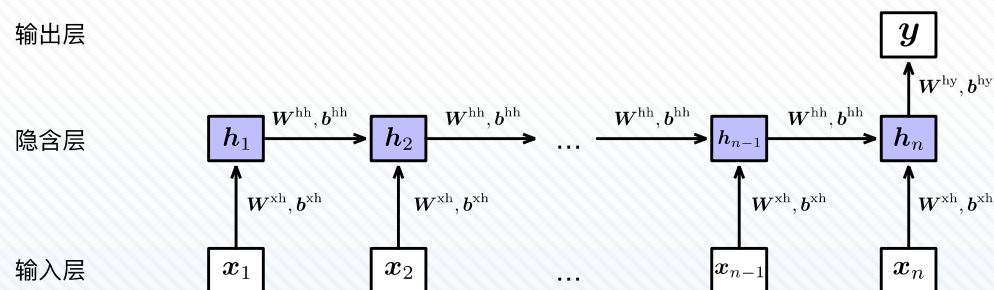
<https://colab.research.google.com/drive/1HaPOfEj8wrLkqeUDuirtS-cNKGxeRdY5?usp=sharing>



❑ 卷积神经网络无法处理长距离的依赖

❑ 循环神经网络的解决方案

❑ 每个时刻的隐状态依赖于当前时刻的输入以及上一时刻的隐状态

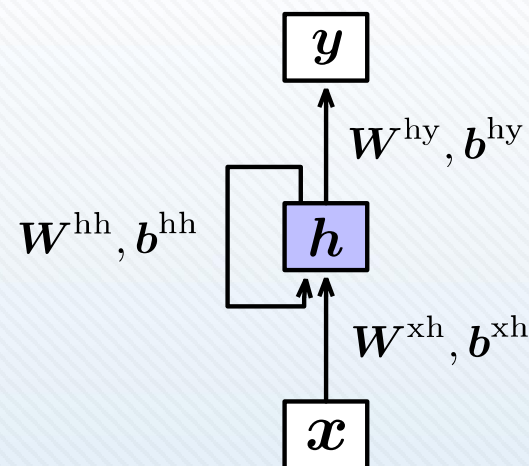
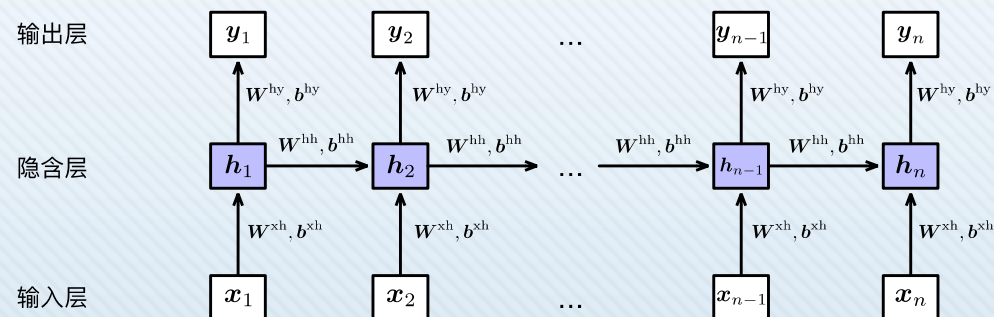


$$h_t = \tanh(W^{xh}x_t + b^{xh} + W^{hh}h_{t-1} + b^{hh})$$

$$y = \text{Softmax}(W^{hy}h_n + b^{hy})$$

❑ 每个时刻的参数共享 (“循环” 的由来)

❑ 每个时刻也可以有相应的输出 (序列标注)



□ 序列过长时，原始的循环神经网络容易导致信息损失

□ 梯度爆炸或梯度消散

□ 新的加性隐状态更新方式

$$u_t = \tanh(W^{xh}x_t + b^{xh} + W^{hh}h_{t-1} + b^{hh})$$

$$h_t = h_{t-1} + u_t$$

long term

short term

□ 相当于直接将  $h_k$  与  $h_t$  ( $k < t$ ) 进行了跨层连接

$$h_t = h_{t-1} + u_t = h_{t-2} + u_{t-1} + u_t = h_k + u_{k+1} + u_{k+2} + \dots + u_t$$

