自然语言处理中的深度学习基础

杨沐昀

语言技术研究中心
哈尔滨工业大学

感知器模型

- 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

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

Logistic回归模型

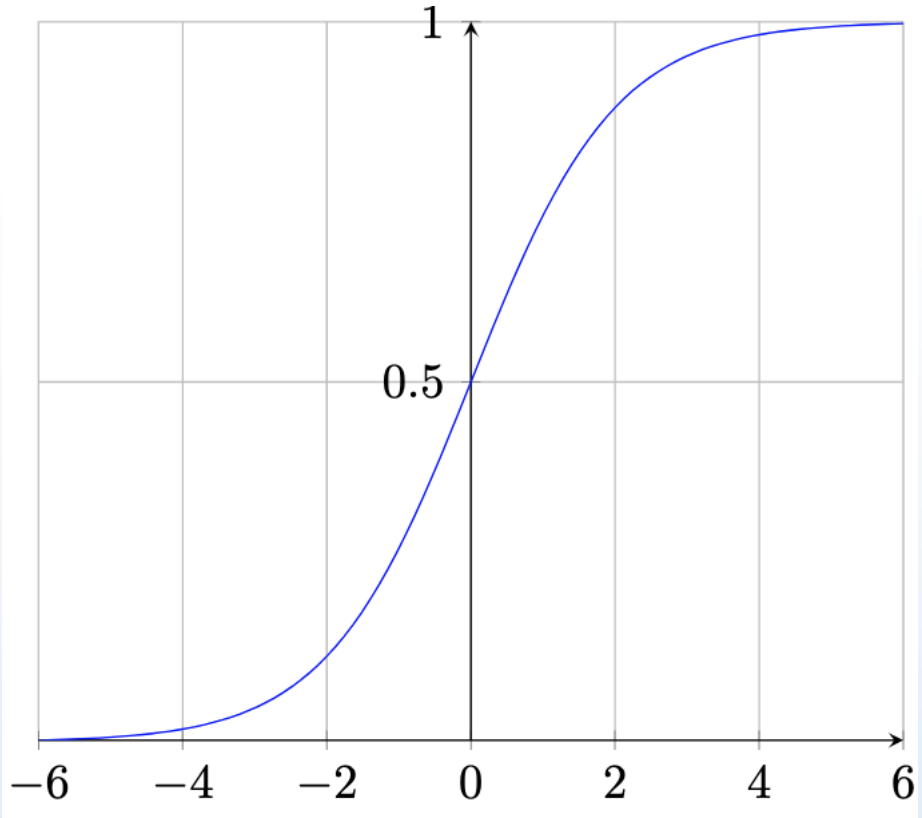
- 线性回归（Linear Regression）
  - $y = \sum_i w_i x_i + b$
- Logistic回归（Logistic Regression）
  - Logistic函数

