多智能体强化学习实训
Multi-agent Reinforcement Learning

Lecture 3:
Introduction to Deep Learning

教师：张寅 zhangyin98@zju.edu.cn

助教：邓悦 devindeng@zju.edu.cn

王子瑞 ziseoiwong@zju.edu.cn

李奕澄 yichengli@zju.edu.cn

浙江大学计算机学院

课程内容安排

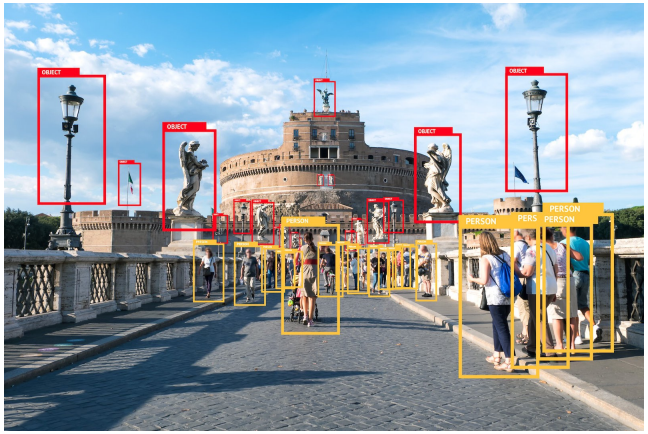
时间	上午	下午
7月1日	第一讲 动态规划理论，Bellman公式，策略评估与策略优化理论	配置强化学习环境，安装gym，完成案例CartPole环境的运行、完成腾讯koh环境部署
7月2日	第二讲 基于表格的Q-learning算法，SARSA算法，eligibility trace的应用	基于gym，测试Q-learning和SARSA算法的表现与区别
7月3日	第三讲 深度学习基础，包含损失函数、梯度回传等知识	安装pytorch，并完成基础的回归、分类任务的网络训练
7月4 日	第四讲 深度强化学习I：基于价值的DQN与基于DQN的算法	基于gym，完成DQN算法实现，并在Freeway等环境测试
7月5日	第五讲 深度强化学习II：基于策略的Reinforce算法、AC算法的讲解、以及A2C、A3C等算法	基于gym，完成基础AC算法的实现
7月8日	第六讲 深度强化学习III：先进算法TRPO、PPO、DDPG等算法原理与实现	基于gym，实现PPO算法
7月9日	第七讲 多智能体强化学习I：基于价值的QMIX算法以及改进算法	在koh 1v1环境中实现PPO算法
7月10日	第八讲 腾讯专家交流	在koh 3v3环境中实现QMIX算法
7月11日	第九讲 多智能体强化学习II：基于策略的MADDPG、MAPPO算法实现	在koh 3v3环境中实现MAPPO算法
7月12日	第十讲 多智能体强化学习研究展望	大作业答疑

课程大纲

- **神经网络及训练**
- 深度神经网络架构
- 大模型和表示收敛
- PyTorch使用

深度学习

深度学习（Deep Learning）是机器学习的分支，是一种以人工神经网络为架构，对资料进行表征学习的算法。



深度学习

深度学习 vs 传统机器学习

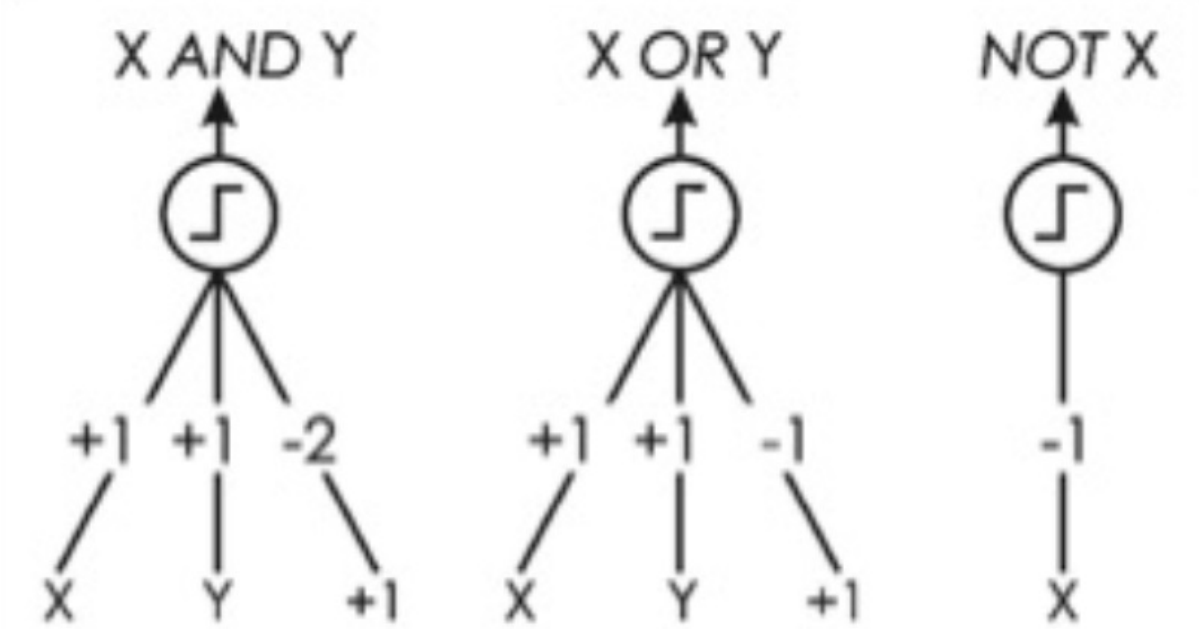
- 传统机器学习
  - 往往需要人工设计特征，然后通过模型进行学习和预测。
  - 通常适用于结构化数据，如表格数据。

- 深度学习
  - 自动提取特征，通过多层神经网络进行端到端学习。
  - 擅长处理非结构化数据，如图像、音频、文本等。

神经网络简史

1943

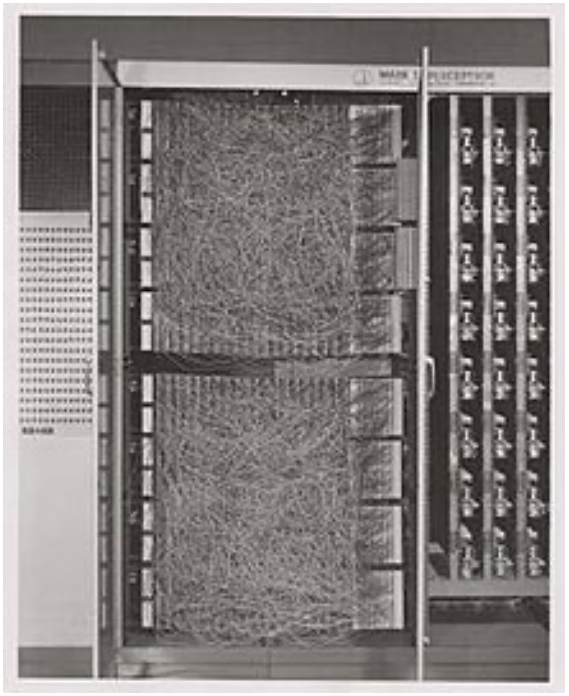
Walter Pitts & Warren McCulloch
基于人脑神经网络的计算机模型



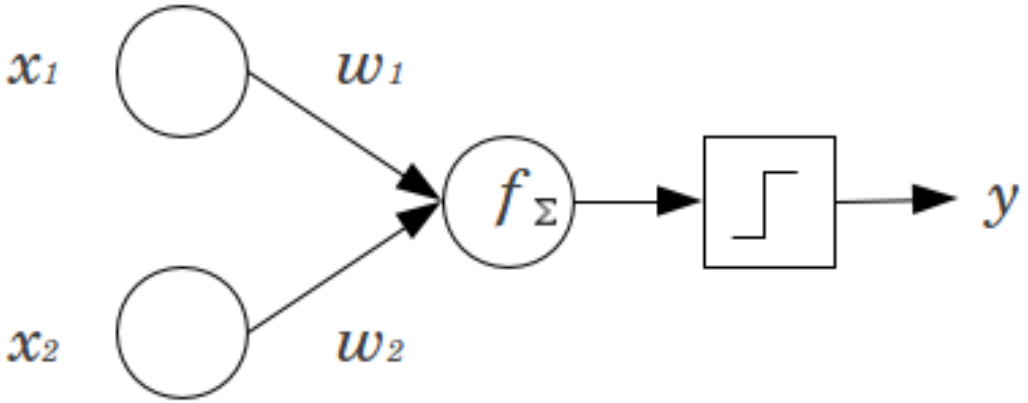
神经网络简史

1957

Frank Rosenblatt
感知机的硬件实现



Mark I Perceptron Machine

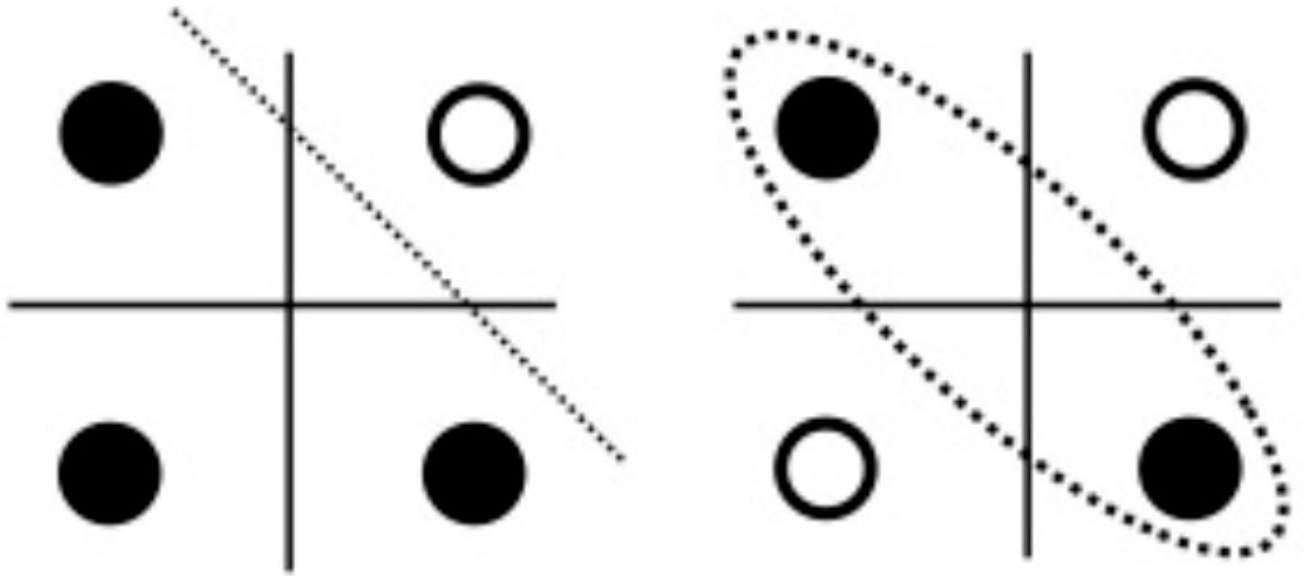


神经网络简史

1969

Marvin Minsky & Syemour Papert

XOR问题



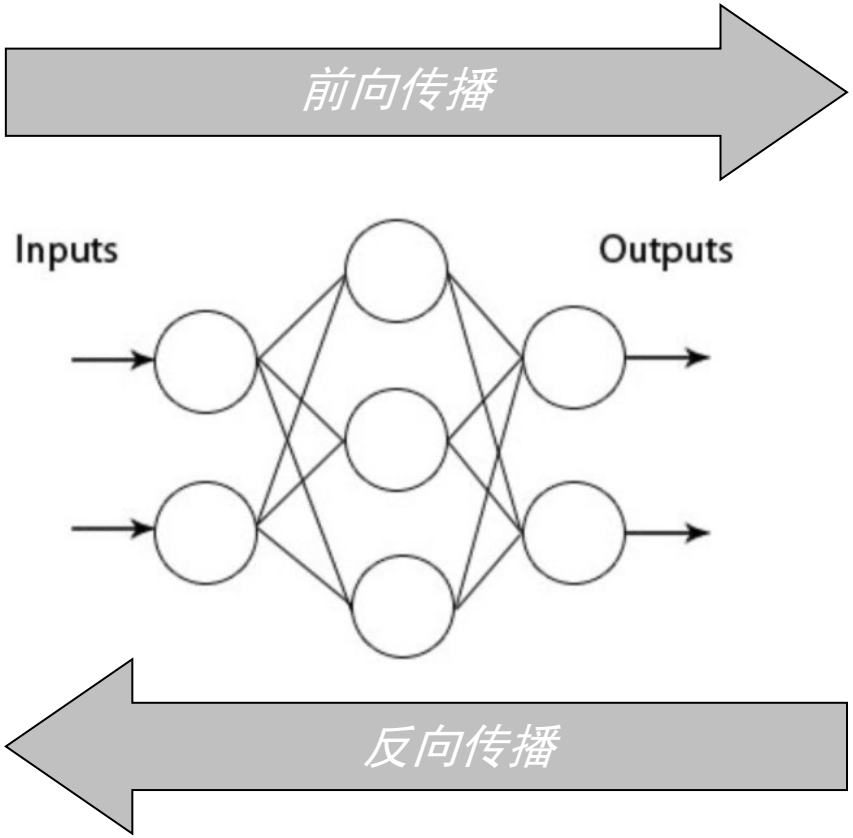
AI Winter

神经网络简史

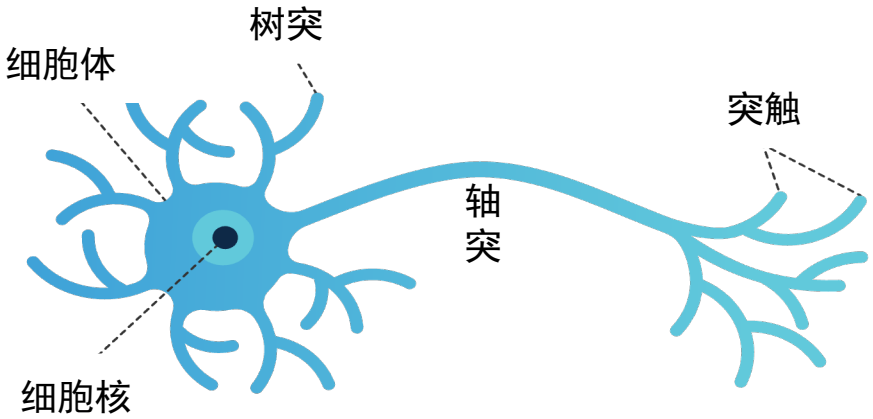
1986

David Rumelhart, Geoffrey Hinton &
Ronald Williams

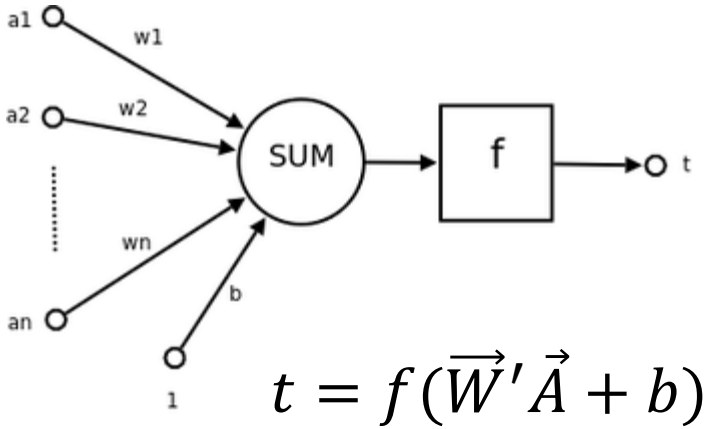
反向传播



神经元 & 神经网络



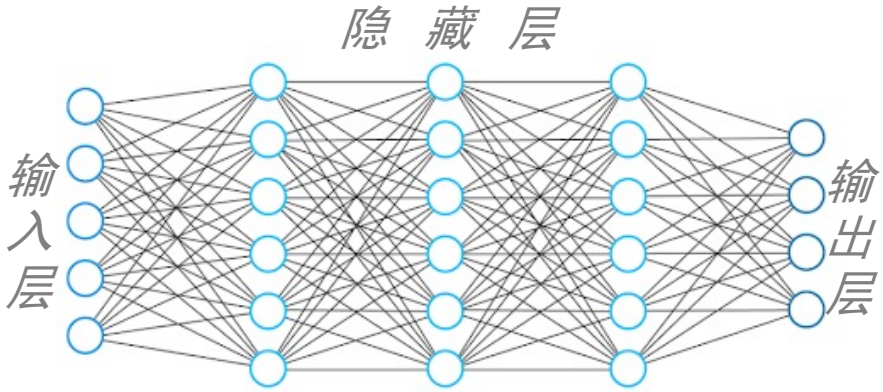
神经元细胞



$$t = f(\vec{W}'\vec{A} + b)$$

人工神经元结构

- 神经元是神经网络的基本单元。
- 神经元一般包括：
  - 输入向量 $\vec{A} = (a1, a2, \ldots, an)$
  - 权重向量 $\vec{W} = (w1, w2, .., wn)$
  - 偏置 b
  - 激活函数 f
  - 输出 t
- 神经元以一定方式连接，形成了神经网络



一个简单的神经网络

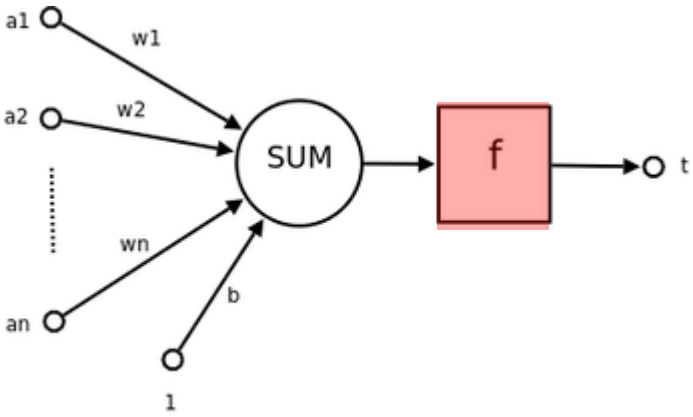
感知机(Perceptron)

感知机是线性分类模型。

在感知机中，激活函数$f = \text{sign}$，即

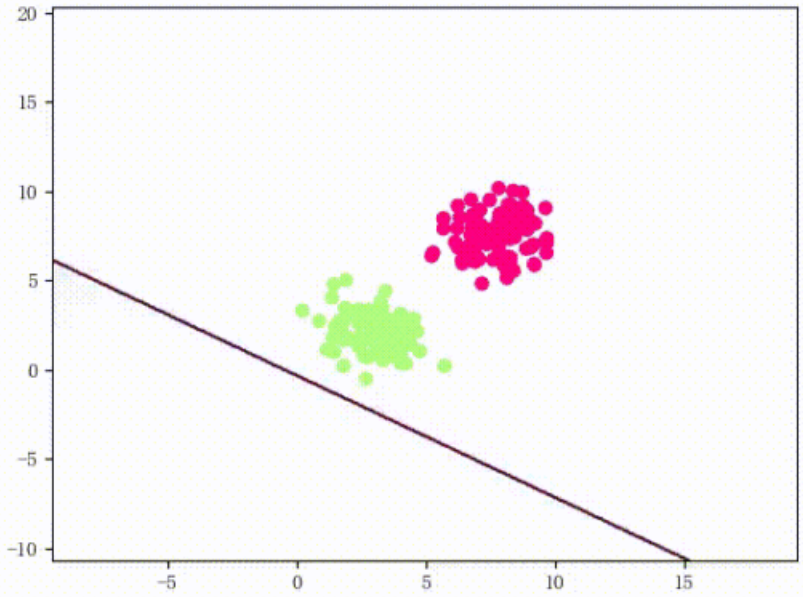
$$t = \text{sign}(\vec{W}'\vec{A} + b)$$

$$\begin{cases} t \geq 0, & \text{正例} \\ t < 0, & \text{负例} \end{cases}$$

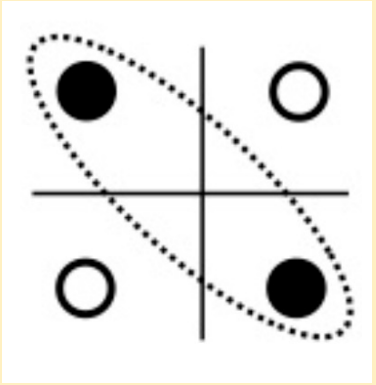


感知机(Perceptron)