□ 进一步改进

□ 遗忘门：考虑旧状态  $h_{t-1}$  和新状态  $u_t$  的贡献

$$f_t = \sigma(W^{f,xh}x_t + b^{f,xh} + W^{f,hh}h_{t-1} + b^{f,hh})$$

$$h_t = f_t \odot h_{t-1} + (1 - f_t) \odot u_t$$

□ 输入门：独立控制  $h_{t-1}$  和  $u_t$  的贡献

$$i_t = \sigma(W^{i,xh}x_t + b^{i,xh} + W^{i,hh}h_{t-1} + b^{i,hh})$$

$$h_t = f_t \odot h_{t-1} + i_t \odot u_t$$

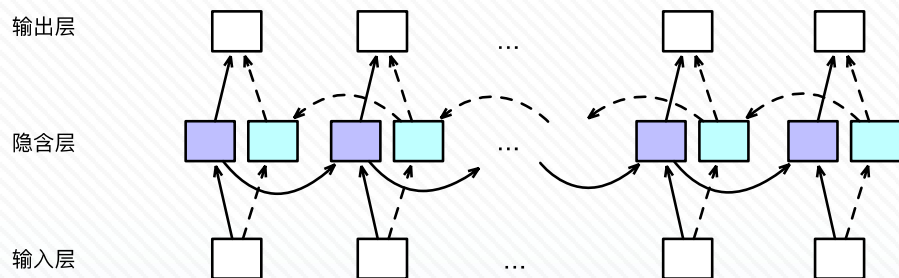
□ 输出门：对输出进行控制

$$o_t = \sigma(W^{o,xh}x_t + b^{o,xh} + W^{o,hh}h_{t-1} + b^{o,hh})$$

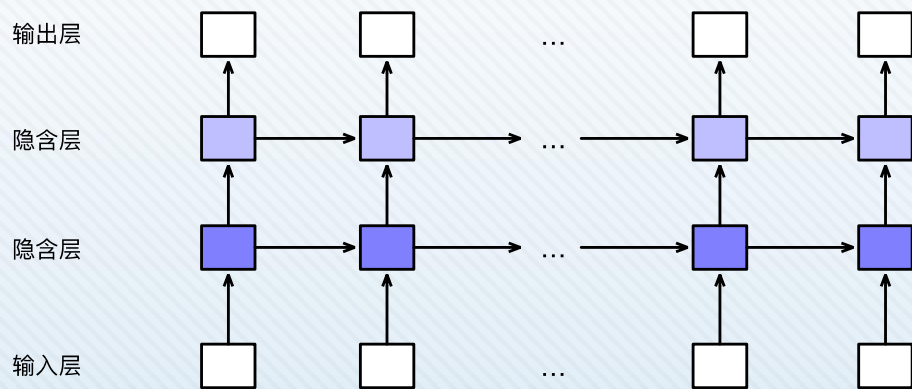
$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$