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

  - 设$z = \sum_i w_i x_i + b$
  - Sigmoid函数（Logistic函数的特例）

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

  - 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



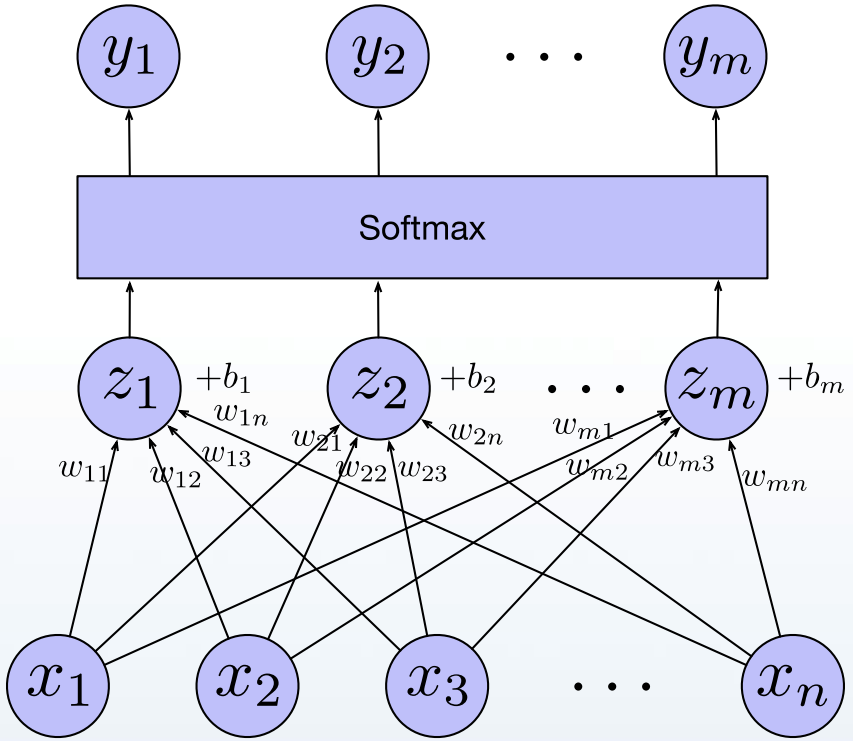
Sigmoid函数示意图

Softmax回归

- 使用Softmax回归处理多元分类
- Softmax函数

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

  - 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



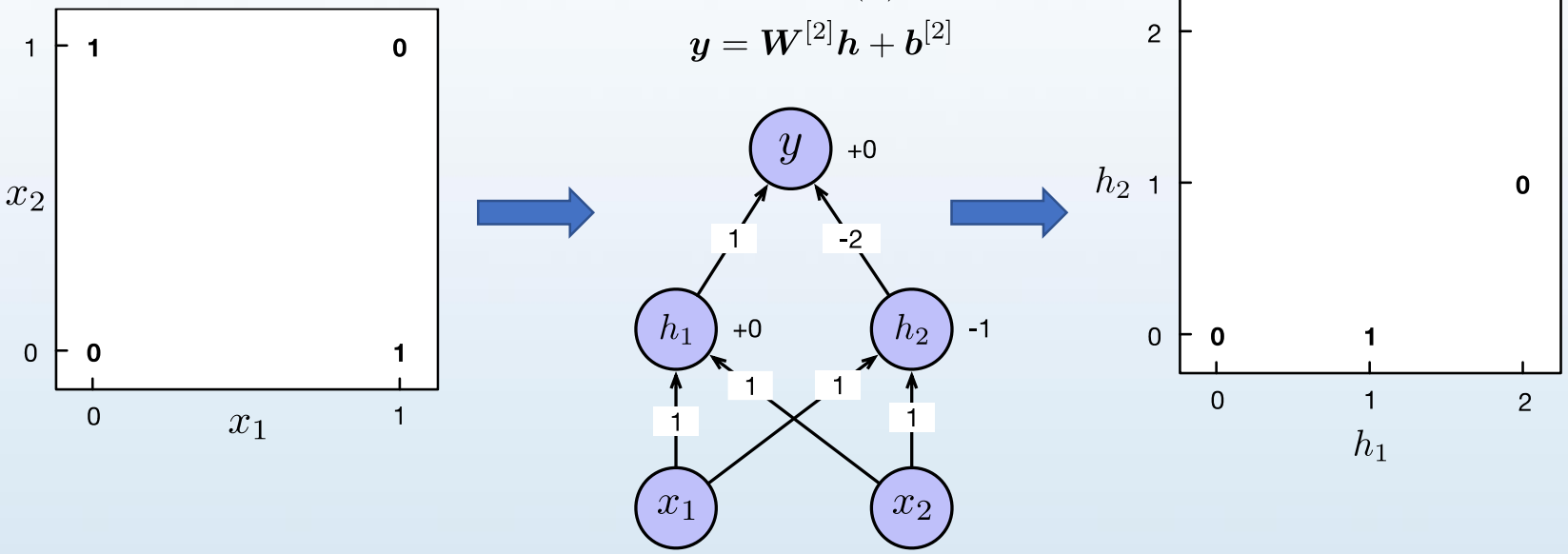
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \text{Softmax} \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n + b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n + b_2 \\ \vdots \\ w_{m1}x_1 + w_{m2}x_2 + \cdots + w_{mn}x_n + b_m \end{pmatrix}$$

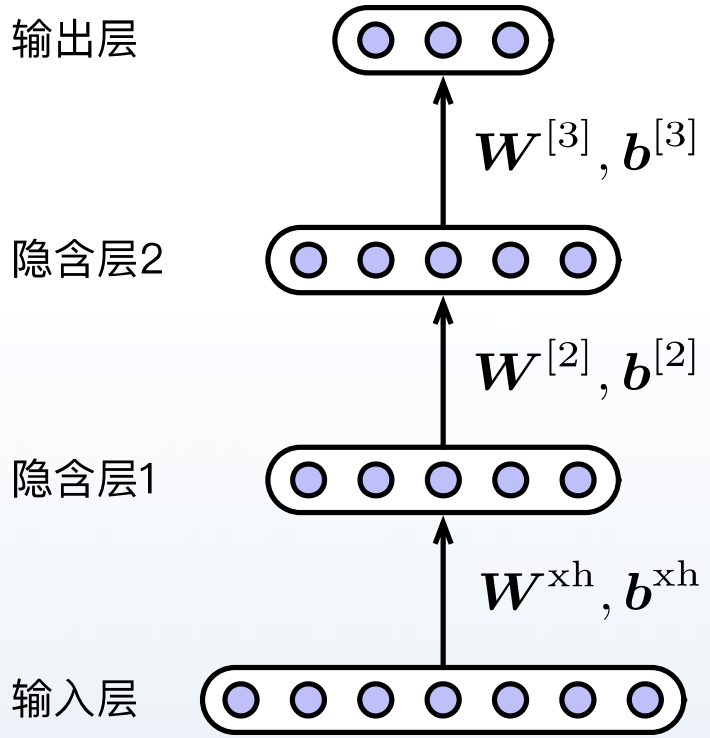
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \text{Softmax} \left(\begin{bmatrix} w_{11}, w_{12}, \cdots, w_{1n} \\ w_{21}, w_{22}, \cdots, w_{2n} \\ \vdots \\ w_{m1}, w_{m2}, \cdots, w_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix}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

$$\boldsymbol{y} = \text{Softmax}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

多层感知器

- 多层感知器（Multi-layer Perceptron，MLP）
  - 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
- 使用MLP解决异或（XOR）问题





多层感知器PyTorch实现

- 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>)
```

多层感知器PyTorch实现

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
```

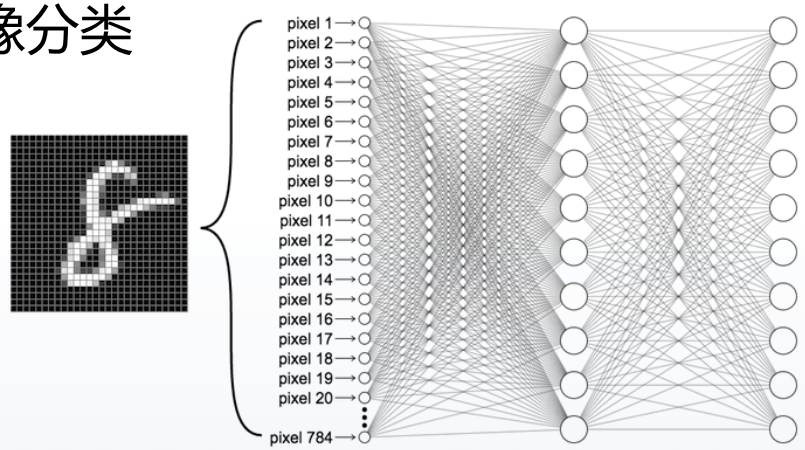
```
mlp = MLP(input_dim=4, hidden_dim=5, num_class=2)
inputs = torch.rand(3, 4) # 输入形状为(3, 4)的张量，其中3表示有3个输入，4表示每个
    输入的维度
probs = mlp(inputs) # 自动调用forward函数
print(probs) # 输出3个输入对应输出的概率
```

https://colab.research.google.com/drive/1EMJJffsmlSlqO3KwuG5WlP-IxyNVXTrK?usp=sharing