感知机分类实质：在输入空间中找到一个能分离正负例的超平面。



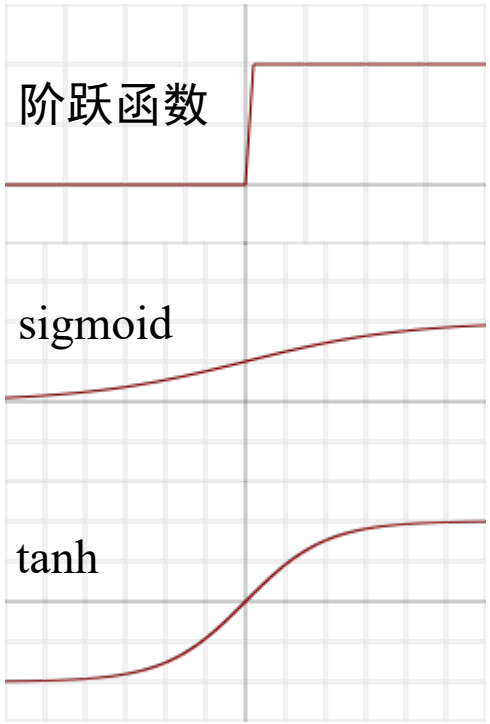
前提：数据线性可分



线性不可分的XOR问题

激活函数

现代神经网络中使用各种激活函数。



$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

$$f(x) \in \{0,1\}$$

$$f(x) = \frac{1}{1+e^{-x}}$$

$$f(x) \in (0,1)$$

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$f(x) \in (-1,1)$$

sigmoid 函数的一个性质

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

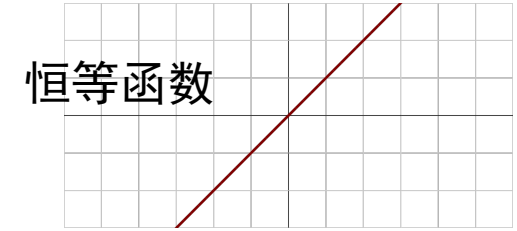
Softmax

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

$$f(x) \in (0,1)$$

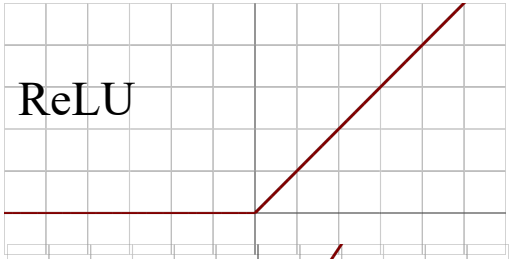
激活函数

恒等函数



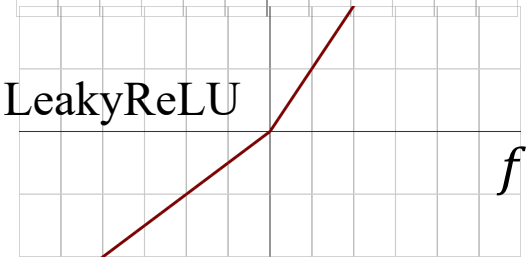
$$f(x) = x$$

ReLU



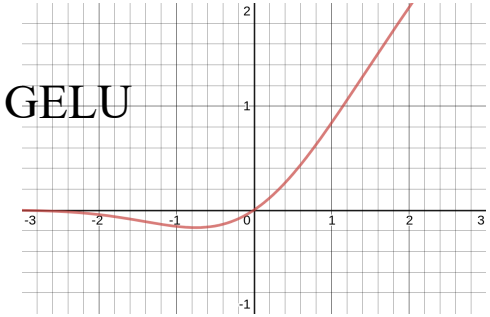
$$f(x) = \max(0, x) = \begin{cases} 0, & x \le 0 \\ x, & x > 0 \end{cases}$$

LeakyReLU

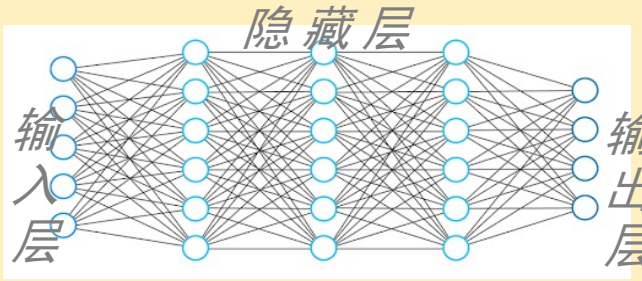


$$f(x) = \begin{cases} 0.01x, & x \le 0 \\ x, & x > 0 \end{cases}$$

GELU



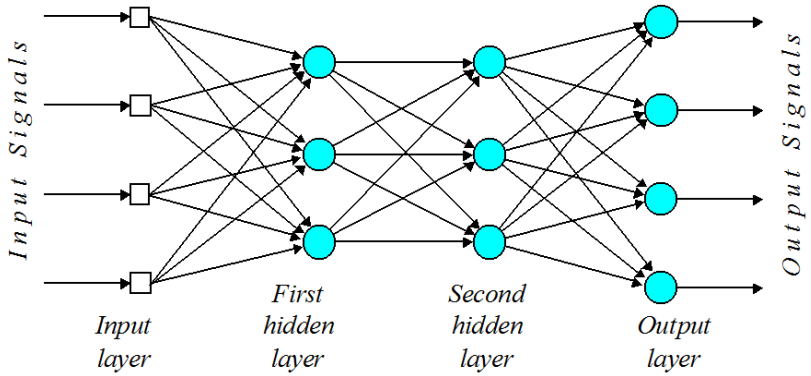
$$f(x) = x\Phi(x) = \frac{1}{2}x\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$



- 在分类任务中，输出层的激活函数一般使用sigmoid（二分类）或softmax（多分类）
- 在回归任务中，输出层的激活函数往往使用恒等函数

激活函数的选择对神经网络的训练有着较大影响

神经网络训练—损失函数



记

- 数据条数：N
- 神经网络输入：$\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$
- 神经网络输出：$\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots, \boldsymbol{f}_N$
- 真实标签/回归值：$\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_N$

对神经网络的参数进行优化，首先需要设计优化的目标函数，即损失函数

回归任务常用：均方误差函数

$$\boldsymbol{loss} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{f}_i - \boldsymbol{y}_i)^2$$

为什么使用平方误差而不用绝对误差？

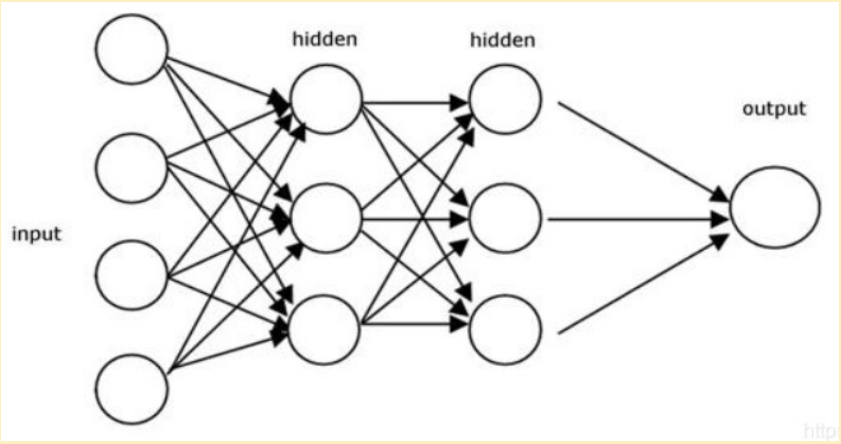
- 平方误差平滑可微
- 将大误差进一步放大

神经网络训练—损失函数

二分类任务常用：二分类交叉熵

神经网络输出:$f_1, f_2, \dots, f_N \in (0,1)$
真实标签/回归值:$y_1, y_2, \dots, y_N \in \{0,1\}$

$$\boldsymbol{loss} = -\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{y}\ln\boldsymbol{f} + (1-\boldsymbol{y})\ln(1-\boldsymbol{f})$$



多分类任务常用：分类交叉熵

使用one-hot编码
记类别数为M
神经网络输出： $\boldsymbol{f}_1, \boldsymbol{f}_2, \dots, \boldsymbol{f}_N \in \{(a_1, a_2, \dots, a_M) | a_1, a_2, \dots, a_M \in (0,1) \; and \; \sum_{i=1}^{M} a_i = 1\}$
真实标签/回归值： $\boldsymbol{y}_1, \boldsymbol{y}_2, \dots, \boldsymbol{y}_N \in \{(a_1, a_2, \dots, a_M) | a_1, a_2, \dots, a_M \in \{0,1\} \; and \; \sum_{i=1}^{M} a_i = 1\}$

$$\boldsymbol{loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{M} y_{ic}\ln(f_{ic})$$

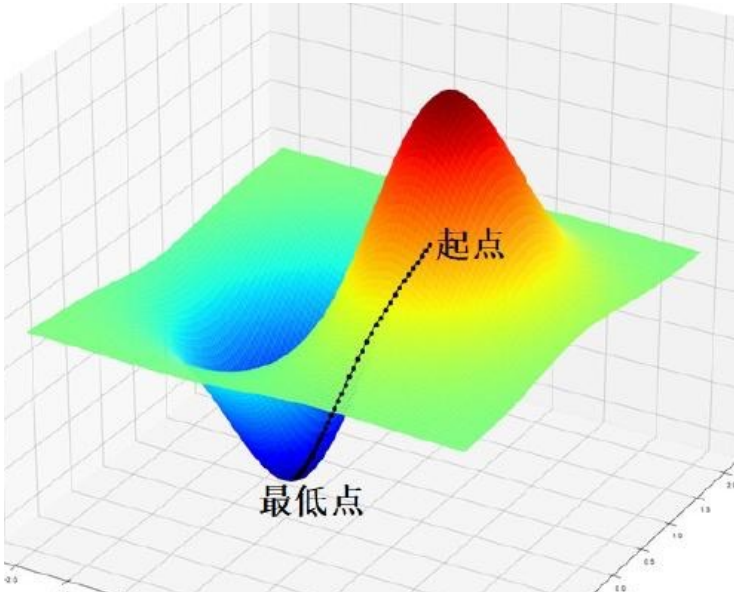
神经网络训练—优化器

确定损失函数后，在训练数据上对损失函数进行优化。

最简单的方法是梯度下降法。

目标：在给定的数据集上找到这样的参数$\boldsymbol{\theta}$，使得$\boldsymbol{loss}$值最小，即

$$\arg\min_{\boldsymbol{\theta}} \boldsymbol{loss}\,(\boldsymbol{\theta})$$

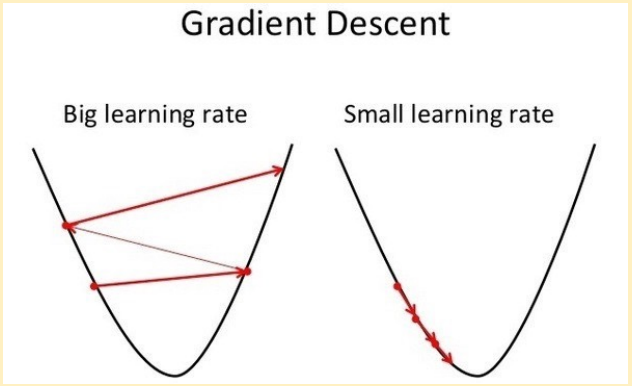


神经网络训练—优化器

梯度下降法步骤

1. 令$t = 0$初始化$\boldsymbol{\theta}^{(t)} \in \boldsymbol{R}^{k}$,
2. 计算$\boldsymbol{loss}(\boldsymbol{\theta}^{(t)})$
3. 计算梯度函数$\nabla \boldsymbol{loss}(\boldsymbol{\theta}^{(t)})$，若满足条件，结束；令$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \nabla \boldsymbol{loss}(\boldsymbol{\theta}^{(t)})$，其中$\lambda$为梯度下降的学习率
4. 令$t = t + 1$，回到2

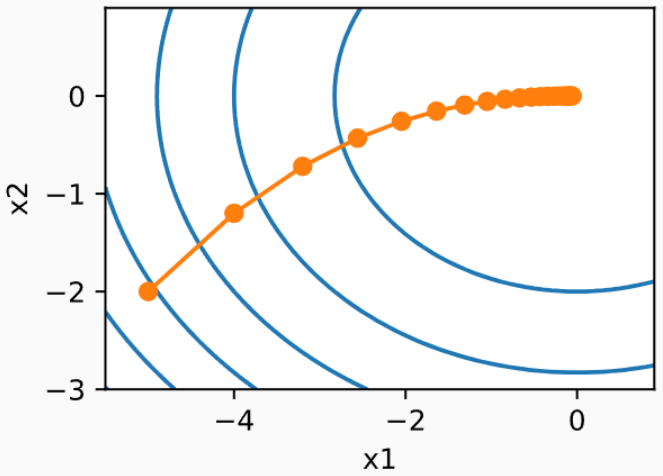
学习率对梯度下降法收敛速度的影响



神经网络训练—优化器

随机梯度下降法

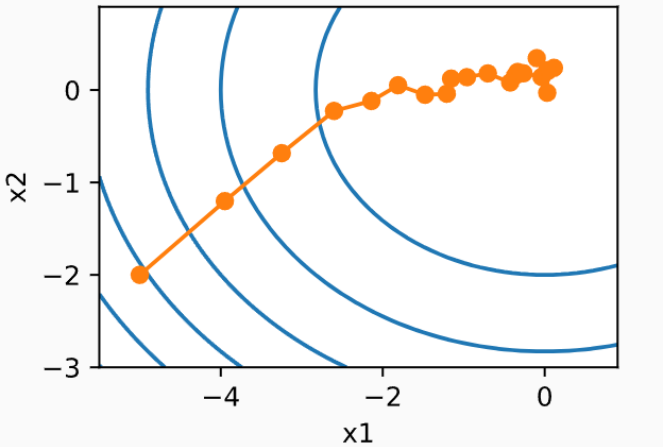
- 每次使用数据中的一个样本的梯度作为更新梯度
- 将局部数据的梯度视作整体数据的期望
- 相较于普通梯度下降法每次迭代需要计算全部数据，大大降低计算开销



二维梯度下降法迭代轨迹

批量梯度下降法

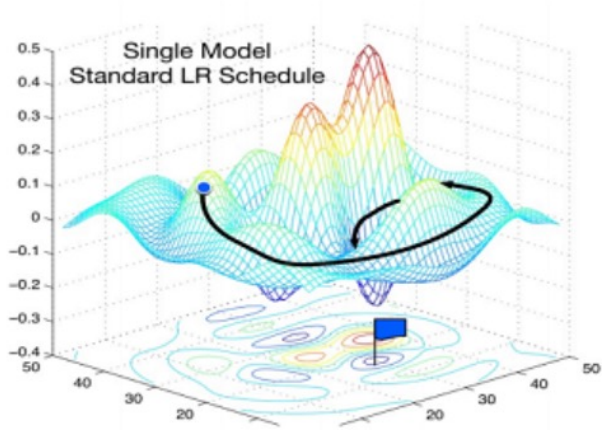
- 每次使用数据中的一个批（batch）的梯度作为更新梯度



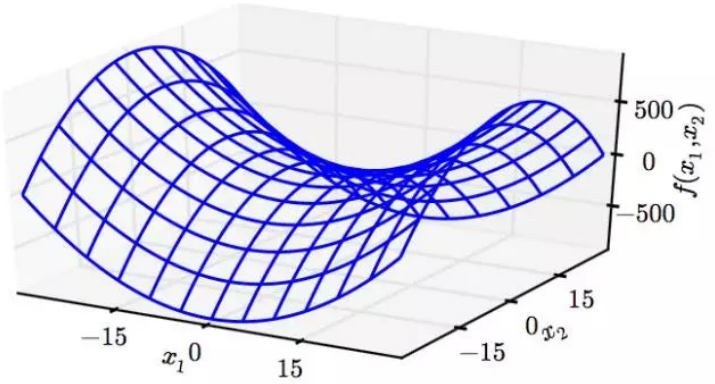
二维随机梯度下降法迭代轨迹

神经网络训练—优化器

引入动量的梯度计算方法



局部最优



鞍点

动量法

引入了速度$\boldsymbol{v}$和衰减率α，为负梯度的指数衰减平均

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \boldsymbol{v}$$

其中

$$\boldsymbol{v}^{(0)} = 0,$$
$$\boldsymbol{v}^{(t+1)} = \alpha \boldsymbol{v}^{(t)} - \lambda \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

- 降低了随机梯度方差的影响
- 使梯度优化能更好走出局部最优和鞍点

神经网络训练—优化器

学习率自适应的优化算法

AdaGrad

对原本的梯度下降算法进行了改进

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\lambda}{\delta + \sqrt{r^{(t)}}} \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

其中

$$\boldsymbol{r}^{(0)} = 0,$$
$$\boldsymbol{r}^{(t+1)} = \boldsymbol{r}^{(t)} + \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right) \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

- 针对每个参数元素有不同的学习率（适合稀疏权重）
  - 历史偏导越大的元素，学习率衰减越快
  - 在平缓的方向，学习率衰减得更慢
- 在凸函数的设定下效果好，在非凸环境下学习率衰减过快

神经网络训练—优化器

学习率自适应的优化算法

RMSProp

对AdaGrad进行了改进，增加衰减速率超参ρ

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\lambda}{\delta + \sqrt{r^{(t)}}} \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

其中

$$\boldsymbol{r}^{(0)} = 0,$$

$$\boldsymbol{r}^{(t+1)} = \rho \boldsymbol{r}^{(t)} + (1-\rho)\ \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right) \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

- 用指数衰减的方法丢弃较远的梯度历史

神经网络训练—优化器

学习率自适应的优化算法

Adam

在梯度一阶矩中引入了动量，同时增加偏置修正

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\lambda \widehat{\boldsymbol{s}^{(t)}}}{\delta + \sqrt{\widehat{r^{(t)}}}} \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$$