$$h_t = o_t \odot \tanh(c_t)$$

## 双向循环神经网络

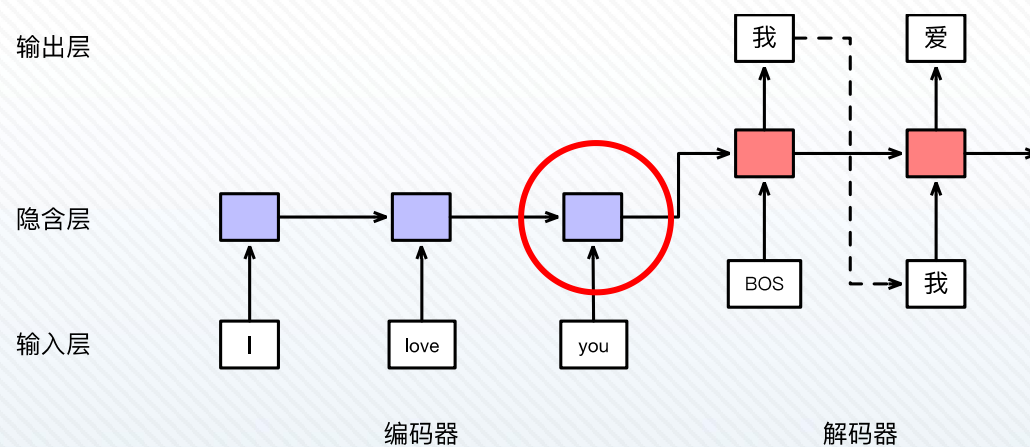


## 堆叠循环神经网络



## 序列到序列模型

- 也称“编码器—解码器”模型
- 机器翻译等多种应用

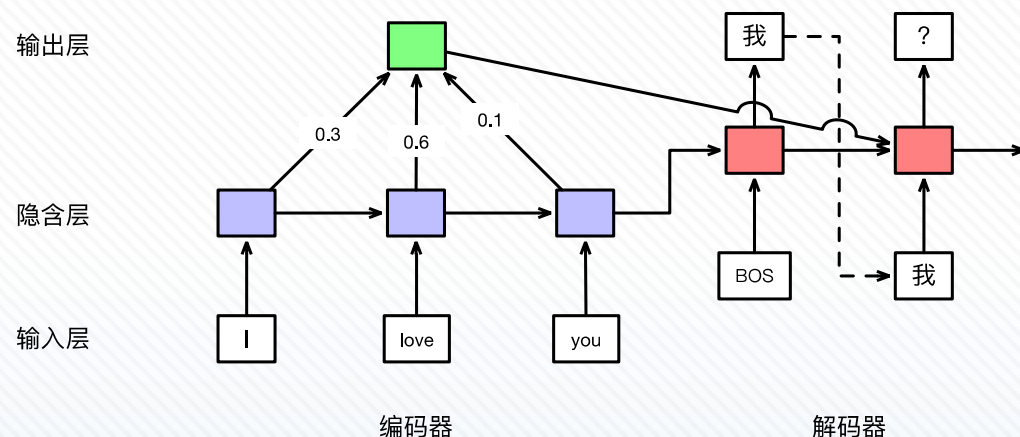


You can't cram the meaning of a whole sentence into a single vector!



Ray Mooney

□ 当前状态除和前一个状态及输入相关外，还应关注原序列的状态

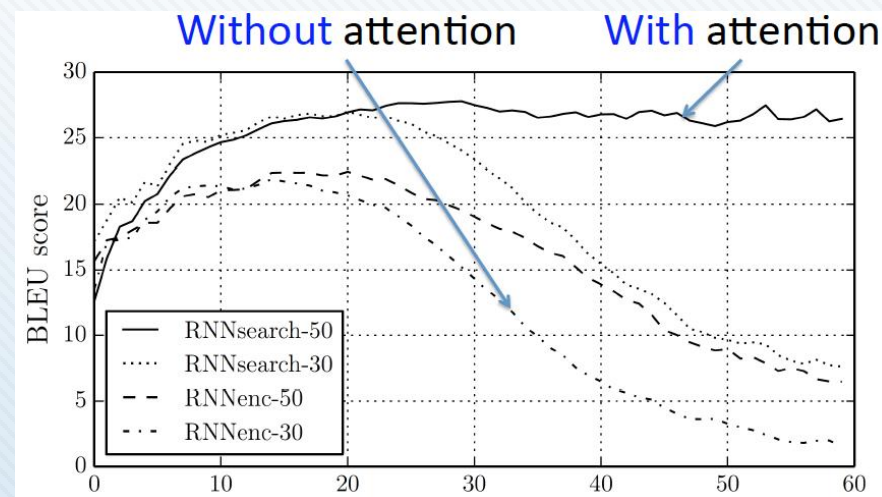
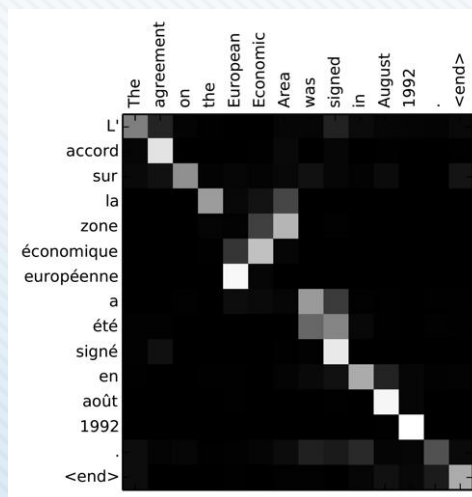