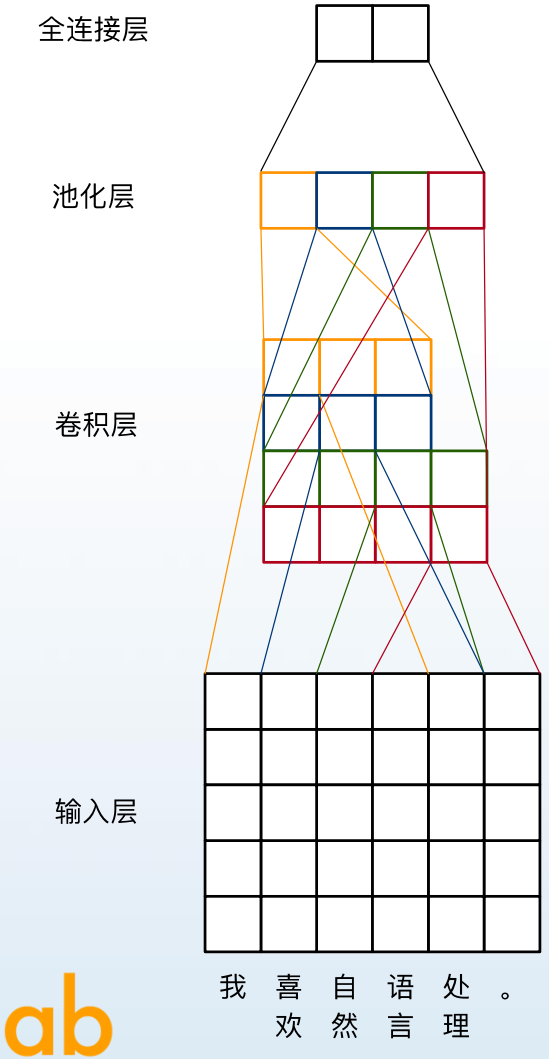
colab

卷积神经网络

- 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）

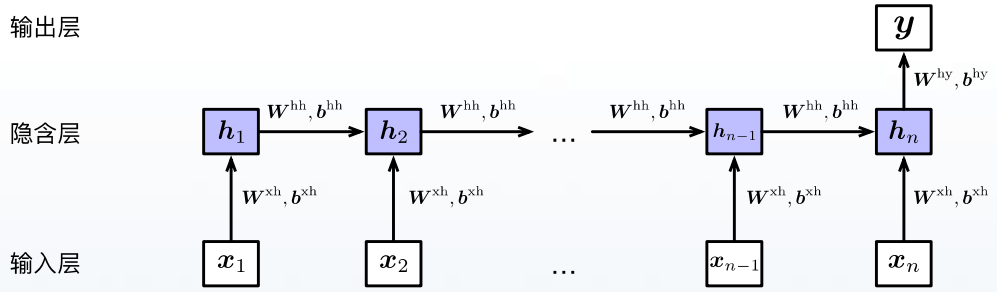


colab

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

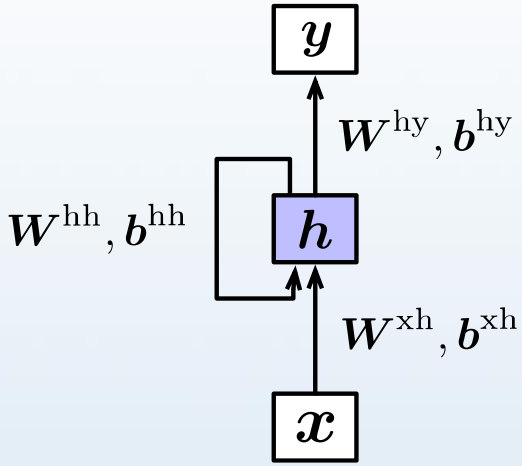
循环神经网络

- 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

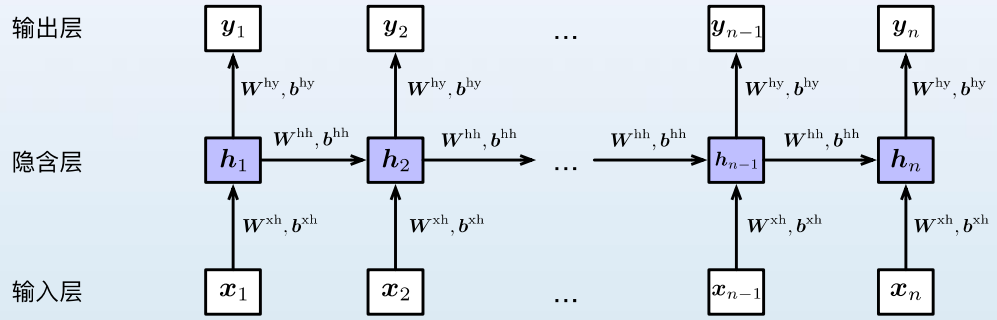


$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}^{\mathrm{xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{xh}} + \boldsymbol{W}^{\mathrm{hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{hh}})$$

$$\boldsymbol{y} = \mathrm{Softmax}(\boldsymbol{W}^{\mathrm{hy}}\boldsymbol{h}_n + \boldsymbol{b}^{\mathrm{hy}})$$