其中

$$\boldsymbol{s}^{(0)} = 0, \boldsymbol{r}^{(0)} = 0,$$

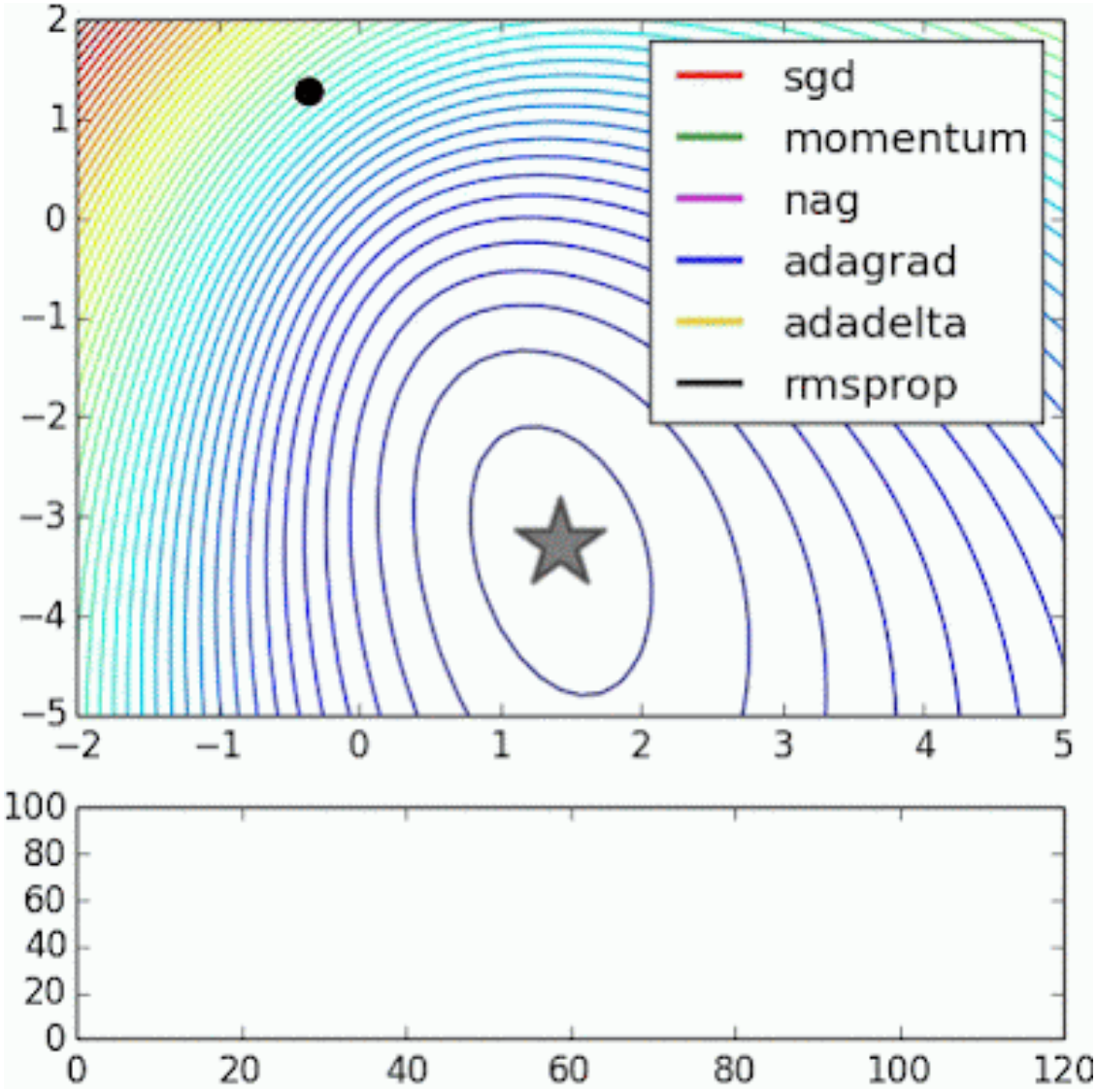
有偏一阶矩估计 $\boldsymbol{s}^{(t+1)} = \rho_1 \boldsymbol{r}^{(t)} + (1-\rho_2)\ \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$

有偏二阶矩估计 $\boldsymbol{r}^{(t+1)} = \rho_2 \boldsymbol{r}^{(t)} + (1-\rho_2)\ \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right) \odot \nabla \boldsymbol{loss}\left(\boldsymbol{\theta}^{(t)}\right)$

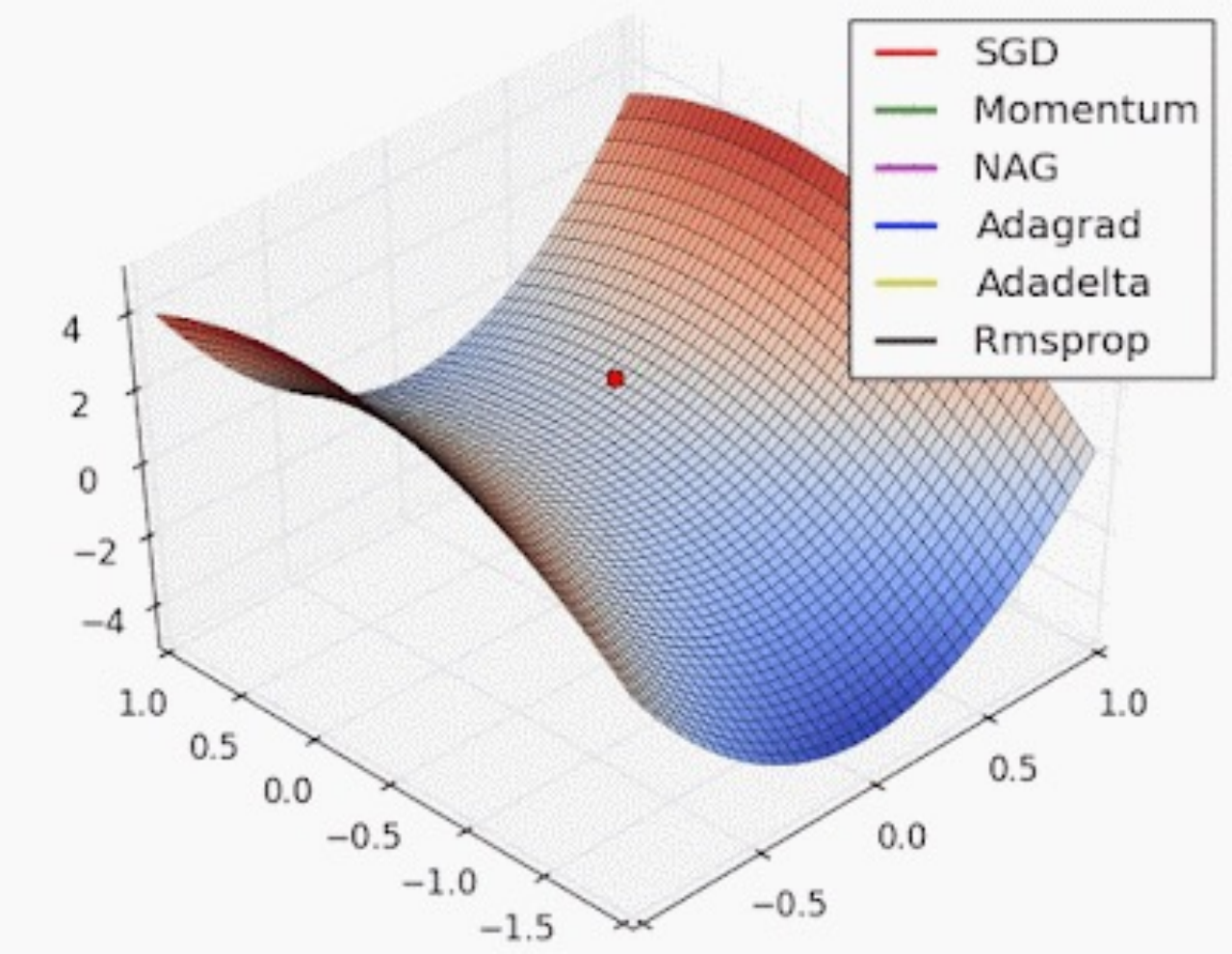
修正偏差 $\widehat{\boldsymbol{s}^{(t)}} = \frac{\boldsymbol{s}}{1-\rho_1^t},\ \widehat{\boldsymbol{r}^{(t)}} = \frac{\boldsymbol{r}}{1-\rho_1^t}$

- 结合动量法，初期快速收敛
- 对稀疏梯度适应性好

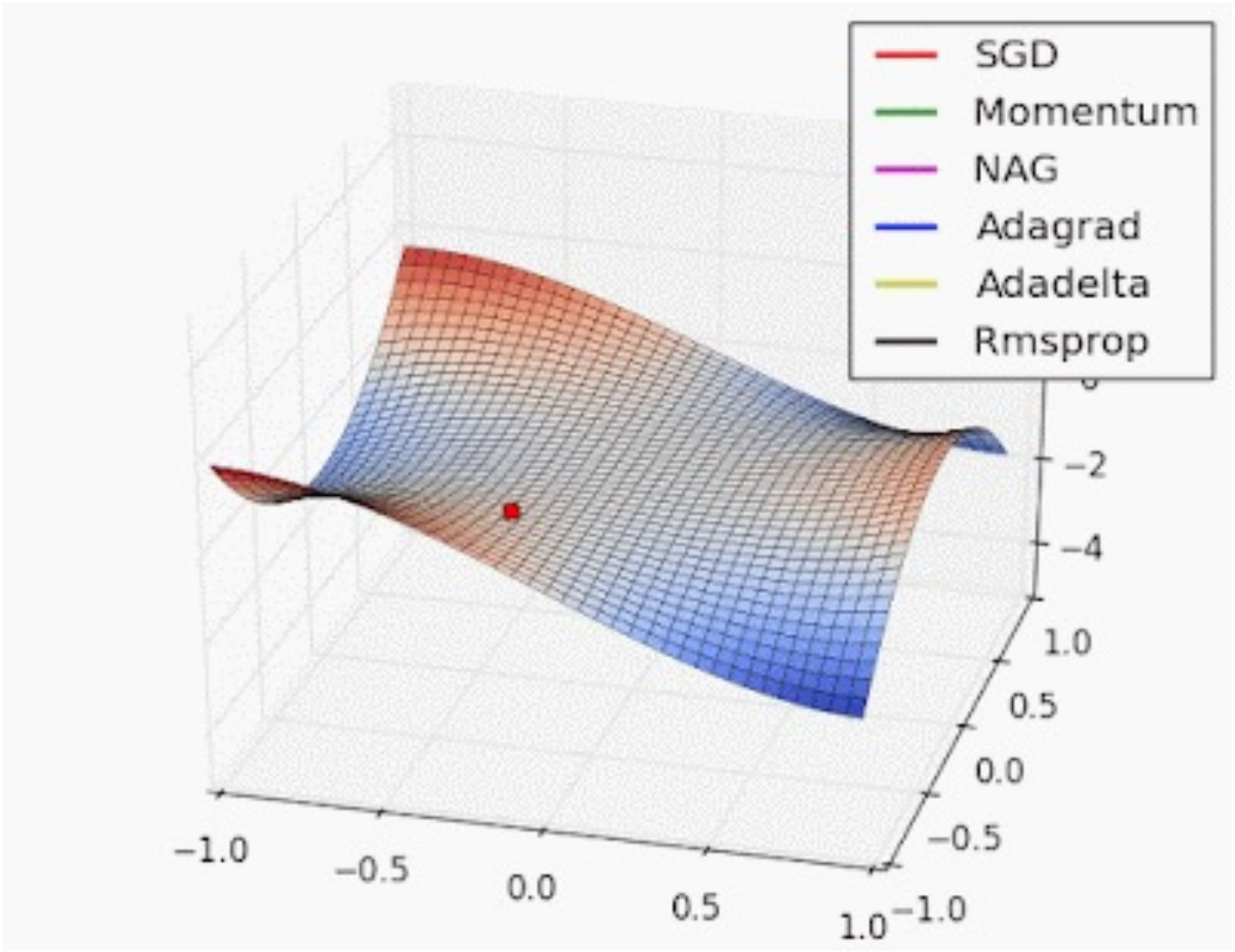
神经网络训练—优化器


sgd
momentum
nag
adagrad
adadelta
rmsprop
2
1
0
−1
−2
−3
−4
−5
−2
−1
0
1
2
3
4
5
100
80
60
40
20
0
0
20
40
60
80
100
120

神经网络训练—优化器


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

神经网络训练—优化器

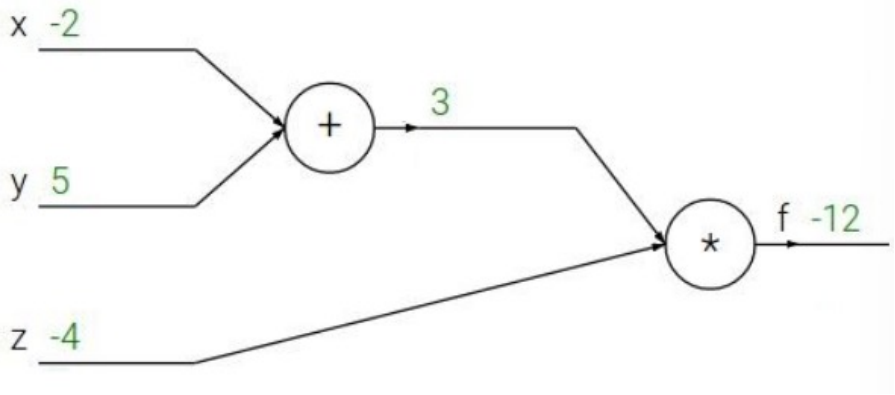

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

神经网络训练—反向传播

确定损失函数后和优化方法后，需要对梯度进行计算。

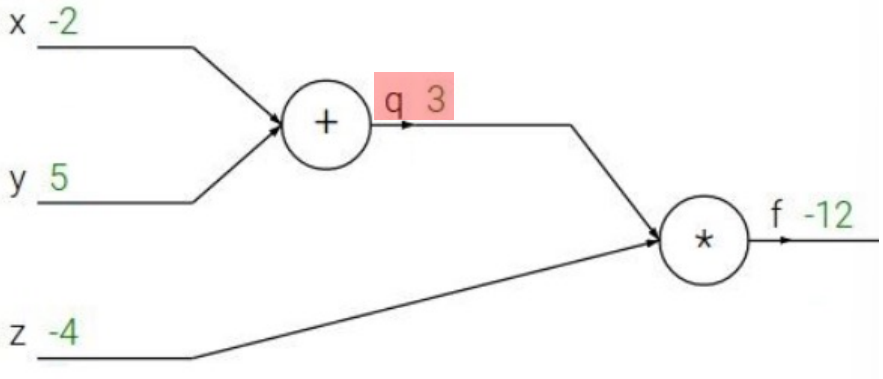
反向传播（Backpropagation）是对多层人工神经网络进行梯度下降的算法，也就是用链式法则以网络每层的权重为变量计算损失函数的梯度，以更新权重来最小化损失函数。

神经网络训练—反向传播



$$f(x, y, z) = (x + y)z$$

神经网络训练—反向传播

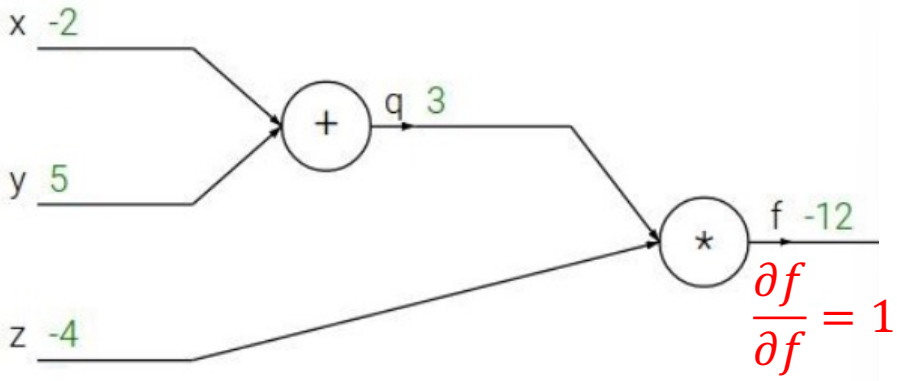


$$f(x,y,z) = (x+y)z$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

神经网络训练—反向传播

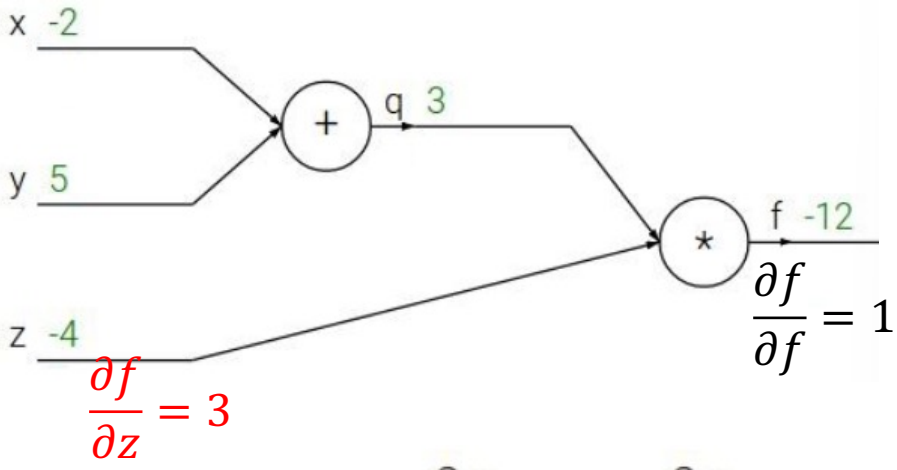


$$f(x, y, z) = (x + y)z$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

神经网络训练—反向传播

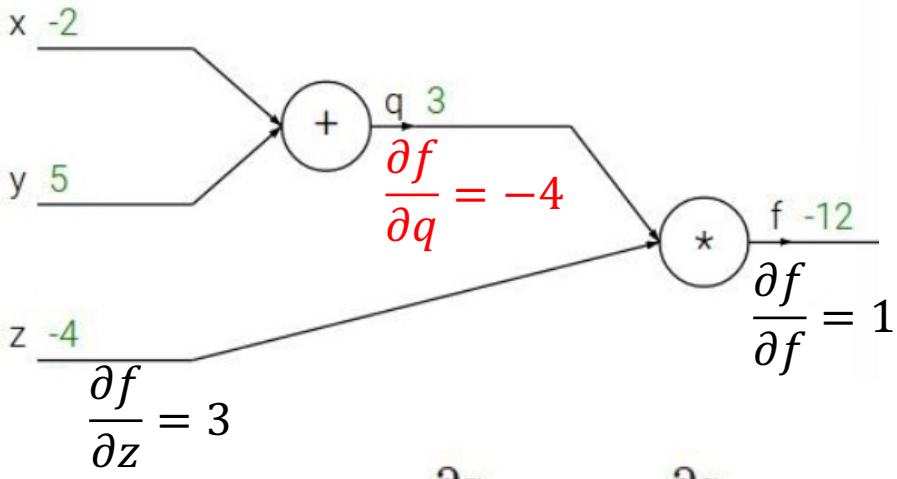


$$f(x, y, z) = (x + y)z$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

神经网络训练—反向传播

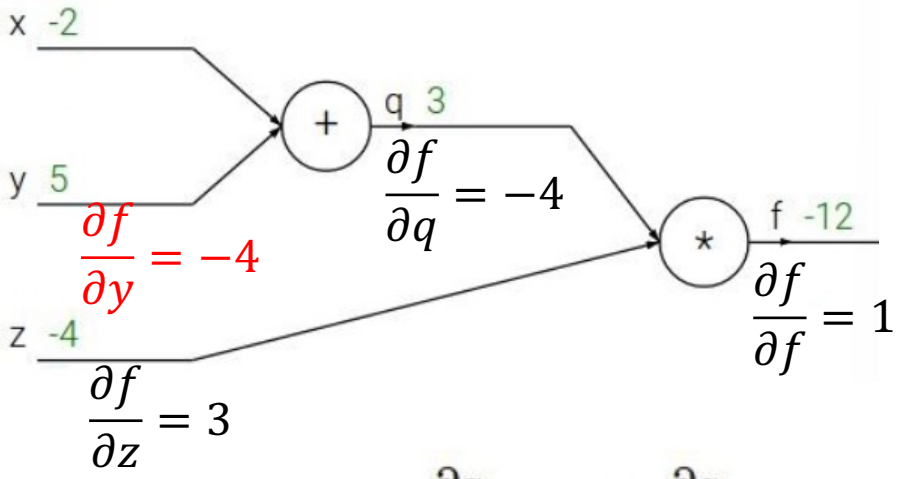


$$f(x, y, z) = (x + y)z$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

神经网络训练—反向传播



$$f(x, y, z) = (x + y)z$$

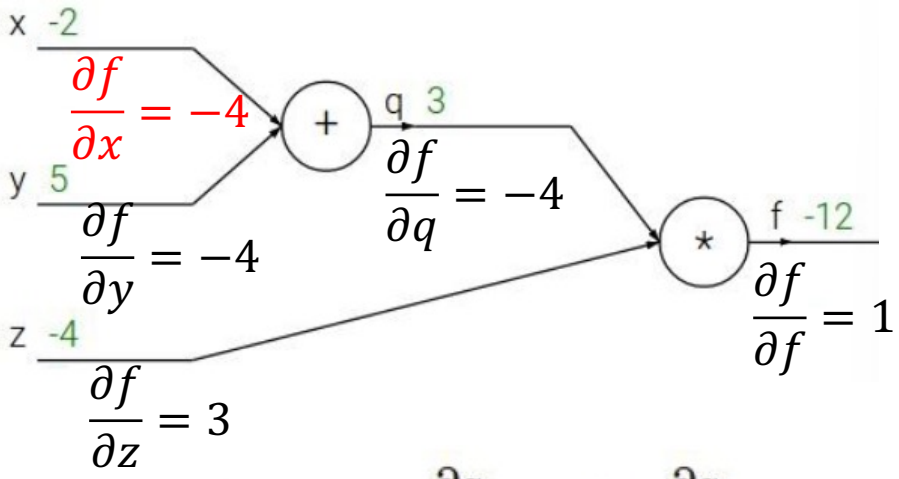
$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