□ 打分公式

$$\hat{\alpha}_s = \text{attn}(\mathbf{h}_s, \mathbf{h}_{t-1})$$

$$\alpha_s = \text{Softmax}(\hat{\alpha})_s$$

$$\text{attn}(\mathbf{q}, \mathbf{k}) = \begin{cases} \mathbf{w}^\top \tanh(\mathbf{W}[\mathbf{q}; \mathbf{k}]) \\ \mathbf{q}^\top \mathbf{W} \mathbf{k} \\ \mathbf{q}^\top \mathbf{k} \\ \frac{\mathbf{q}^\top \mathbf{k}}{\sqrt{d}} \end{cases}$$



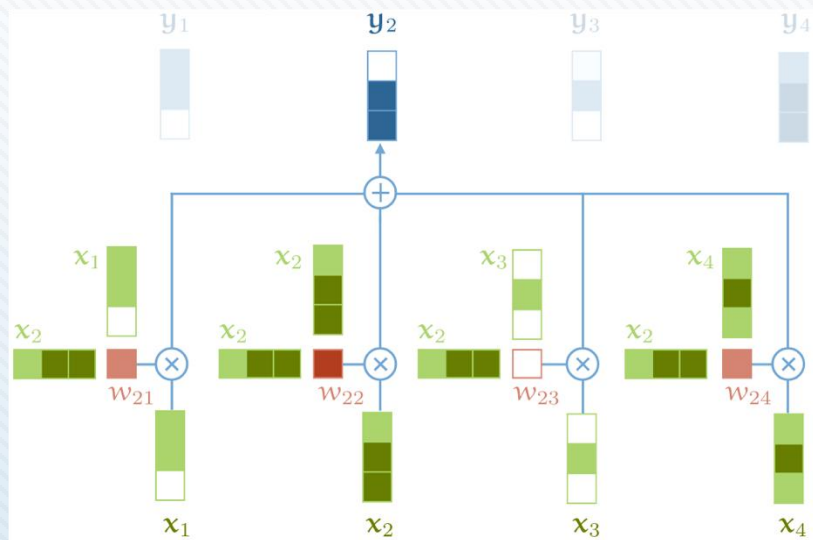
□ 通过某一输入与周围输入的相关性（注意力）来更新该输入

□ “观其伴、知其义”

□ 输入：  $n$  个向量构成的序列  $x_1, x_2, \dots, x_n$

□ 输出： 每个向量对应的新向量  $y_1, y_2, \dots, y_n$

□  $y_i = \sum_{j=1}^n \alpha_{ij} x_j$ ,  $\alpha_{ij}$  为  $x_i$  与  $x_j$  之间的注意力值



```
1 import torch
2 import torch.nn.functional as F
3
4 x = torch.rand(3, 4, 5)
5
6 # torch.bmm is a batched matrix multiplication. It
7 # applies matrix multiplication over batches of
8 # matrices.
9
10 raw_weights = torch.bmm(x, x.transpose(1, 2))
11
12 weights = F.softmax(raw_weights, dim=2)
13
14 y = torch.bmm(weights, x)
15
16 print(y)
```



- Vaswani et al., **Attention** Is All You Need, NIPS 2017
- Transformer的翻译
  - 变压器?
  - 变形金刚?
- 自注意力模型还需要解决的几个问题
  - 没有考虑输入的**位置**信息
  - 输入向量  $x_i$  同时承担三种**角色**, 不易学习
    - 计算注意力权重时的两个向量以及被加权的向量
  - 只考虑了两个输入向量之间的关系, 无法建模多个向量之间的**更复杂关系**
  - 自注意力计算结果互斥, 无法**同时关注**多个输入