- 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）



colab

https://colab.research.google.com/drive/1mTeml7lAA9B2pS-O9yWm1_M7LiyHPPUJ?usp=sharing

长短时记忆网络（LSTM）

- 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

$$\boldsymbol{u}_t = \tanh(\boldsymbol{W}^{\mathrm{xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{xh}} + \boldsymbol{W}^{\mathrm{hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{hh}})$$

$$\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \boldsymbol{u}_t$$

long term　　short term

  - 相当于直接将 $\boldsymbol{h}_k$ 与 $\boldsymbol{h}_t$（$k < t$）进行了跨层连接
    - $\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \boldsymbol{u}_t = \boldsymbol{h}_{t-2} + \boldsymbol{u}_{t-1} + \boldsymbol{u}_t = \boldsymbol{h}_k + \boldsymbol{u}_{k+1} + \boldsymbol{u}_{k+2} + \cdots + \boldsymbol{u}_t$
- 进一步改进
  - 遗忘门：考虑旧状态 $\boldsymbol{h}_{t-1}$ 和新状态 $\boldsymbol{u}_t$ 的贡献

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}^{\mathrm{f,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{f,xh}} + \boldsymbol{W}^{\mathrm{f,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{f,hh}})$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{f}_t) \odot \boldsymbol{u}_t$$

  - 输入门：独立控制 $\boldsymbol{h}_{t-1}$ 和 $\boldsymbol{u}_t$ 的贡献

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}^{\mathrm{i,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{i,xh}} + \boldsymbol{W}^{\mathrm{i,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{i,hh}})$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{u}_t$$

  - 输出门：对输出进行控制

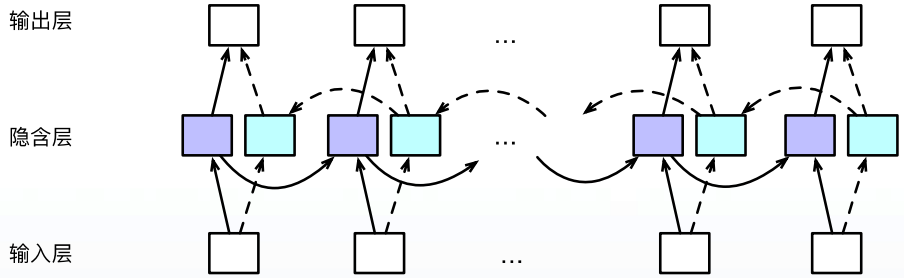
$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}^{\mathrm{o,xh}}\boldsymbol{x}_t + \boldsymbol{b}^{\mathrm{o,xh}} + \boldsymbol{W}^{\mathrm{o,hh}}\boldsymbol{h}_{t-1} + \boldsymbol{b}^{\mathrm{o,hh}})$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{u}_t$$

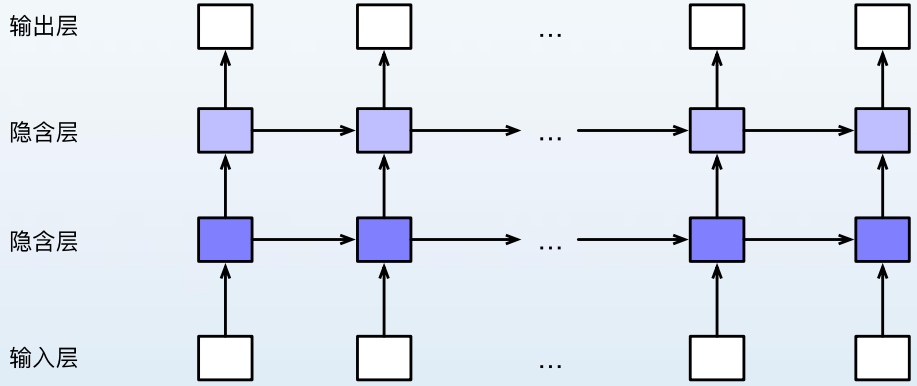
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$$