链式求导法则

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

神经网络训练—反向传播



$$f(x, y, z) = (x + y)z$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

链式求导法则

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

神经网络训练—反向传播

反向传播(Backpropagation)由两个环节组成：
前向计算(Forward Computation)和反向计算(Backward Computation)

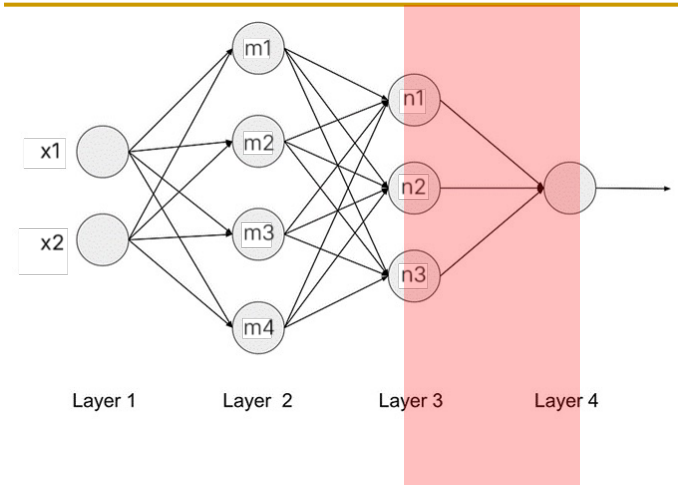
前向计算

1. 使用一定的算法生成$y = f(x)$对应的有向无环的计算图；
2. 以拓扑排序顺序访问各个结点，对每个结点计算当前结点的值并存储。

反向计算

1. 偏导初始化：令$\frac{\partial y}{\partial y} = 1$,其他偏导值为0；
2. 以逆拓扑排序顺序访问各个结点，使用链式求导法则对每个结点计算当前结点的偏导值并存储。

神经网络训练—反向传播



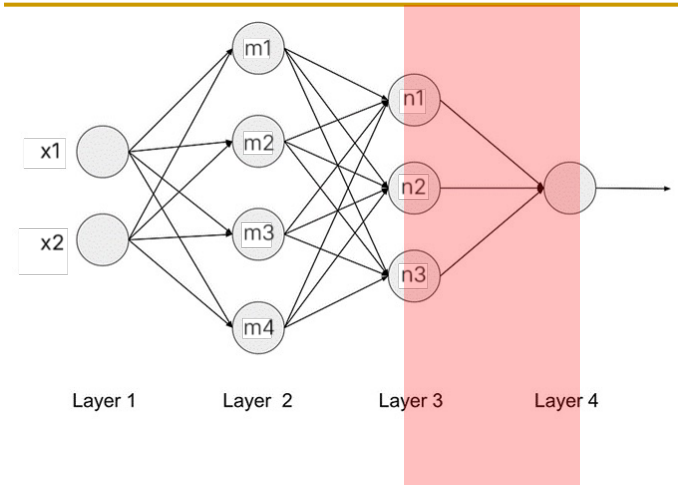
对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2)$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3)$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial f} = -\frac{y}{f} + \frac{1-y}{1-f}$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$

$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$

$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$

$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

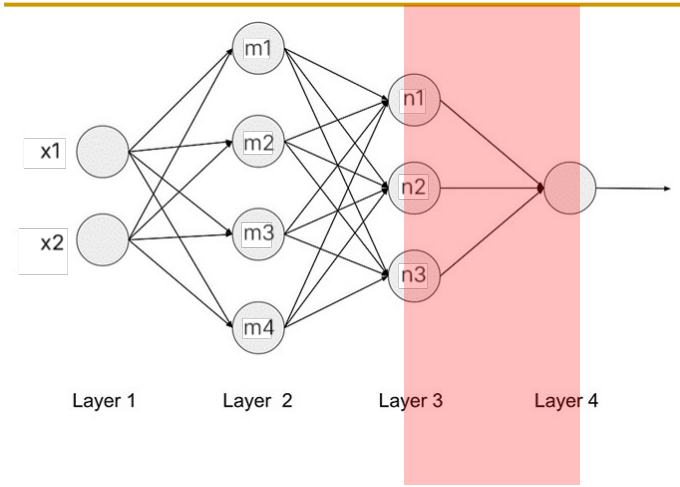
Sigmoid函数性质：

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$\frac{\partial loss}{\partial f} = -\frac{y}{f} + \frac{1-y}{1-f}$$

$$\frac{\partial loss}{\partial \tilde{f}} = \frac{\partial loss}{\partial f} \times \frac{\partial f}{\partial \tilde{f}} = \left(-\frac{y}{f} + \frac{1-y}{1-f}\right) \times f(1-f) = f - y$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial f} = -\frac{y}{f} + \frac{1-y}{1-f}$$

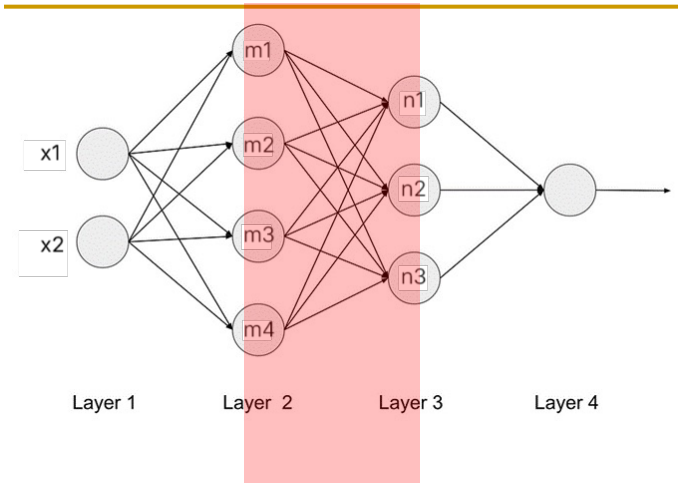
$$\frac{\partial loss}{\partial \tilde{f}} = \frac{\partial loss}{\partial f} \times \frac{\partial f}{\partial \tilde{f}} = \left(-\frac{y}{f} + \frac{1-y}{1-f}\right) \times f(1-f) = f - y$$

$$\frac{\partial loss}{\partial \boldsymbol{W}_3} = \frac{\partial loss}{\partial \tilde{f}} \times \frac{\partial \tilde{f}}{\partial \boldsymbol{W}_3} = \frac{\partial loss}{\partial \tilde{f}} \boldsymbol{n}^T$$

$$\frac{\partial loss}{\partial \boldsymbol{b}_3} = \frac{\partial loss}{\partial \tilde{f}} \times \frac{\partial \tilde{f}}{\partial \boldsymbol{b}_3} = \frac{\partial loss}{\partial \tilde{f}}$$

$$\frac{\partial loss}{\partial \boldsymbol{n}} = \frac{\partial loss}{\partial \tilde{f}} \times \frac{\partial \tilde{f}}{\partial \boldsymbol{n}} = \frac{\partial loss}{\partial \tilde{f}} \boldsymbol{W}_3^T$$

神经网络训练—反向传播



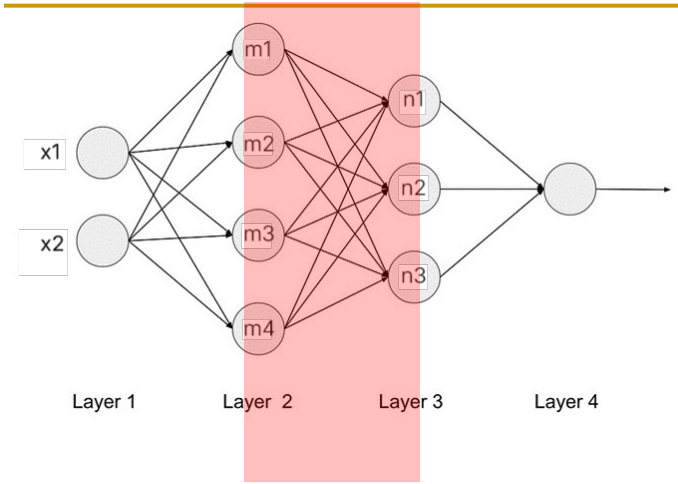
对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial \boldsymbol{n}} = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

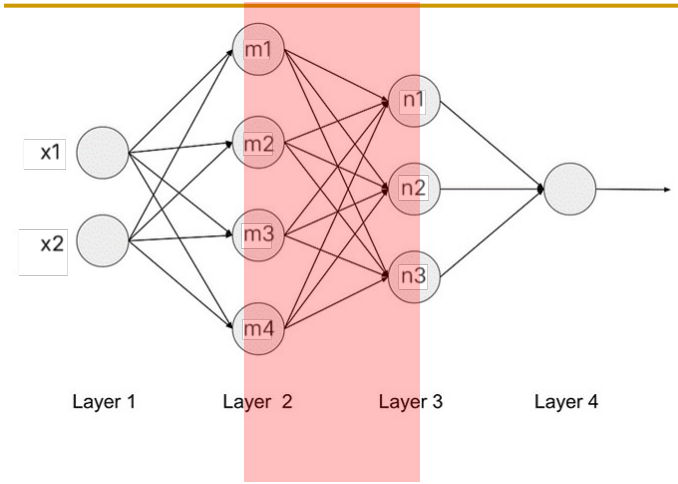
$$\frac{\partial loss}{\partial \boldsymbol{n}} = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T$$

$$\frac{\partial loss}{\partial \tilde{\boldsymbol{n}}} = \frac{\partial loss}{\partial \boldsymbol{n}} \times \frac{\partial \boldsymbol{n}}{\partial \tilde{\boldsymbol{n}}} = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T \mathrm{diag}(n_1(1-n_1), n_2(1-n_2), \dots) = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T \odot \boldsymbol{n} \odot (1-\boldsymbol{n})$$

Sigmoid函数性质：

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial \boldsymbol{n}} = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T$$

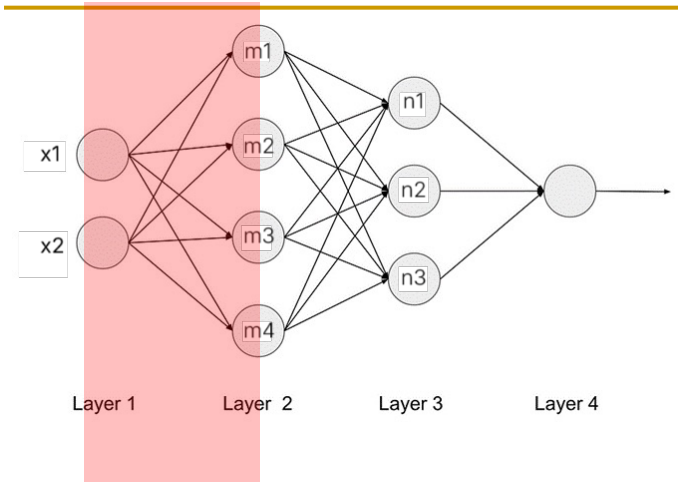
$$\frac{\partial loss}{\partial \tilde{\boldsymbol{n}}} = \frac{\partial loss}{\partial \boldsymbol{n}} \times \frac{\partial \boldsymbol{n}}{\partial \tilde{\boldsymbol{n}}} = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T \mathrm{diag}(n_1(1-n_1), n_2(1-n_2), \dots) = \frac{\partial loss}{\partial \tilde{f}}\boldsymbol{W}_3^T \odot \boldsymbol{n} \odot (1-\boldsymbol{n})$$

$$\frac{\partial loss}{\partial \boldsymbol{W}_2} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}} \times \frac{\partial \tilde{\boldsymbol{n}}}{\partial \boldsymbol{W}_2} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{m}^T$$

$$\frac{\partial loss}{\partial \boldsymbol{b}_2} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}} \times \frac{\partial \tilde{\boldsymbol{n}}}{\partial \boldsymbol{b}_2} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}$$

$$\frac{\partial loss}{\partial \boldsymbol{m}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}} \times \frac{\partial \tilde{\boldsymbol{n}}}{\partial \boldsymbol{m}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T$$

神经网络训练—反向传播



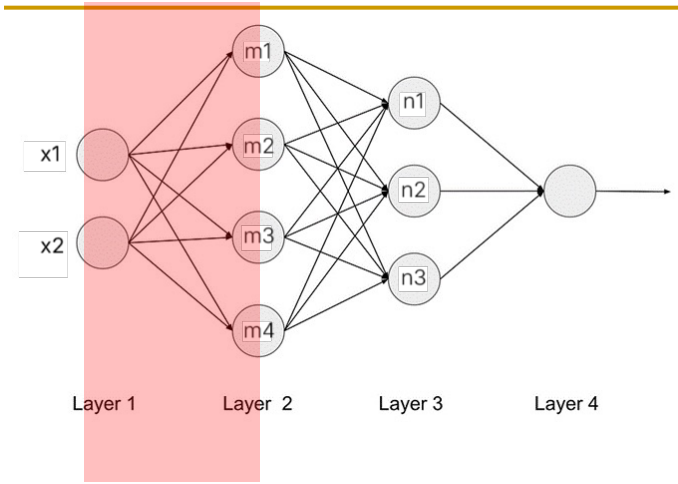
对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial \boldsymbol{m}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

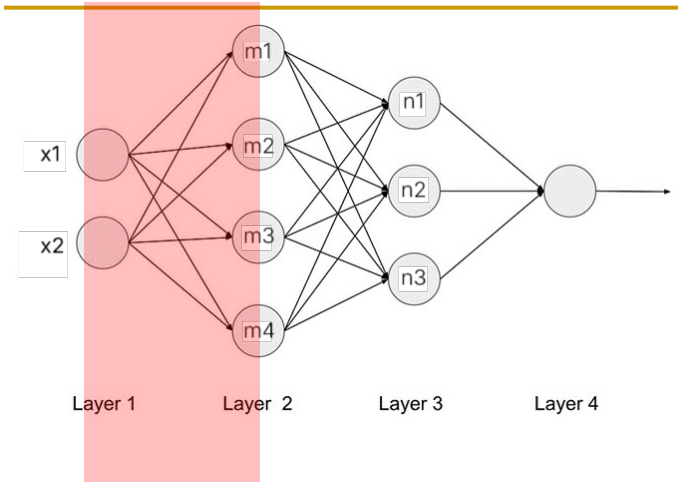
$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial \boldsymbol{m}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T$$

$$\frac{\partial loss}{\partial \tilde{\boldsymbol{m}}} = \frac{\partial loss}{\partial \boldsymbol{m}} \times \frac{\partial \boldsymbol{m}}{\partial \tilde{\boldsymbol{m}}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T \odot \boldsymbol{m} \odot (1-\boldsymbol{m})$$

神经网络训练—反向传播



对于一个样本$(\boldsymbol{X}, y)$,

$$\boldsymbol{m} = \sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) = \sigma(\tilde{\boldsymbol{m}})$$
$$\boldsymbol{n} = \sigma(\boldsymbol{W}_2\boldsymbol{m} + \boldsymbol{b}_2) = \sigma(\tilde{\boldsymbol{n}})$$
$$f = \sigma(\boldsymbol{W}_3\boldsymbol{n} + \boldsymbol{b}_3) = \sigma(\tilde{f})$$
$$loss = -y\ln f - (1-y)\ln(1-f)$$

通过前向计算，已知$\boldsymbol{x}, \boldsymbol{m}, \boldsymbol{n}, f, y, loss$

$$\frac{\partial loss}{\partial \boldsymbol{m}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T$$

$$\frac{\partial loss}{\partial \tilde{\boldsymbol{m}}} = \frac{\partial loss}{\partial \boldsymbol{m}} \times \frac{\partial \boldsymbol{m}}{\partial \tilde{\boldsymbol{m}}} = \frac{\partial loss}{\partial \tilde{\boldsymbol{n}}}\boldsymbol{W}_2^T \odot \boldsymbol{m} \odot (1-\boldsymbol{m})$$

$$\frac{\partial loss}{\partial \boldsymbol{W}_1} = \frac{\partial loss}{\partial \tilde{\boldsymbol{m}}} \times \frac{\partial \tilde{\boldsymbol{m}}}{\partial \boldsymbol{W}_1} = \frac{\partial loss}{\partial \tilde{\boldsymbol{m}}}\boldsymbol{x}^T$$

$$\frac{\partial loss}{\partial \boldsymbol{b}_1} = \frac{\partial loss}{\partial \tilde{\boldsymbol{m}}} \times \frac{\partial \tilde{\boldsymbol{m}}}{\partial \boldsymbol{b}_1} = \frac{\partial loss}{\partial \tilde{\boldsymbol{m}}}$$