## □ 位置嵌入 (Position Embeddings)

- 类似词嵌入，每个绝对位置赋予一个连续、低维、稠密的向量表示
- 向量参数参与模型学习

## □ 位置编码 (Position Encodings)

- 直接将一个整数（位置索引值）映射为一个向量

$$\text{PosEnc}(p, i) = \begin{cases} \sin\left(\frac{p}{10000^{\frac{i}{d}}}\right) & \text{if } i \text{ is even} \\ \cos\left(\frac{p}{10000^{\frac{i-1}{d}}}\right) & \text{else} \end{cases}$$

## □ 输入向量 = 词向量 + 位置嵌入/编码

## 输入向量的三种角色

查询 (Query)

键 (Key)

值 (Value)

## 分别对输入向量进行线性映射

$$q_i = W^q x_i$$

$$k_i = W^k x_i$$

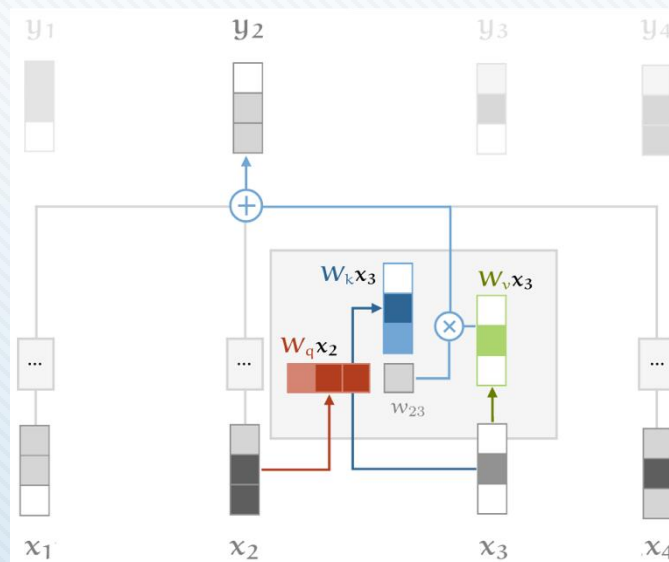
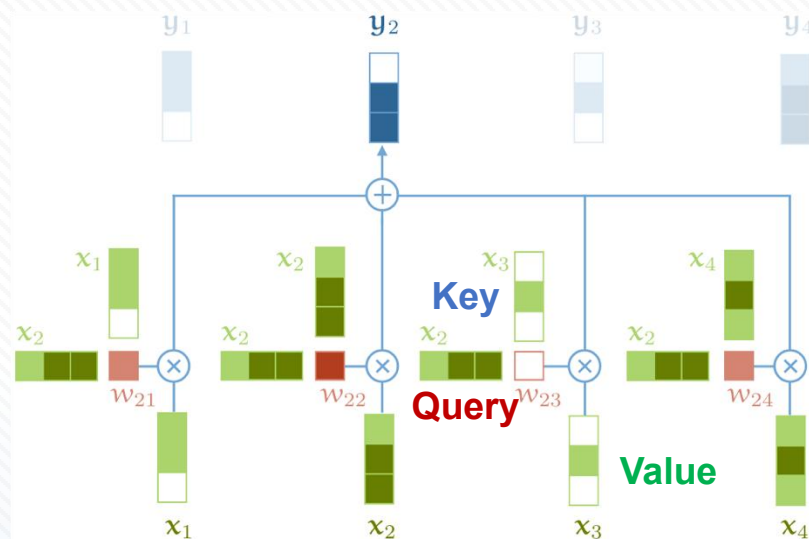
$$v_i = W^v x_i$$