循环神经网络的应用

双向循环神经网络

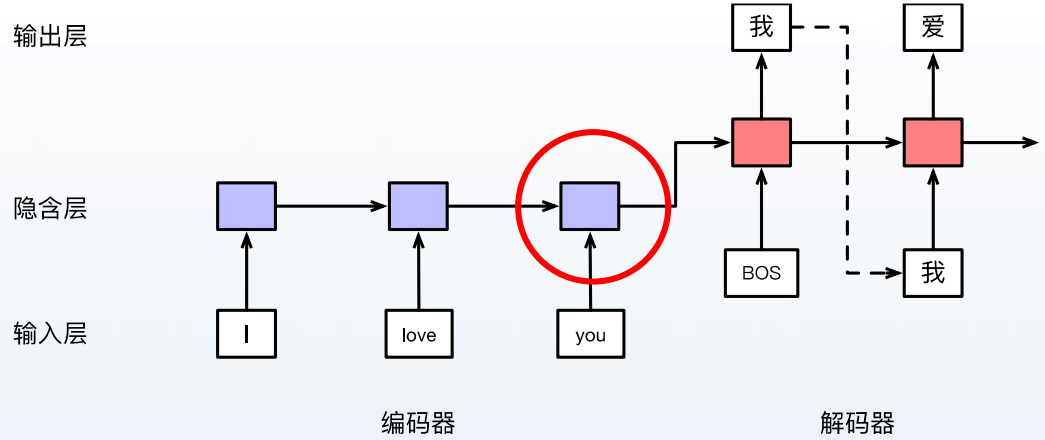


堆叠循环神经网络



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

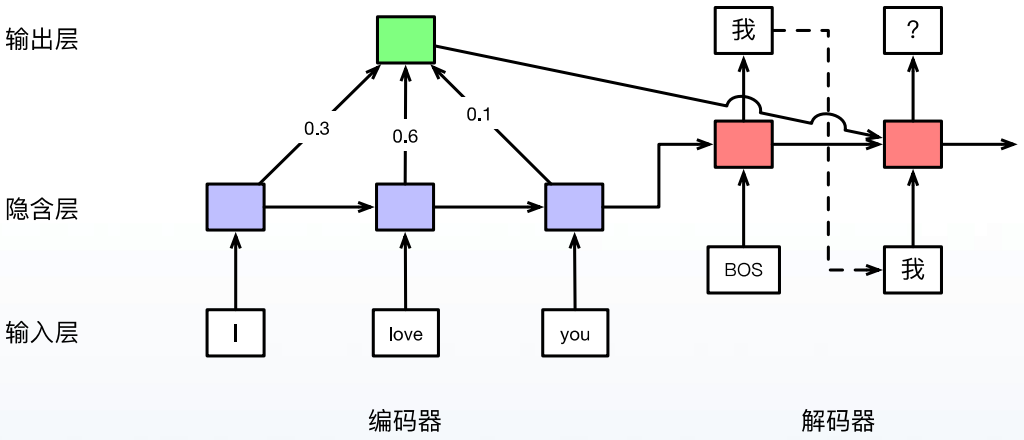


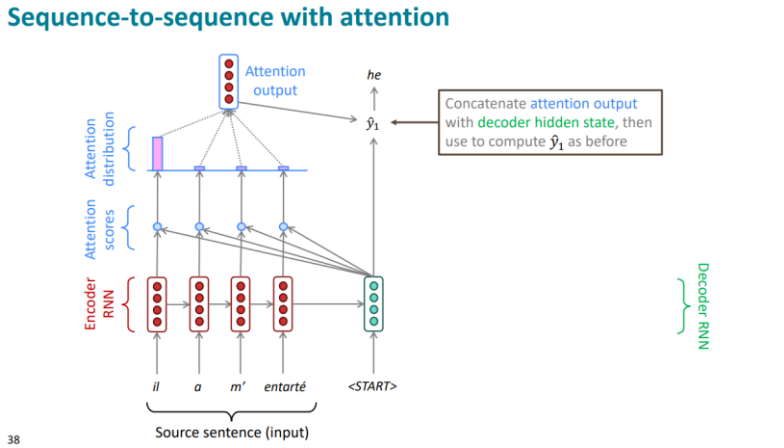
You can’t cram the meaning of a whole %&!$ing sentence into a single $&!*ing vector!