神经网络训练—评估

完成模型训练后，需要对模型的效果进行评估。

在神经网络的训练过程中，通常将完整的数据集分为训练集和测试集。

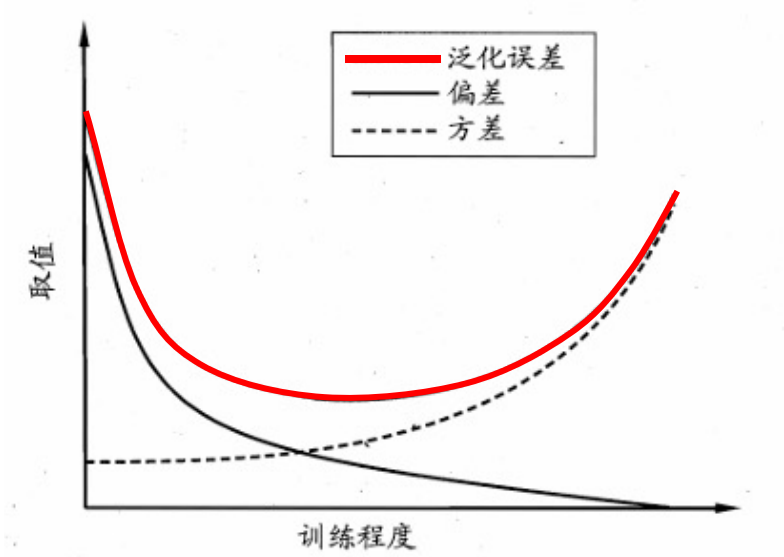
- 训练误差：在训练集上的误差
- 泛化误差：在真实分布上的误差
- 测试误差：在测试集上的误差，用于近似泛化误差

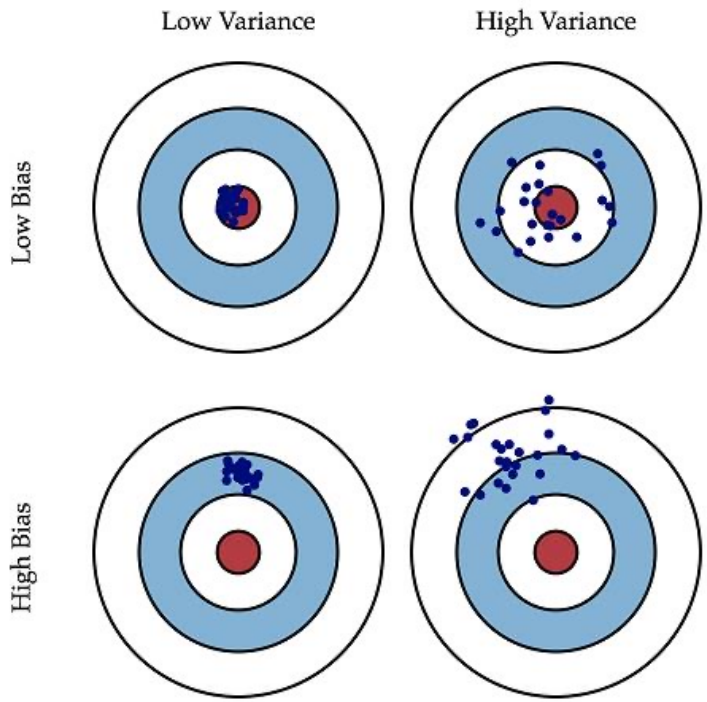
神经网络训练—评估

泛化误差的偏差-方差分解

$$E(f;D) = bias^2(\boldsymbol{x}) + var(\boldsymbol{x}) + \varepsilon^2$$

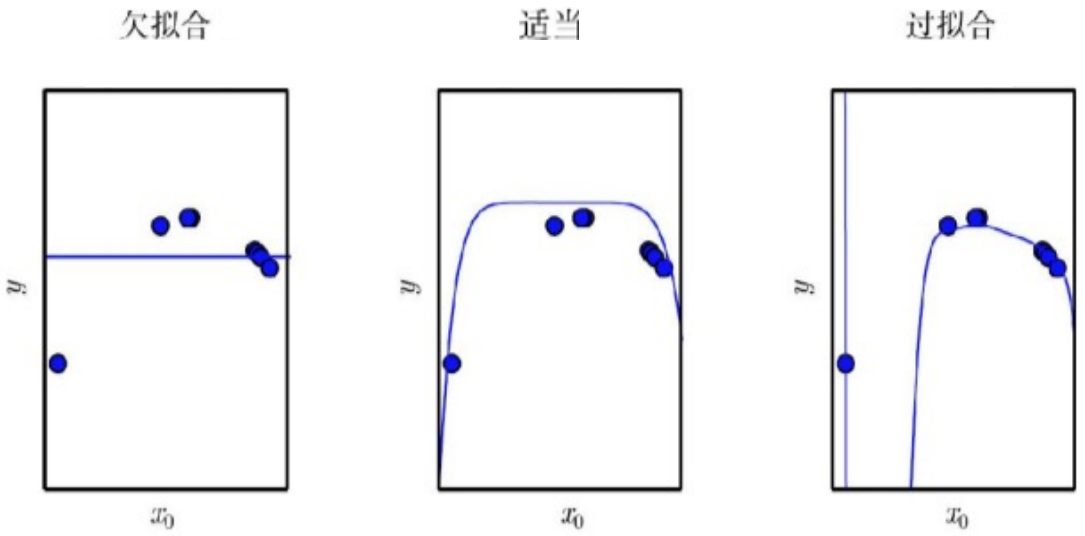
偏差：算法的期望预测与真实结果的偏离
方差：训练集数据扰动的影响
噪音：期望泛化误差的下界，即问题本身的难度





神经网络训练—评估

神经网络训练中常出现的问题是过拟合。



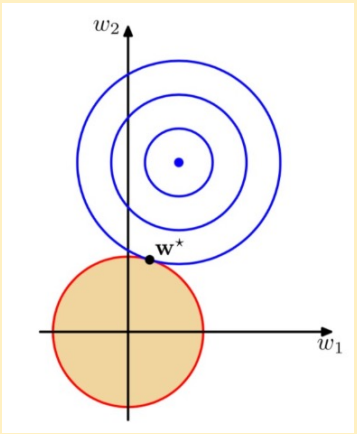
- 在训练程度充足后，学习器的拟合能力已非常强，训练数据发生的轻微扰动都会导致学习器发生显著变化
- 若训练数据自身的、非全局的特性被学习器学到了，则将发生过拟合。

神经网络训练—评估

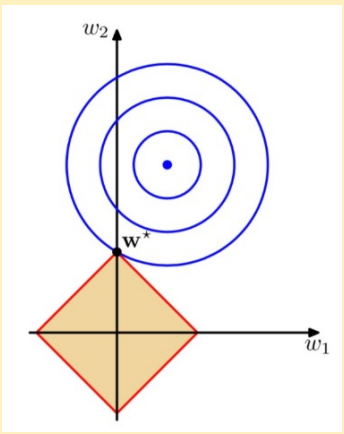
解决过拟合的方法：

- 早停法：在数据集中额外划分出一部分作为验证集，若训练集误差降低而验证集误差升高，则停止训练

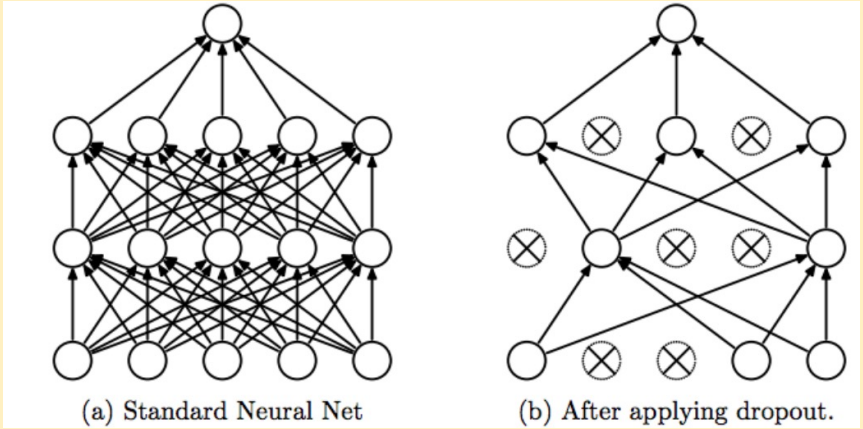
- 正则化：通过引入噪声或限制模型的复杂度，降低模型对输入或者参数的敏感性，避免过拟合，提高模型的泛化能力。



L2正则



L1正则



Dropout

Kolmogorov-Arnold表示定理

- Robert Hecht-Nielsen (HNC创始人)，1987，证明三层以上神经网络可以无限逼近任意连续函数。
- 理论依据是苏联数学家Kolmogorov 1957年为解决希尔伯特第十三问题证明的Kolmogorov-Arnold表示定理，也被称为Kolmogorov映射定理。

The works of Vladimir Arnold and Andrey Kolmogorov established that if *f* is a multivariate continuous function, then *f* can be written as a finite composition of continuous functions of a single variable and the binary operation of addition.[4] More specifically,

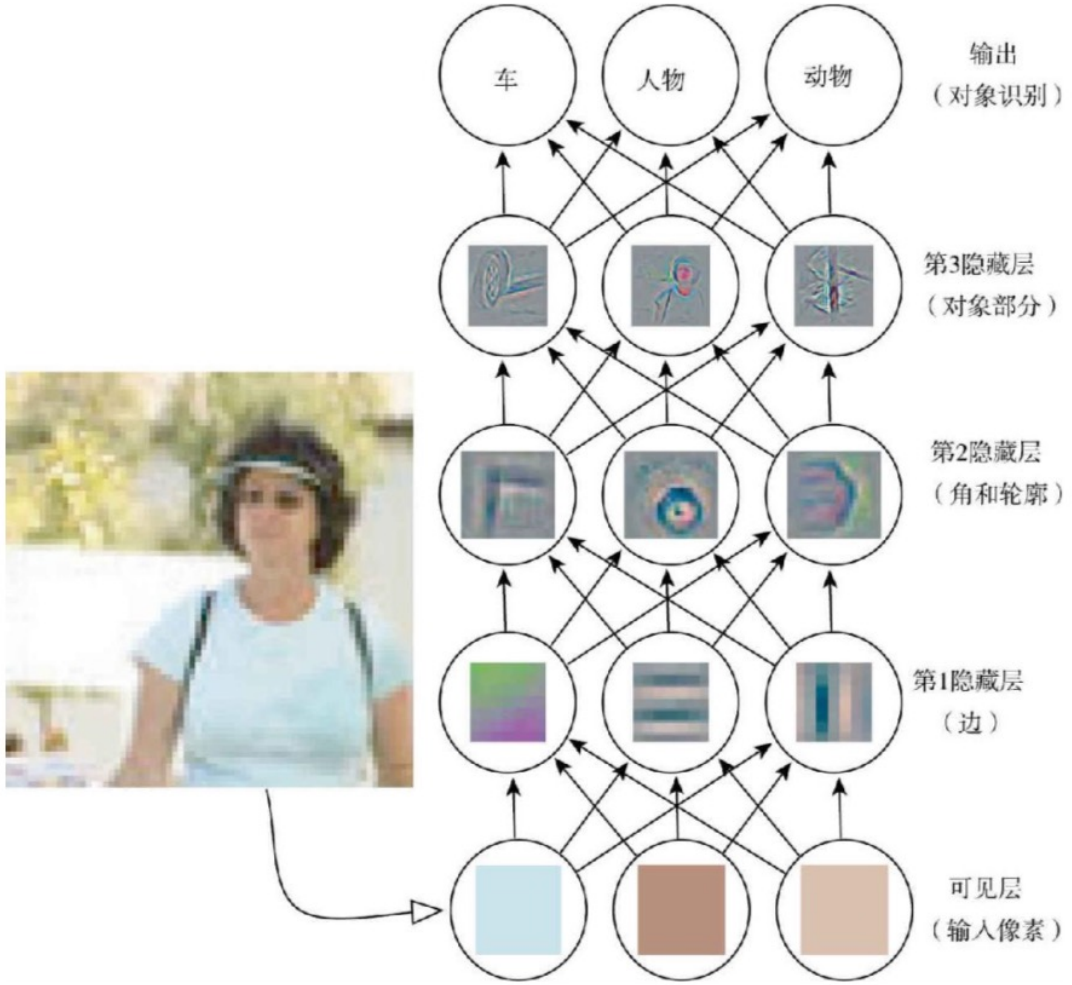
$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right).$$

where $\phi_{q,p} : [0,1] \to \mathbb{R}$ and $\Phi_q : \mathbb{R} \to \mathbb{R}$.

课程大纲

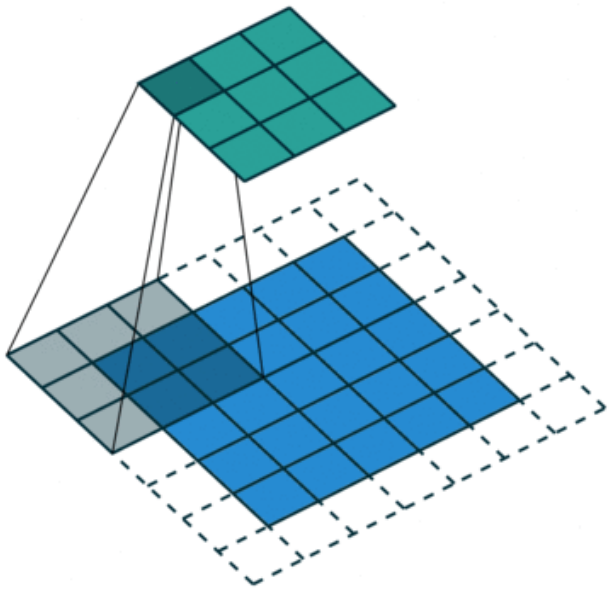
- 神经网络及训练
- **深度学习网络架构**
- 大模型和表示收敛
- PyTorch使用

深度学习——卷积神经网络(CNN)



深度学习的“深”体现在通过模块的堆叠，从简单的表示中构建复杂的概念，解决了表示学习中的核心问题。

深度学习——卷积神经网络(CNN)



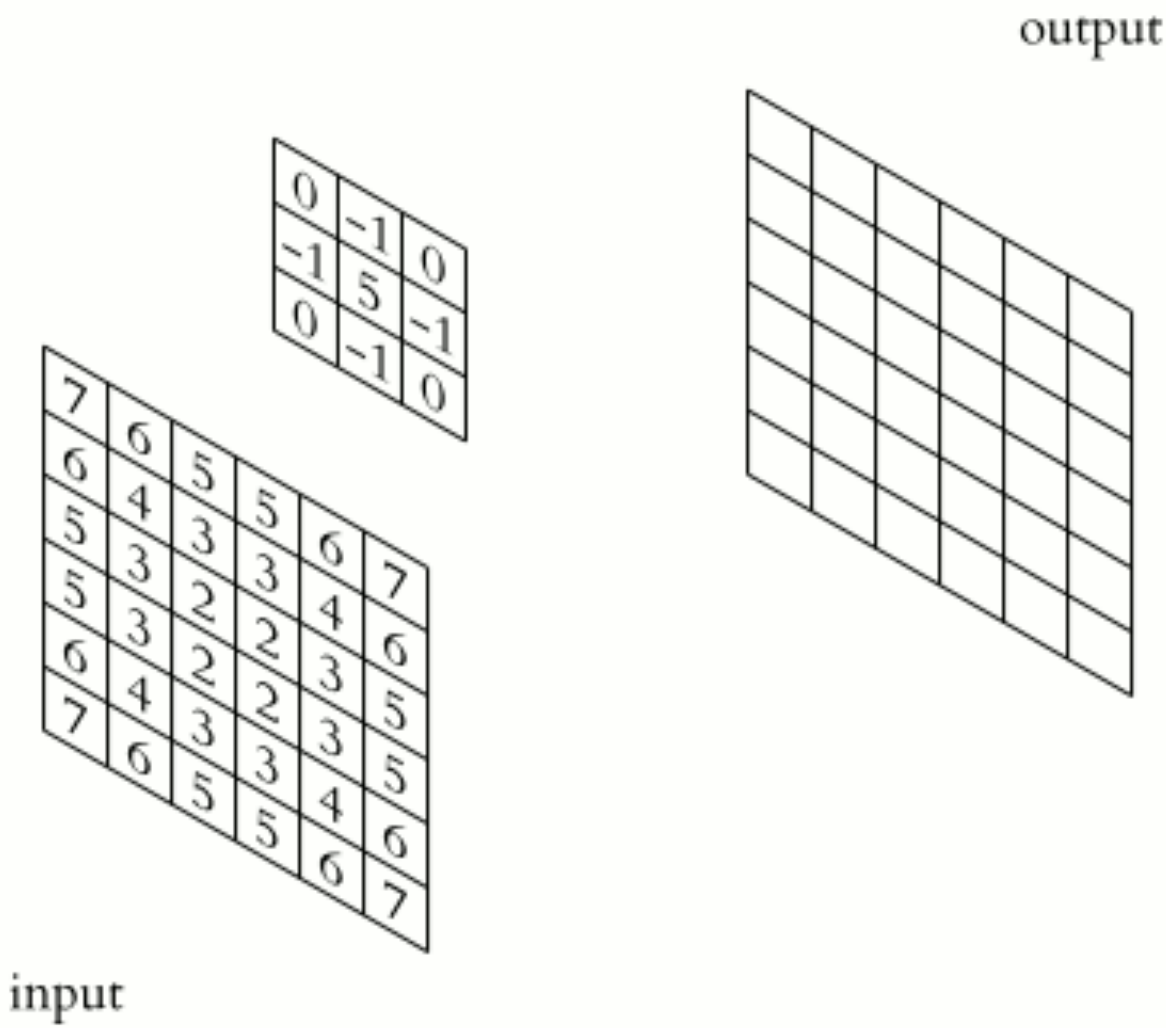
卷积操作包括以下几个要素：

- 卷积核大小（kernel size): 3*3
- 步长（step size）： 2
- 输出通道数/特征映射数/卷积核数

卷积（互相关）公式

$$\boldsymbol{H}_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} \boldsymbol{V}_{a,b,c,d} \boldsymbol{X}_{i+a,j+b,c}$$

深度学习——卷积神经网络(CNN)

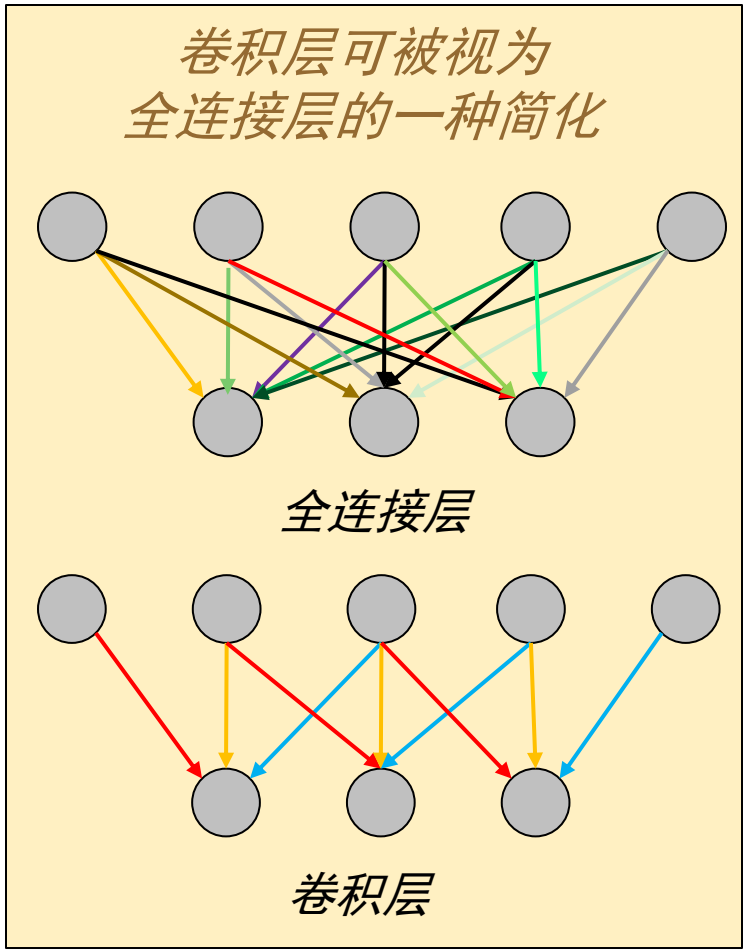


https://commons.wikimedia.org/wiki/File:2D_Convolution_Animation.gif

深度学习——卷积神经网络(CNN)