## 新的自注意力模型

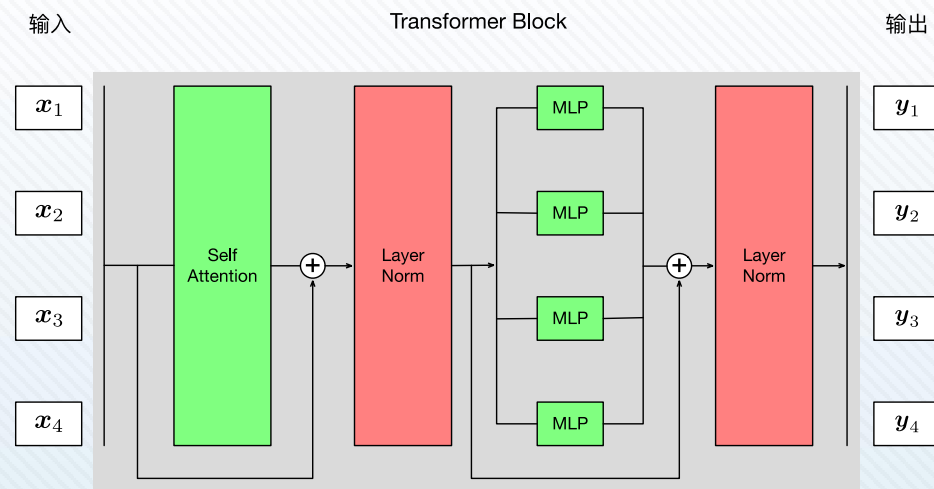
$$y_i = \sum_{j=1}^n \alpha_{ij} v_j$$

$$\alpha_{ij} = \text{Softmax}(\hat{\alpha}_{ij})$$

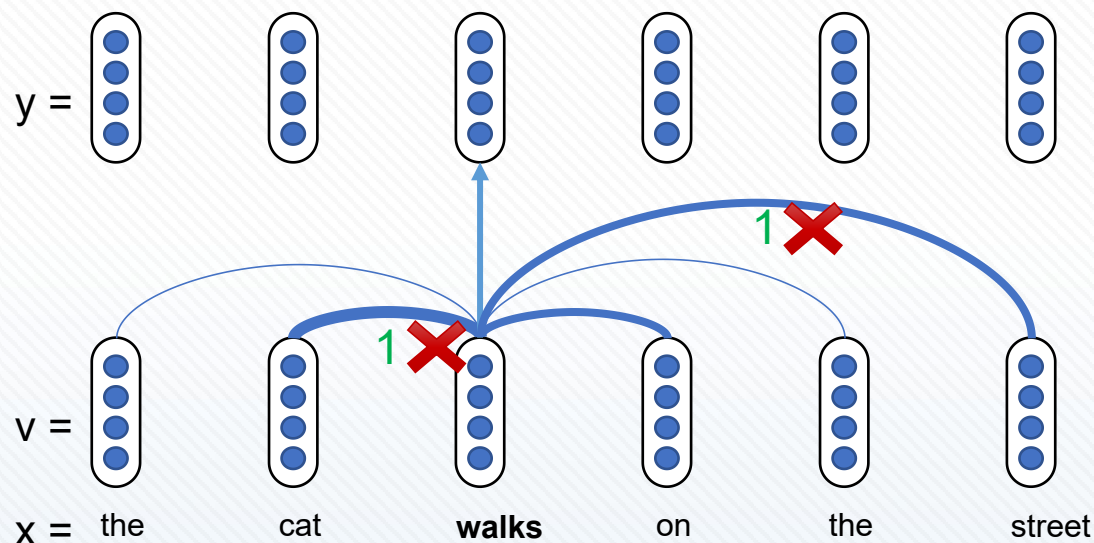
$$\hat{\alpha}_{ij} = \text{attn}(q_i, k_j)$$



- 原始自注意力模型仅考虑了任意两个向量之间的关系
- 如何建模高阶关系？
  - 直接建模高阶关系导致模型复杂度过高
  - 堆叠多层自注意力模型（消息传播机制）
- 增强模型的表示能力——增加非线性
  - 增加非线性的多层感知器模型（MLP）
- 使模型更容易学习
  - 层归一化（Layer Normalization）
  - 残差连接（Residual Connections）