Ray Mooney

注意力机制（Attention Mechanism）

- 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



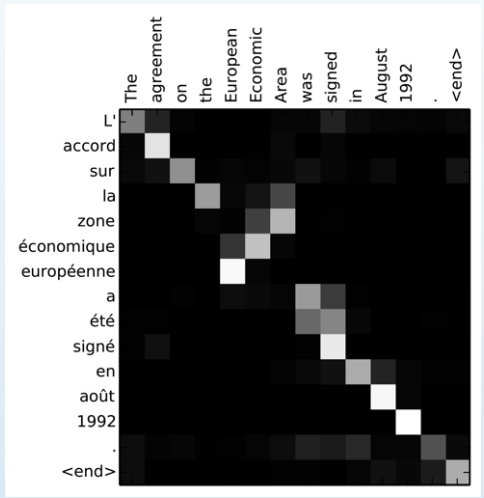


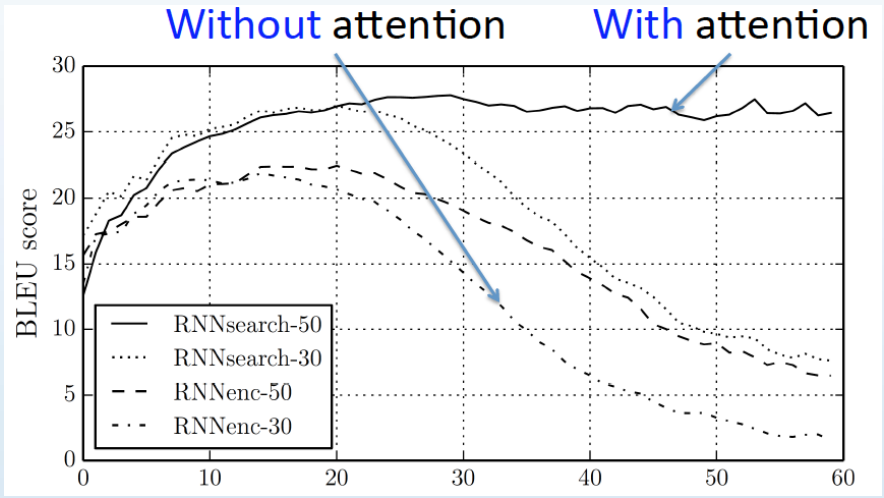
- 打分公式

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

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

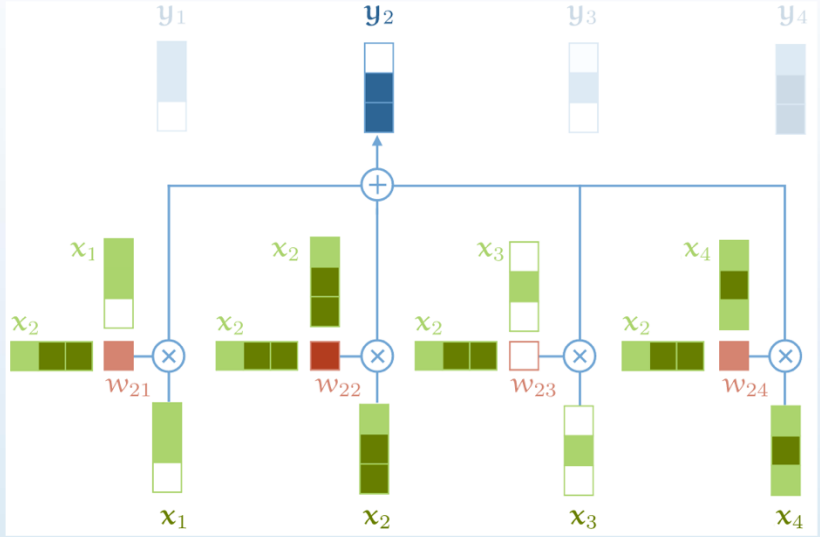
$$\text{attn}(\boldsymbol{q}, \boldsymbol{k}) = \begin{cases} \boldsymbol{w}^\top \tanh(\boldsymbol{W}[\boldsymbol{q}; \boldsymbol{k}]) \\ \boldsymbol{q}^\top \boldsymbol{W} \boldsymbol{k} \\ \boldsymbol{q}^\top \boldsymbol{k} \\ \frac{\boldsymbol{q}^\top \boldsymbol{k}}{\sqrt{d}} \end{cases}$$





自注意力模型

- 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$\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$
- 输出：每个向量对应的新向量$\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n$
- $\boldsymbol{y}_i = \sum_{j=1}^{n} \alpha_{ij} \boldsymbol{x}_j$，　α_{ij}为 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$之间的注意力值




```
>>> import torch
>>> import torch.nn.functional as F
>>> # 生成一个(2, 3, 4)的张量，第1维表示批次大小，第2维表示序列长度，第3维表示词
    向量维度
>>> x = torch.randn(2, 3, 4)
>>> # @是矩阵乘法运算符，保持参与运算张量前面的维度不变，最后两维进行矩阵乘法运算
>>> attn = x @ x.transpose(1, 2)
>>> attn = F.softmax(attn, dim=-1)
>>> y = attn @ x
>>> print(y, y.shape)
tensor([[[-0.0933,  0.1734, -0.4230, -0.2408],
         [-0.0098,  0.2139, -0.4638, -0.2008],
         [ 0.6867,  1.4132,  1.0766,  1.7611]],

        [[-0.1461,  0.0825,  0.7220, -1.6230],
         [ 1.4327, -0.3362,  0.3039, -0.6506],
         [-0.0056,  0.1133,  1.3207, -0.0676]]]) torch.Size([2, 3, 4])
```

Transformer

- 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
  - 自注意力计算结果互斥，无法同时关注多个输入
  - 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

融入位置信息

- 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

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

- 输入向量 = 词向量 + 位置嵌入/编码

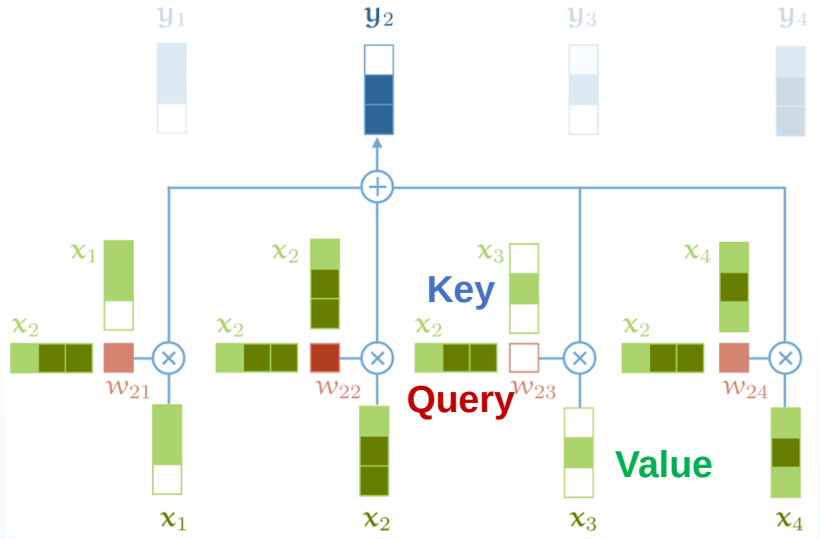
赋予输入向量角色信息

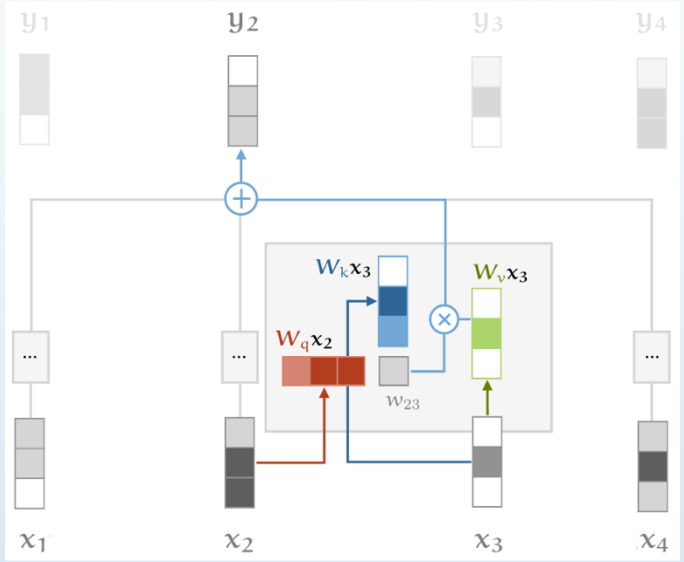
- 输入向量的三种角色
  - 查询（Query）
  - 键（Key）
  - 值（Value）
- 分别对输入向量进行线性映射
  - $\boldsymbol{q}_i = \boldsymbol{W}^{\mathrm{q}}\boldsymbol{x}_i$
  - $\boldsymbol{k}_i = \boldsymbol{W}^{\mathrm{k}}\boldsymbol{x}_i$
  - $\boldsymbol{v}_i = \boldsymbol{W}^{\mathrm{v}}\boldsymbol{x}_i$
- 新的自注意力模型