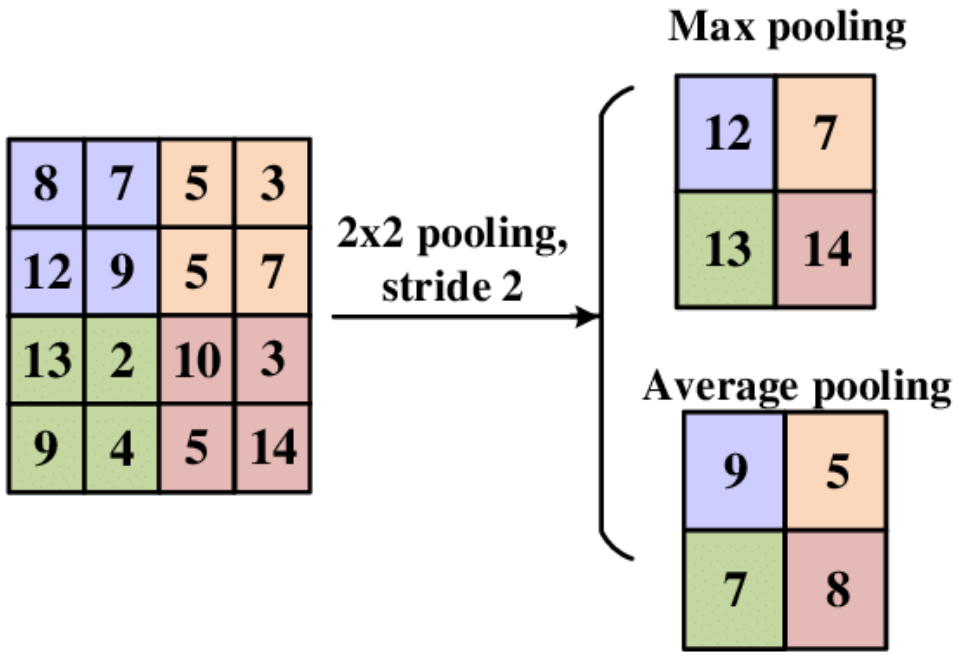
卷积操作的特点：

- 局部感知：卷积核大小一般小于输入大小，提取局部特征
- 参数共享：减少参数量

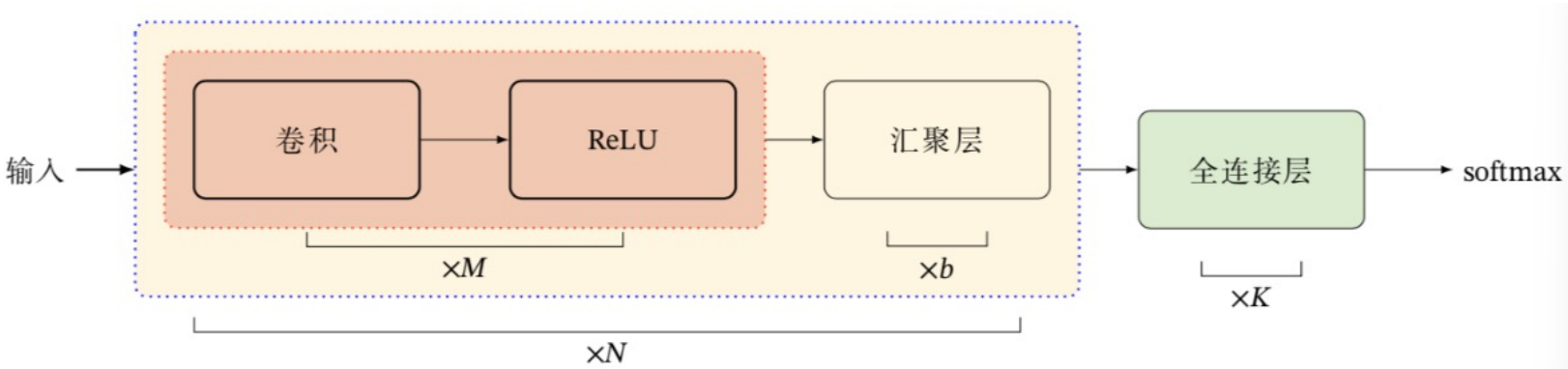


池化操作：特征选择，缓解过拟合

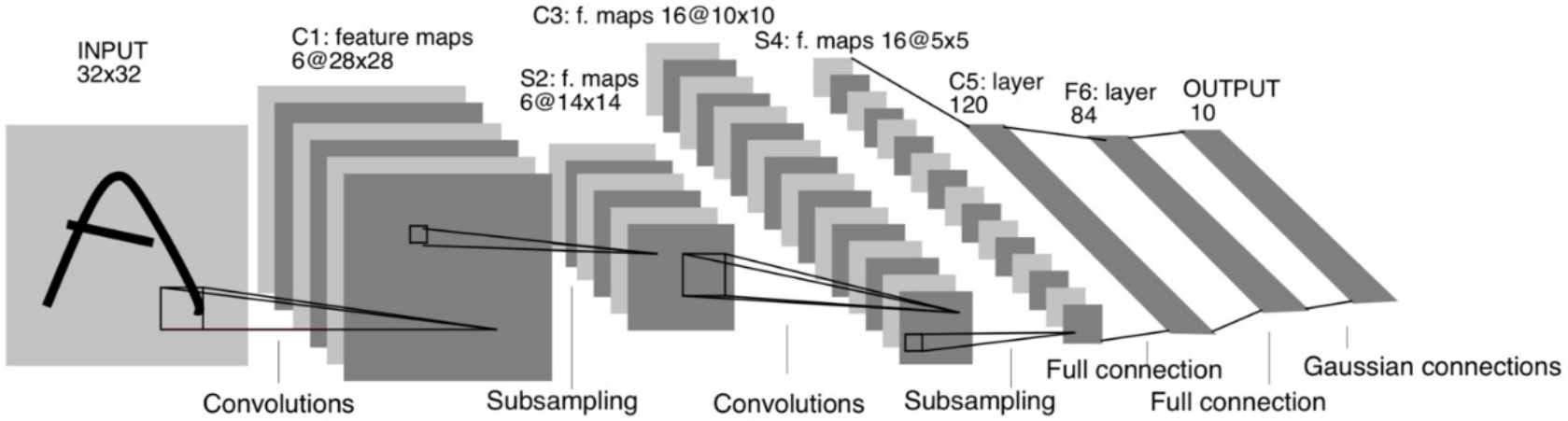
- 最大池化(Max Pooling)：选择区域内最大值
- 平均池化(Average Pooling)：选择区域内平均值



深度学习——卷积神经网络(CNN)



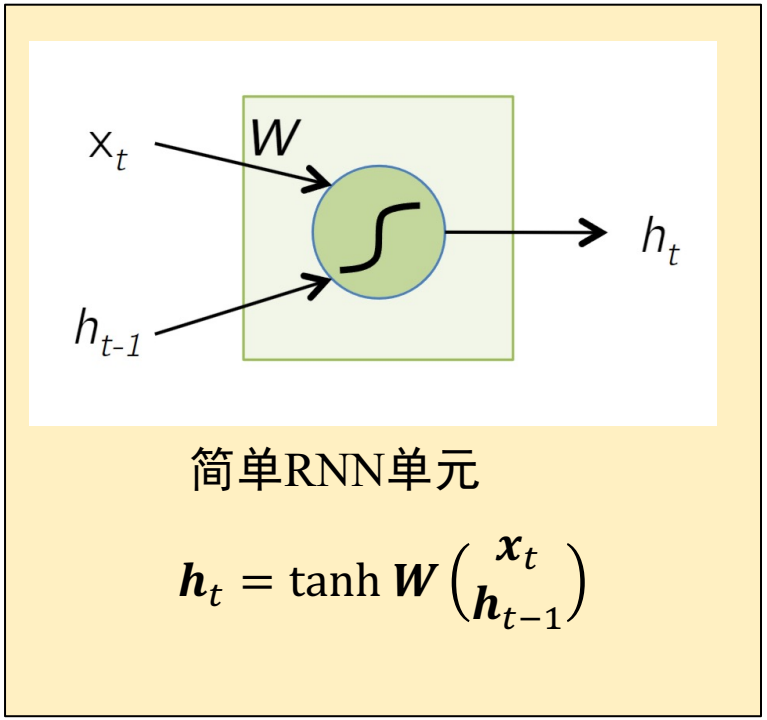
典型的CNN架构



LeNet-5

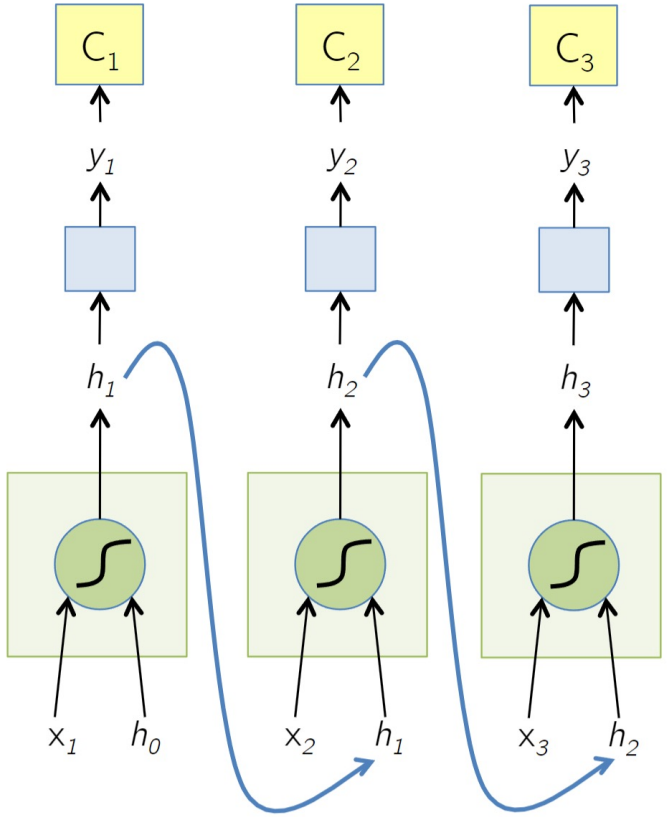
深度学习—循环神经网络(RNN)

循环神经网络是专门对序列数据进行建模的的神经网络。



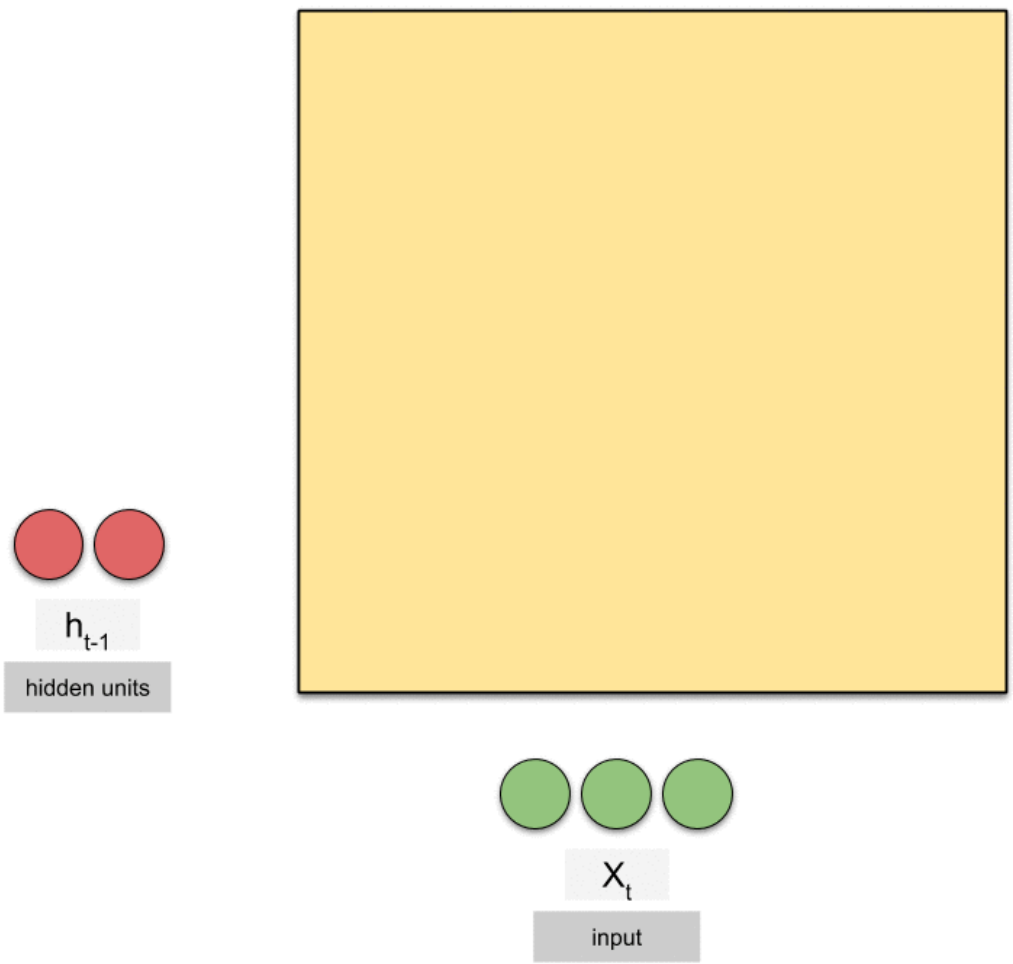
简单RNN单元

$$\boldsymbol{h}_t = \tanh \boldsymbol{W} \begin{pmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{pmatrix}$$



时序上的参数共享

深度学习——循环神经网络(RNN)



深度学习——循环神经网络(RNN)

在实际训练过程中，RNN存在梯度爆炸的问题。

RNN反向传播中隐藏层梯度

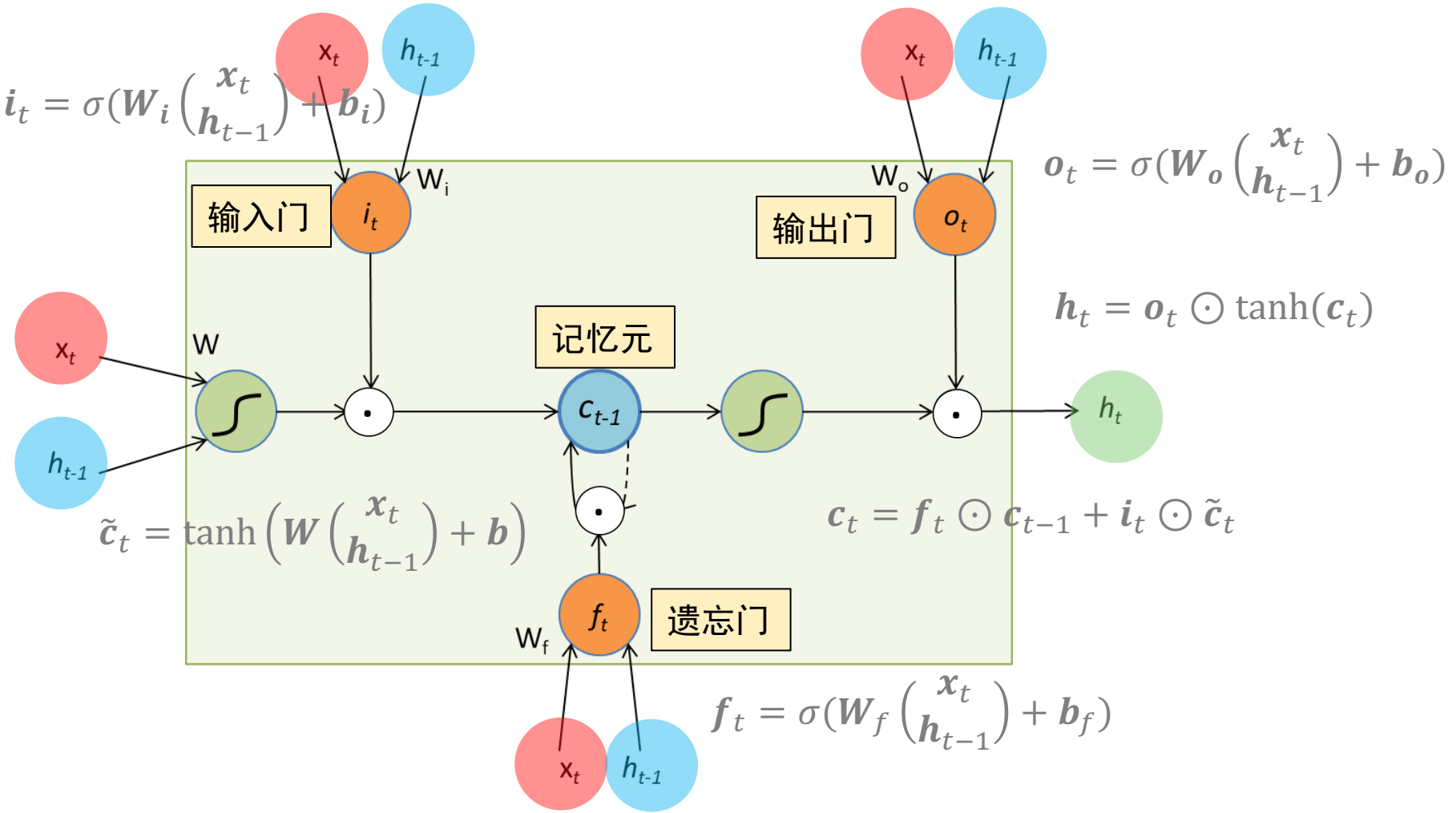
$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i} = \prod_{k=i}^{t-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} = \prod_{k=i}^{t-1} \boldsymbol{W}_{hh}^T \odot \boldsymbol{\sigma}'_k$$

- 当i很小时，矩阵高次幂带来的不稳定性和激活函数的导数连乘带来的指数速度的衰减有可能导致偏导数$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_i}$的爆炸