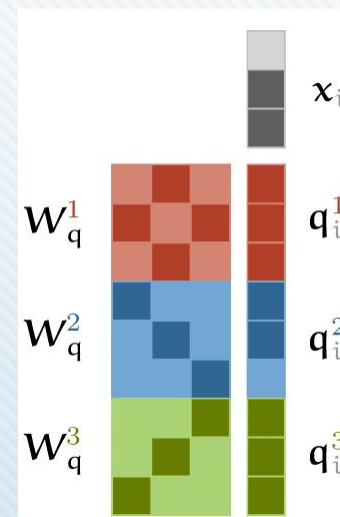


□ 由于Softmax函数的性质，无法使得多个自注意力分数趋近于1



□ 使用多组自注意力模型产生多组不同的注意力结果

- 设置多组输入映射矩阵
- 类似使用多个卷积核提取不同的特征





## 优点

- 直接建模**更长**距离的依赖关系
- **更快**的训练速度（与RNN相比）

## 缺点

- 参数量**过大**导致模型不容易训练
- 需要基于大规模数据**预训练**

□ 寻找一组**优化**的模型参数

□ 又叫做模型**训练**或**学习**

□ 损失函数 (Loss Function)

□ 评估参数好坏的**准则**

□ 为什么不直接使用**准确率**等指标进行评估?

□ 两种常用的损失函数

□ **均方误差** (Mean Squared Error, MSE)

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

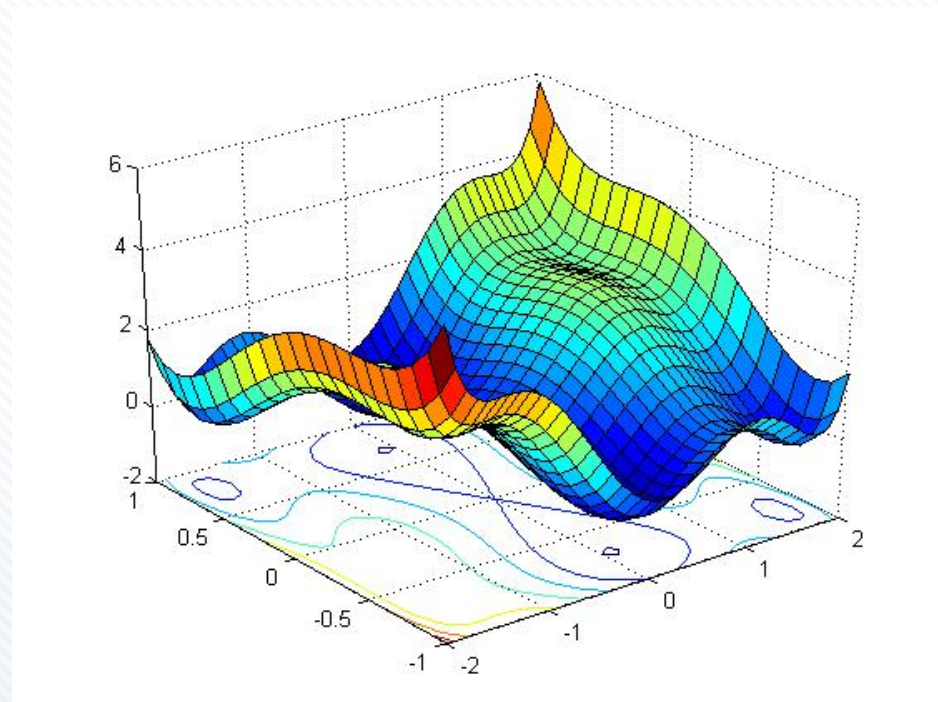
□ **交叉熵** (Cross-Entropy, CE)

$$\text{CE} = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^c y_j^{(i)} \log \hat{y}_j^{(i)}$$