$$\boldsymbol{y}_i = \sum_{j=1}^{n} \alpha_{ij}\boldsymbol{v}_j$$

$$\alpha_{ij} = \mathrm{Softmax}(\hat{\boldsymbol{\alpha}}_i)_j$$

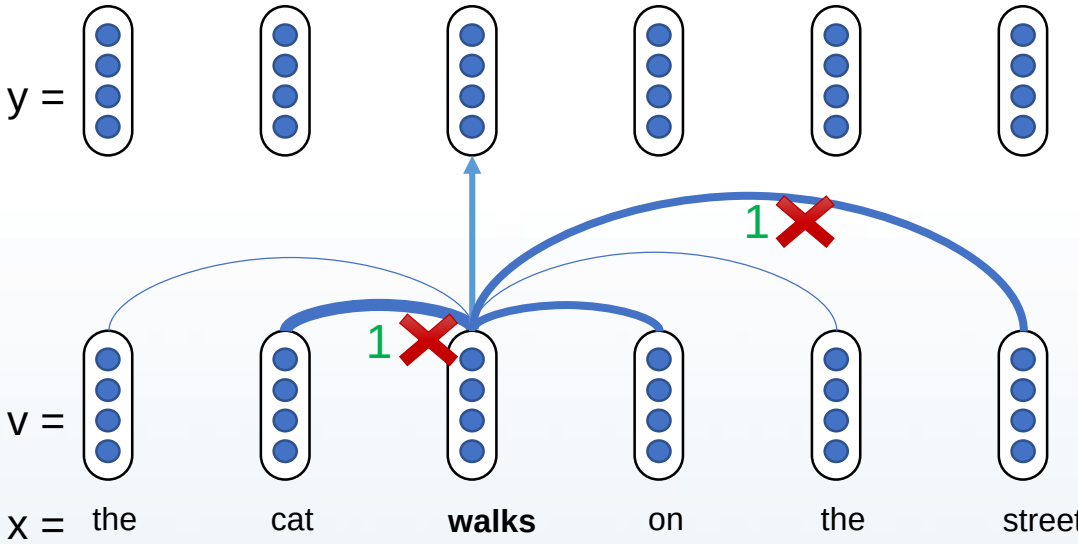
$$\hat{\alpha}_{ij} = \mathrm{attn}(\boldsymbol{q}_i, \boldsymbol{k}_j)$$



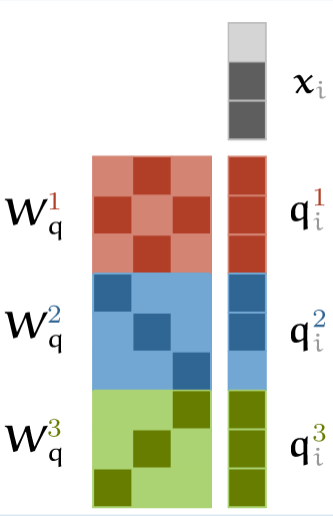


多头自注意力

- 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



多头自注意力的实现

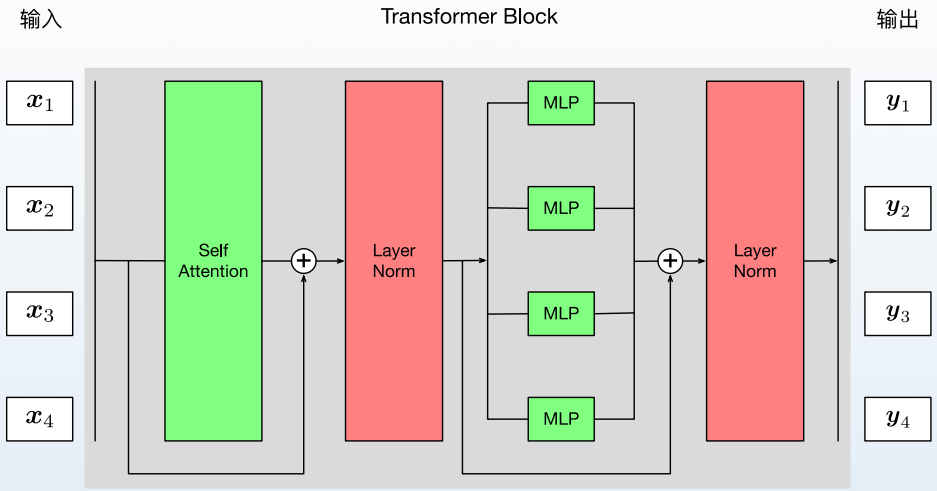
```python
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.proj = nn.Linear(config.n_embd, config.n_embd * 3)

    def forward(self, x):
        B, T, C = x.size() # batch_size, seq_len, n_embd

        # 获得batch中每个输入的q, k, v, 并将q, k, v分解为n_head组
        q, k, v = self.proj(x).chunk(3, dim=-1)
        k = k.view(B, T, self.config.n_head, -1).transpose(1, 2)
        q = q.view(B, T, self.config.n_head, -1).transpose(1, 2)
        v = v.view(B, T, self.config.n_head, -1).transpose(1, 2)

        # 计算自注意力:
        # (B, n_head, T, hs) x (B, n_head, hs, T) -> (B, n_head, T, T)
        attn = (q @ k.transpose(-2, -1)) / (k.size(-1) ** 0.5)
        attn = F.softmax(attn, dim=-1)
        y = attn @ v
        y = y.transpose(1, 2).reshape(B, T, C)
        return y
```

```python
@dataclass
class Config:
    batch_size: int = 2
    seq_len: int = 3
    n_embd: int = 4
    n_head: int = 2
```

```python
if __name__ == '__main__':
    config = Config()
    x = torch.randn(config.batch_size, config.seq_len, config.n_embd)
    self_attn = MultiHeadSelfAttention(config)
    y = self_attn(x)
    print(y, y.shape)
```