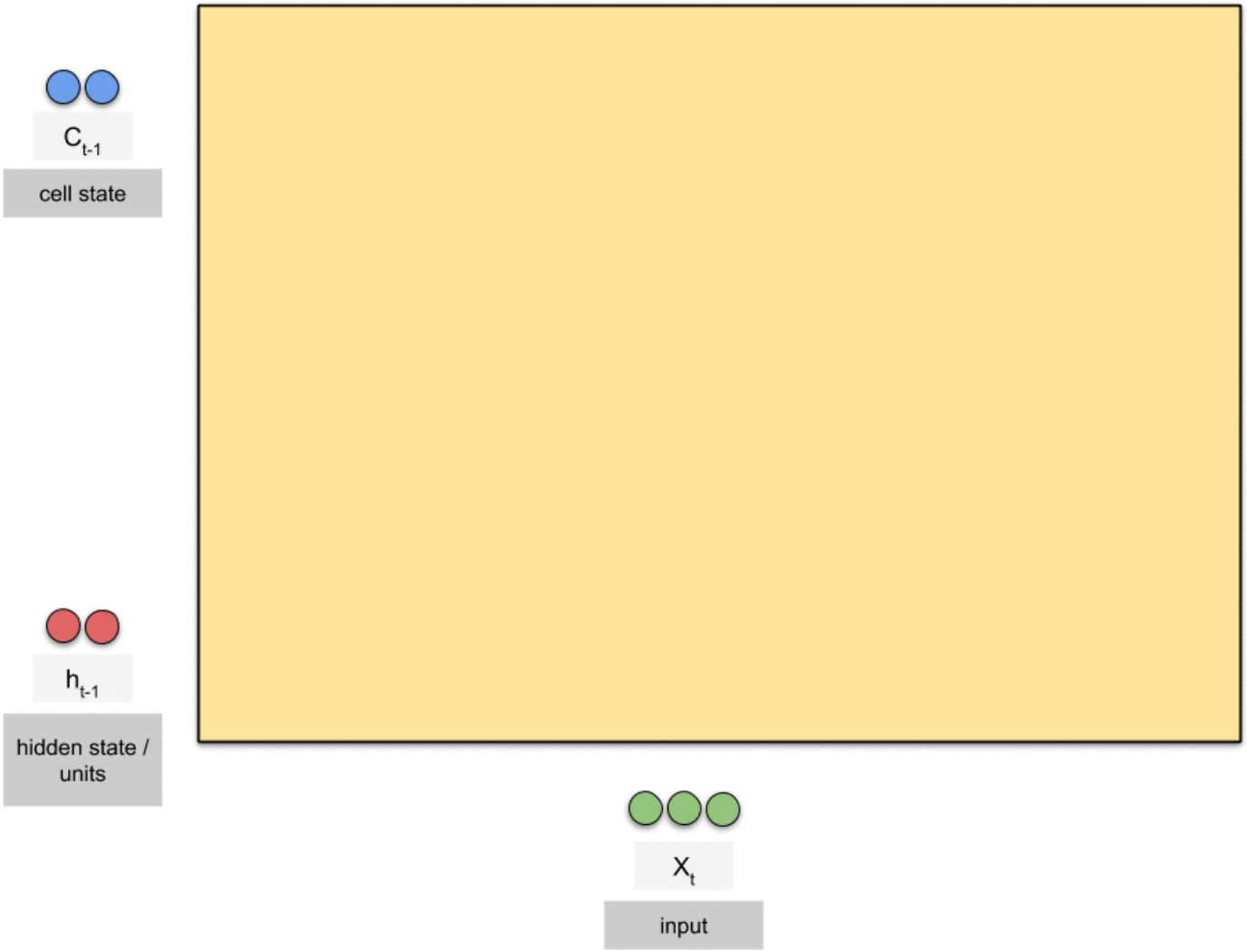
- 解决梯度爆炸
  - 权重衰减
  - 梯度截断

深度学习—循环神经网络(RNN)

长短期记忆神经网络(LSTM)



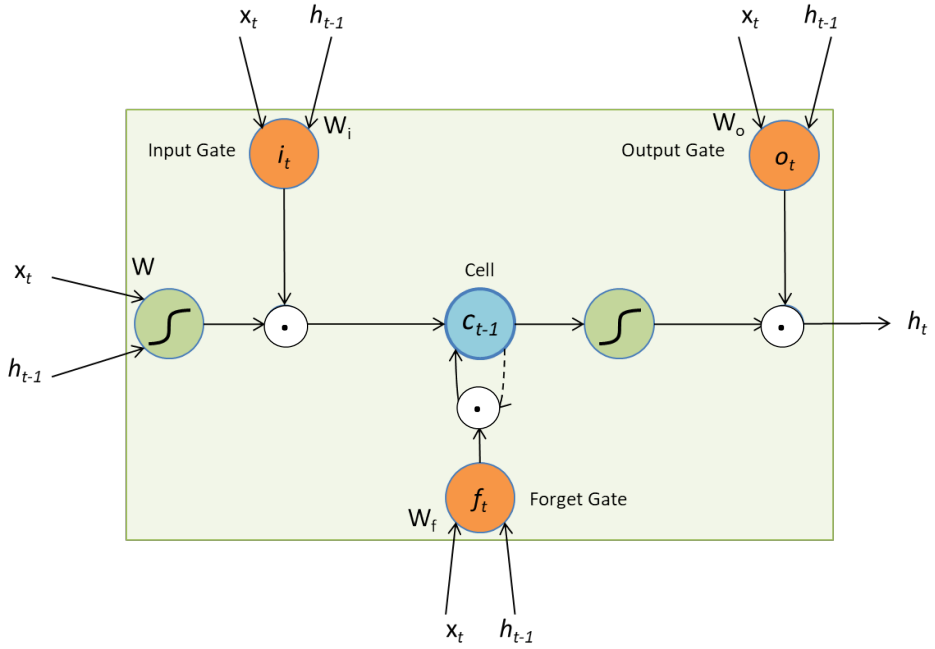
深度学习——循环神经网络(RNN)



深度学习——循环神经网络(RNN)

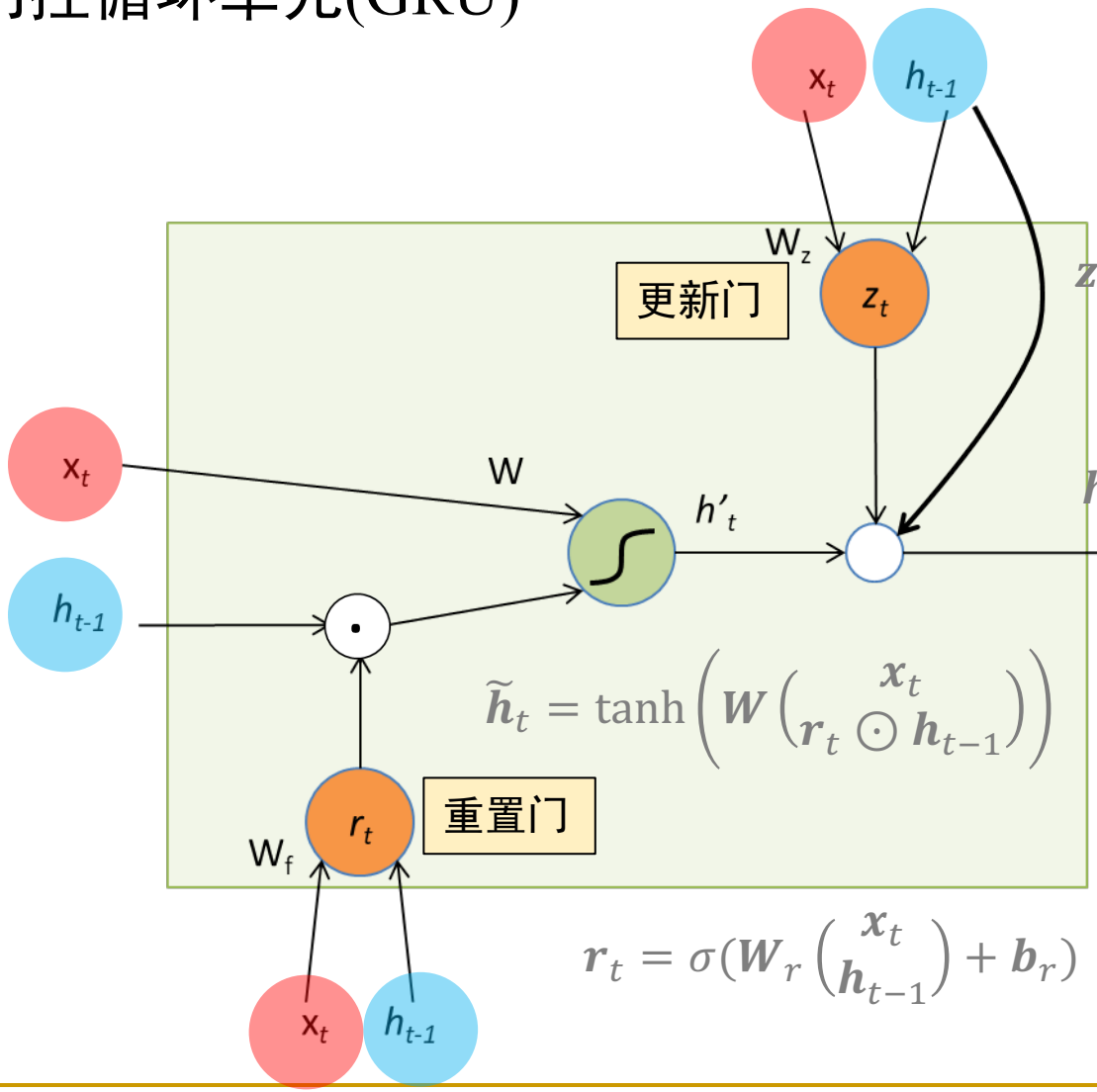
- 长期记忆
  - 模型权重参数
  - 在训练结束后长期存在
- 短期记忆
  - 记忆单元的参数
  - 在单次序列计算过程中存在

- LSTM具有较长的短期记忆（Long Short-Term Memory)。



深度学习——循环神经网络(RNN)

门控循环单元(GRU)



$$z_t = \sigma(W_z \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_z)$$

$$h_t = (1 - z_t) \odot c_{t-1} + z_t \odot \tilde{h}_t$$

$$\tilde{h}_t = \tanh\left(W \begin{pmatrix} x_t \\ r_t \odot h_{t-1} \end{pmatrix}\right)$$

$$r_t = \sigma(W_r \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_r)$$

与LSTM的不同

- 记忆单元与隐藏状态合并
- 将遗忘门和输入门合并为更新门

参数更少，性能不减

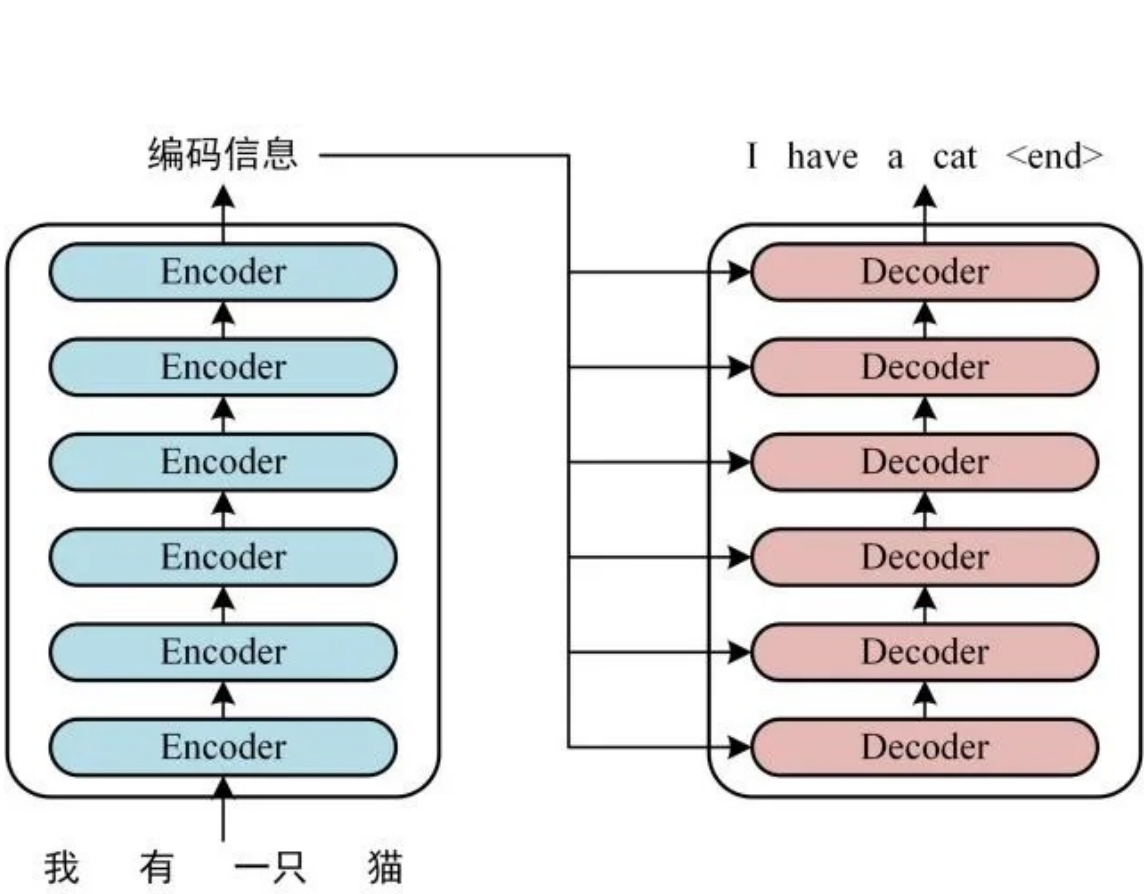
深度学习—循环神经网络(RNN)