$$\text{CE} = -\frac{1}{m} \sum_{i=1}^m \log \hat{y}_t^{(i)}$$

负对数似然损失  
Negative Log Likelihood, NLL



## ▣ 梯度 (Gradient)

- ▣ 以向量的形式写出的对多元函数各个参数求得的偏导数
- ▣ 是函数值增加最快的方向
- ▣ 沿着梯度相反的方向，更容易找到函数的极小值

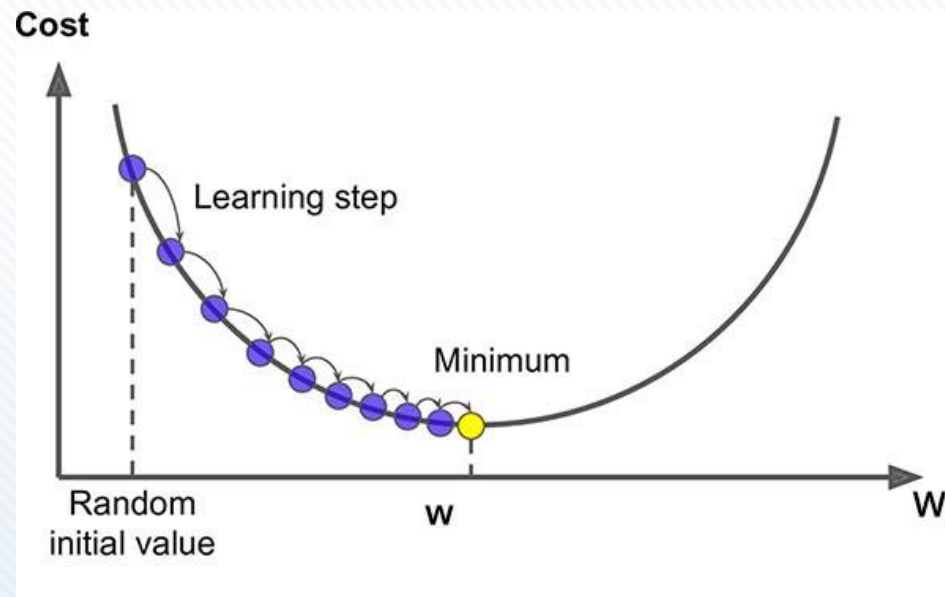
## ▣ 梯度下降算法 (Gradient Descent, GD)

### 算法 4.1 梯度下降算法

**Input:** 学习率  $\alpha$ ; 含有  $m$  个样本的训练数据

**Output:** 优化参数  $\theta$

1. 设置损失函数为  $L(f(\mathbf{x}; \theta), y)$ ;
2. 初始化参数  $\theta$ 。
3. **while** 未达到终止条件 **do**
4.     计算梯度  $\mathbf{g} = \frac{1}{m} \nabla_{\theta} \sum_i^m L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$ ;
5.      $\theta = \theta - \alpha \mathbf{g}$ 。
6. **end**



## ▣ 小批次梯度下降法 (Mini-batch Gradient Descent)

- ▣ 每次随机采样小规模训练数据来估计梯度
- ▣ 提高算法的运行速度

```
# 创建多层感知器模型，输入层大小为2，隐含层大小为5，输出层大小为2（即有两个类别）
model = MLP(input_dim=2, hidden_dim=5, num_class=2)

criterion = nn.NLLLoss() # 当使用log_softmax输出时，需要调用负对数似然损失 (Negative Log Likelihood, NLL)
optimizer = optim.SGD(model.parameters(), lr=0.05) # 使用梯度下降参数优化方法，学习率设置为0.05

for epoch in range(500):
    y_pred = model(x_train) # 调用模型，预测输出结果
    loss = criterion(y_pred, y_train) # 通过对比预测结果与正确的结果，计算损失
    optimizer.zero_grad() # 在调用反向传播算法之前，将优化器的梯度值置为零，否则每次循环的梯度将进行累加
    loss.backward() # 通过反向传播计算参数的梯度
    optimizer.step() # 在优化器中更新参数，不同优化器更新的方法不同，但是调用方式相同
```

 [https://colab.research.google.com/drive/1v-4G7WgiEhV\\_wOuOC\\_2loSIVNzRHQlnP?usp=sharing](https://colab.research.google.com/drive/1v-4G7WgiEhV_wOuOC_2loSIVNzRHQlnP?usp=sharing)



理解语言，认知社会  
以中文技术，助民族复兴



长按二维码，关注哈工大SCIR  
微信号：HIT\_SCIR

# 谢谢！