多层自注意力

- 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）



多层自注意力的实现

```python
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.fc1  = nn.Linear(config.n_embd, 4 * config.n_embd)
        self.gelu = nn.GELU()
        self.fc2  = nn.Linear(4 * config.n_embd, config.n_embd)

    def forward(self, x):
        x = self.fc1(x)
        x = self.gelu(x)
        x = self.fc2(x)
        return x
```

```python
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = MultiHeadSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = MLP(config)

    def forward(self, x):
        x = self.ln_1(x + self.attn(x))
        x = self.ln_2(x + self.mlp(x))
        return x
```

```python
class Transformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.blocks = nn.ModuleList([Block(config) for _ in range(config.
    n_layer)])

    def forward(self, x):
        for block in self.blocks:
            x = block(x)
        return x
```

Transformer的优缺点

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

神经网络模型的训练

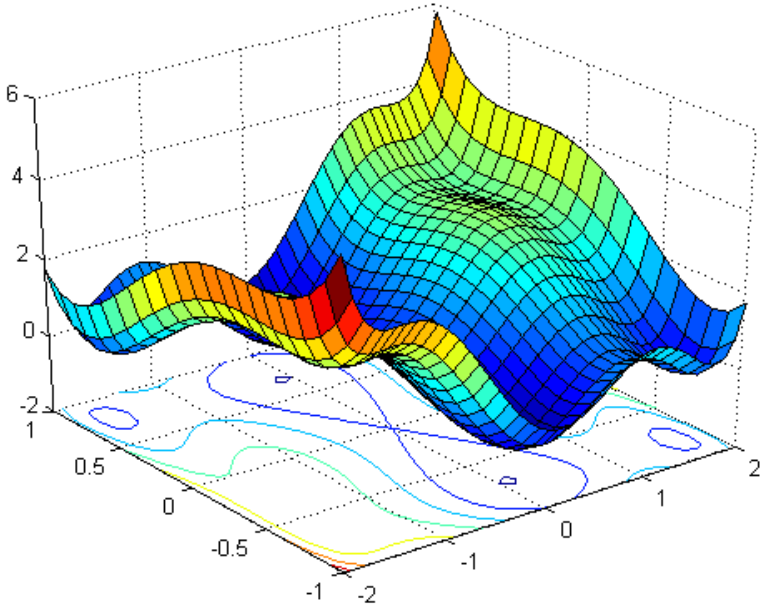
- 寻找一组优化的模型参数
  - 又叫做模型训练或学习
- 损失函数（Loss Function）
  - 评估参数好坏的准则
  - 为什么不直接使用准确率等指标进行评估?
- 两种常用的损失函数
  - 均方误差（Mean Squared Error，MSE）

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

  - 交叉熵（Cross-Entropy，CE）

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

负对数似然损失
Negative Log Likelihood，NLL



梯度下降

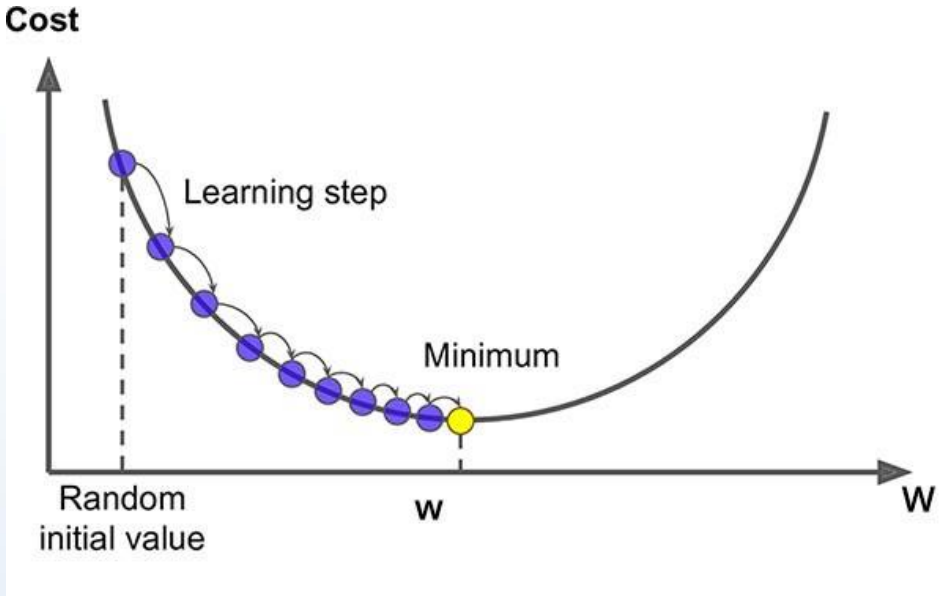
- 梯度（Gradient）
  - 以向量的形式写出的对多元函数各个参数求得的偏导数
  - 是函数值增加最快的方向
  - 沿着梯度相反的方向，更加容易找到函数的极小值
- 梯度下降算法（Gradient Descent，GD）

算法 4.1 梯度下降算法

Input: 学习率 α；含有 m 个样本的训练数据

Output: 优化参数 $\boldsymbol{\theta}$

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



- 小批次梯度下降法（Mini-batch Gradient Descent）
  - 每次随机采样小规模的训练数据来估计梯度
  - 提高算法的运行速度

PyTorch代码示例

```
# 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
```

colab https://colab.research.google.com/drive/1v-4G7WgiEhV_wOuOC_2IoSIVNzRHQlnP?usp=sharing

谢 谢!

语言技术紫丁香

微信扫描二维码，关注我的公众号