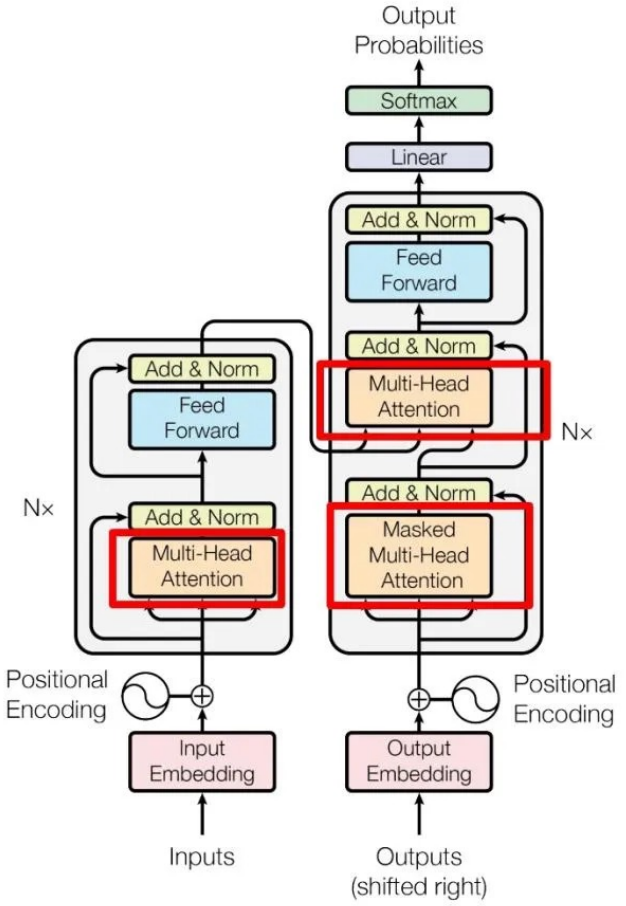
课程大纲

- 神经网络及训练
- 深度学习网络架构
- **大模型和表示收敛**
- PyTorch使用

Transformer

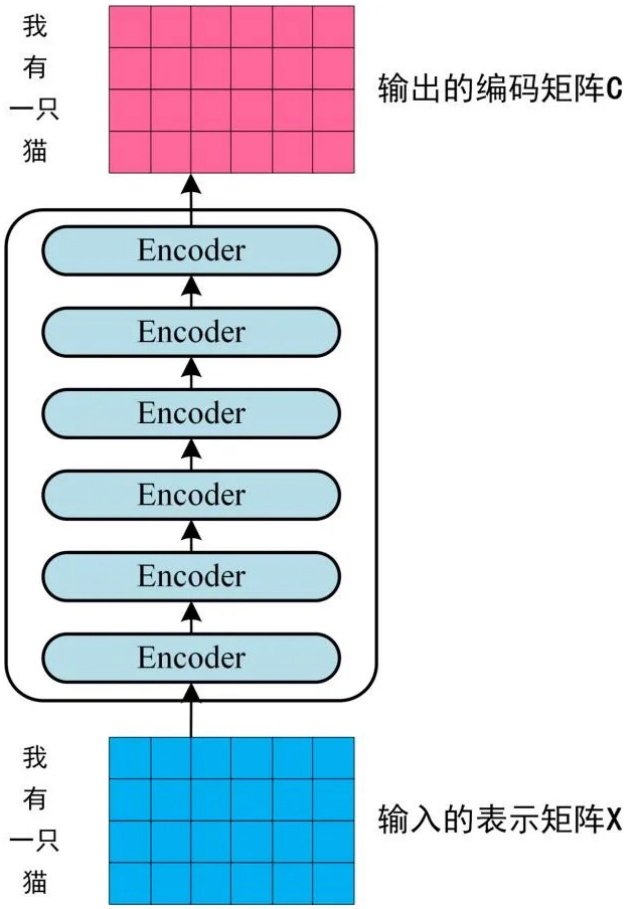
- 整体结构

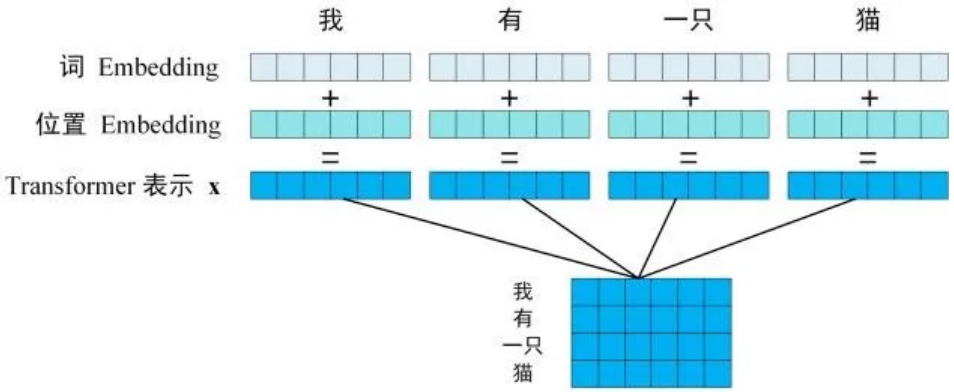




Transformer

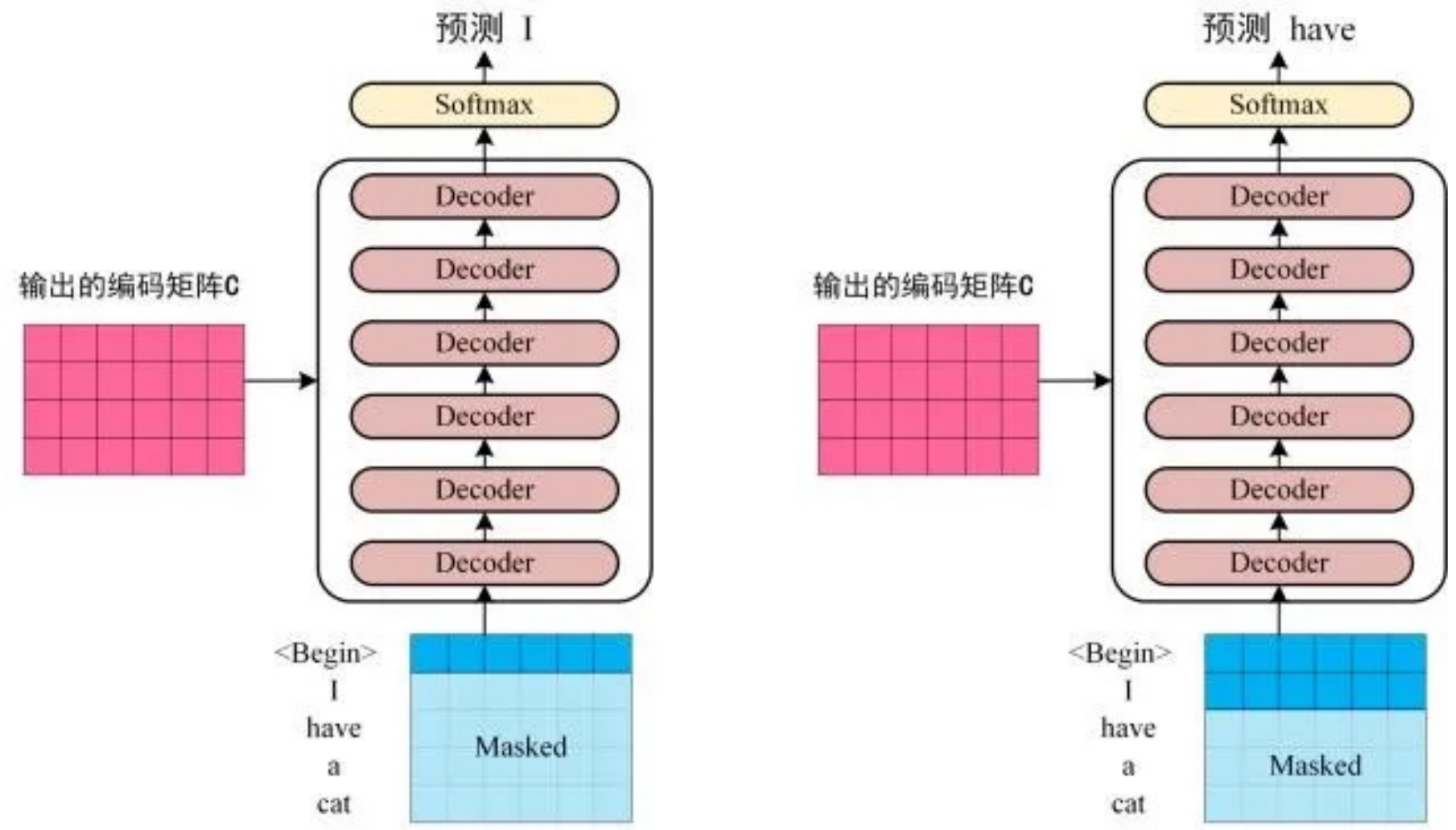
- 编码





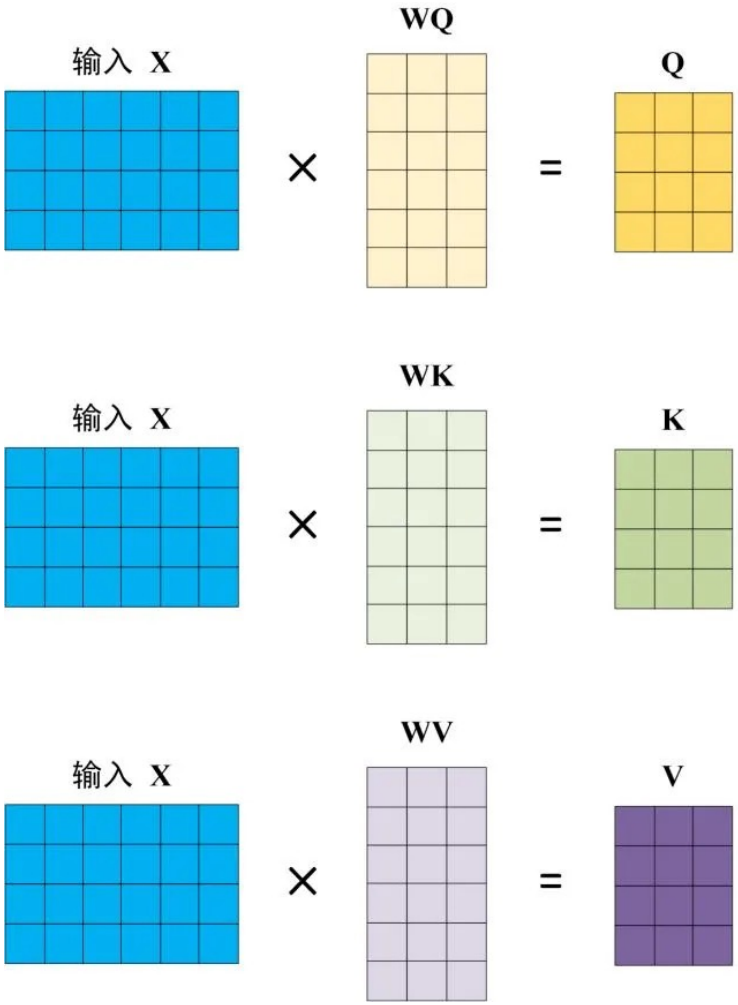
Transformer

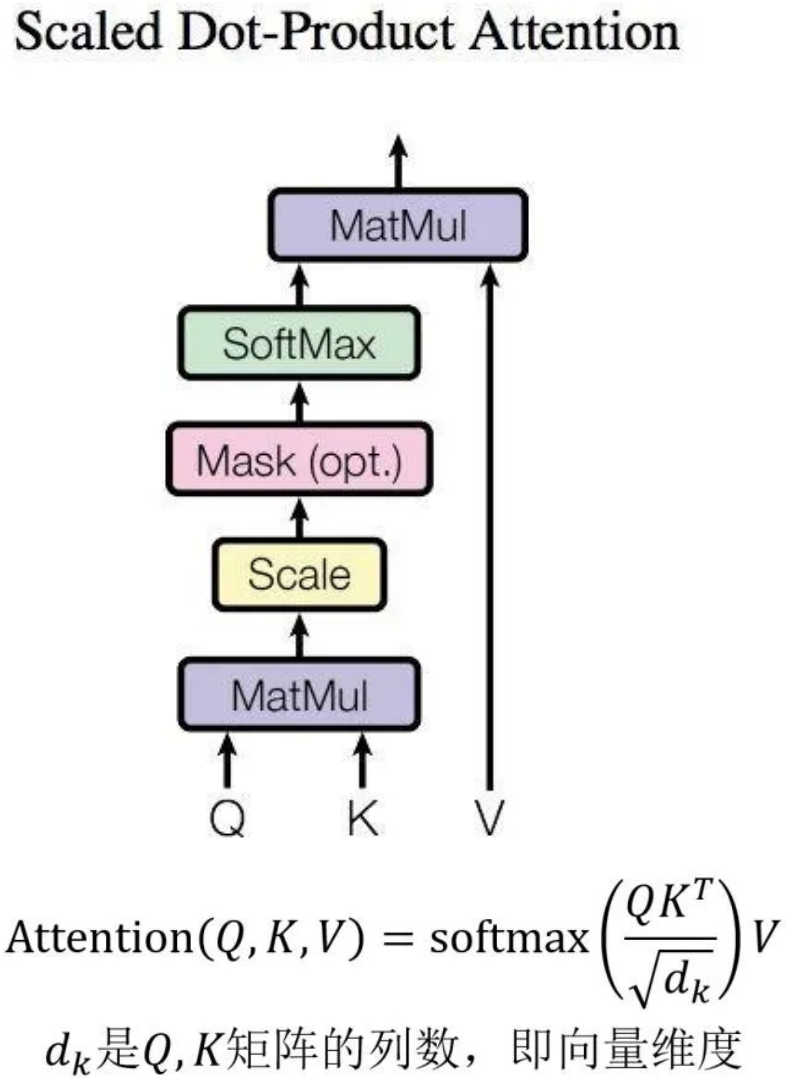
- 解码



Transformer

- 自注意力



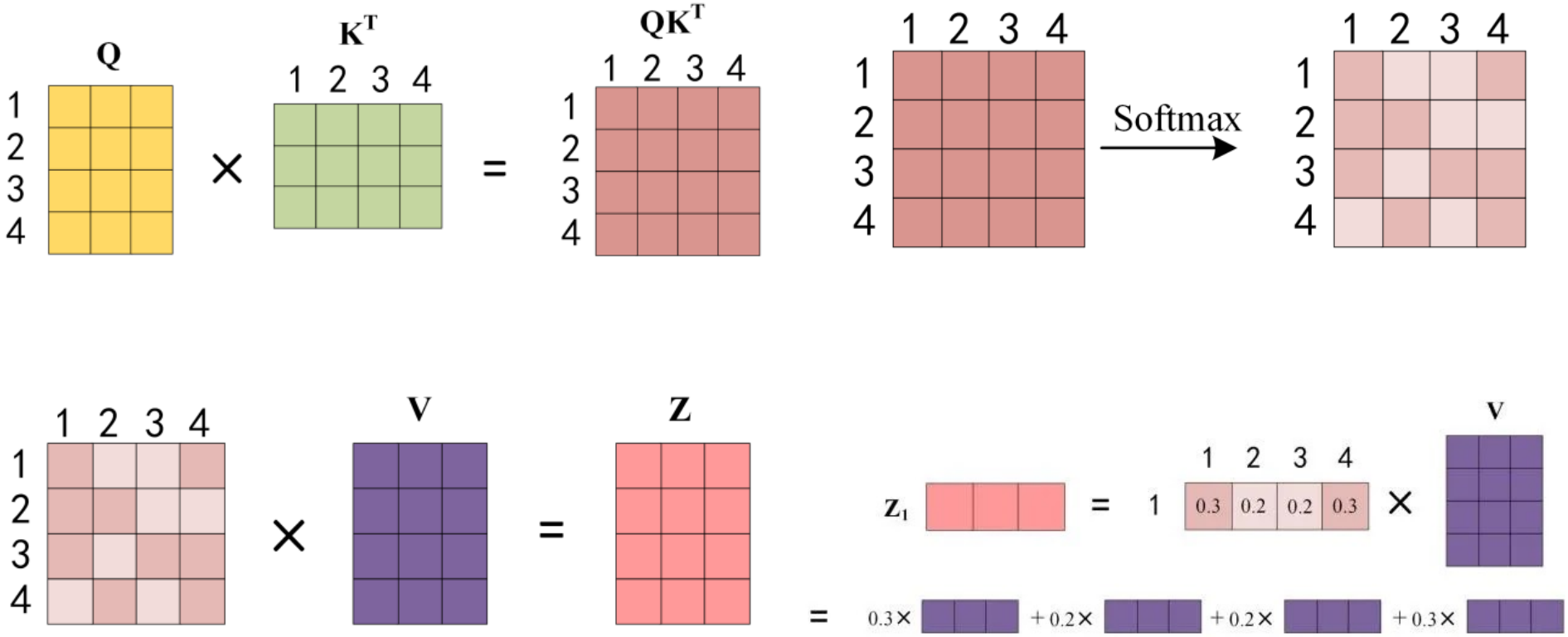


$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k是Q, K矩阵的列数，即向量维度

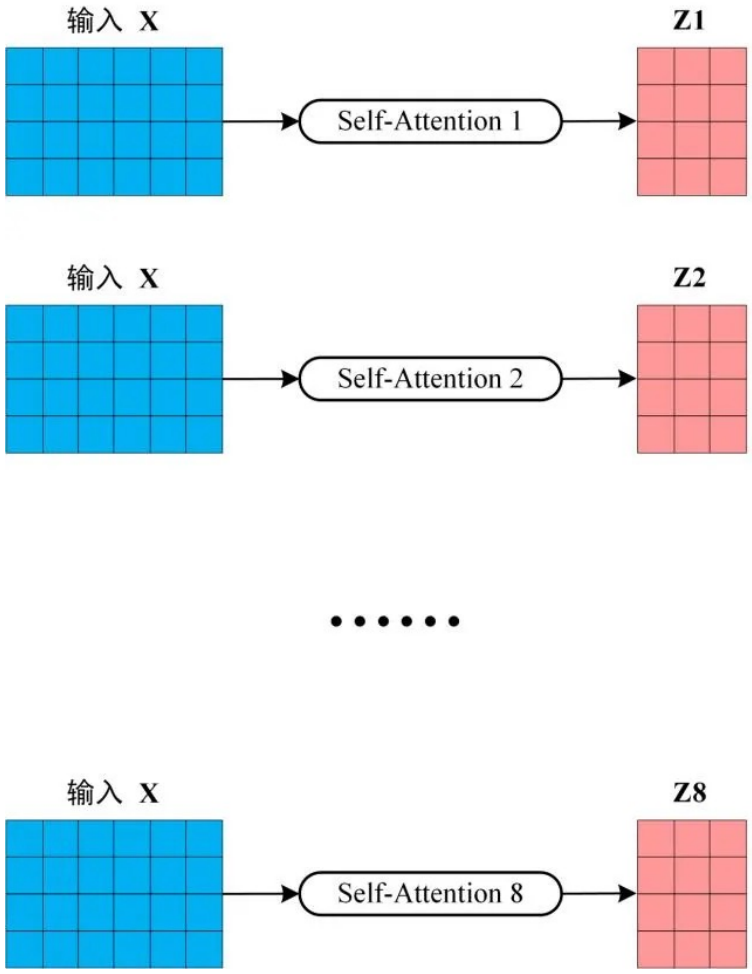
Transformer

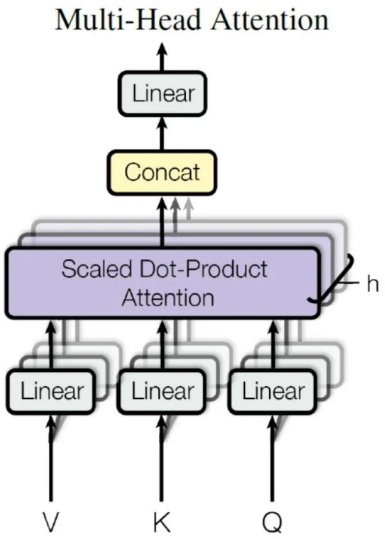
- 自注意力

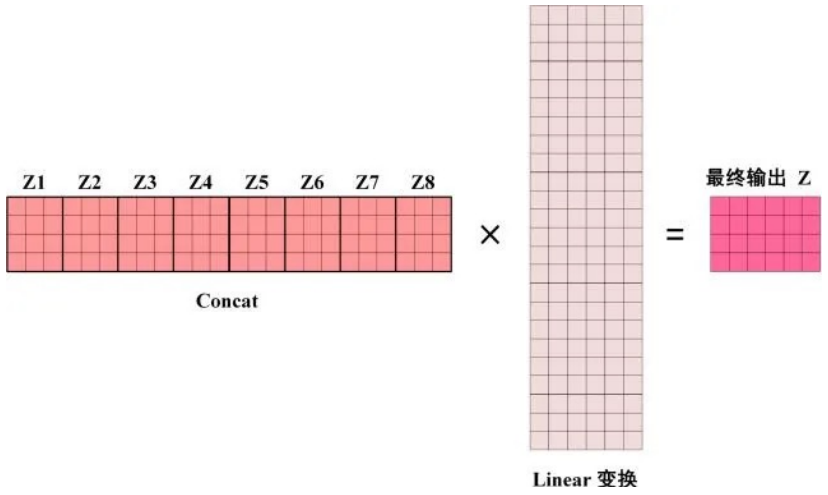


Transformer

- 多头自注意力

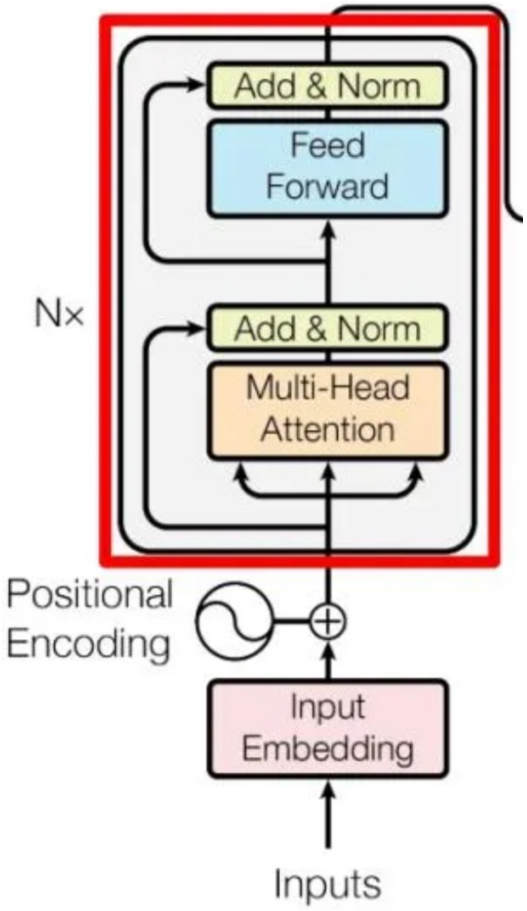






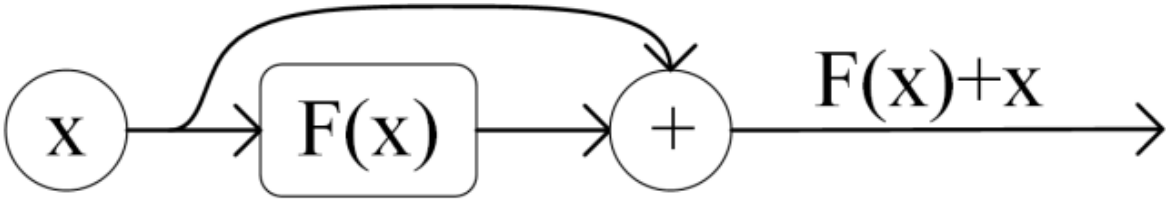
Transformer

- 编码块



$$\mathrm{LayerNorm}\left(X + \mathrm{MultiHeadAttention}(X)\right)$$

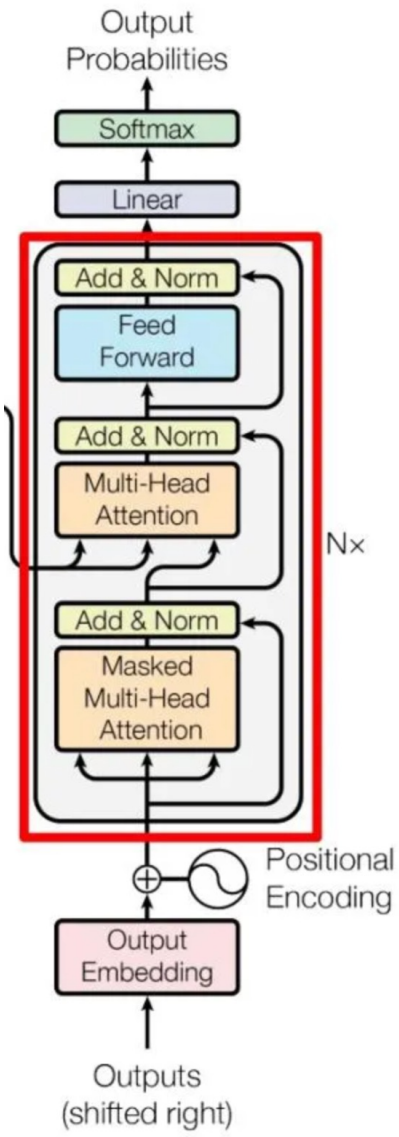
$$\mathrm{LayerNorm}\left(X + \mathrm{FeedForward}(X)\right)$$



$$\max(0, XW_1 + b_1)W_2 + b_2$$

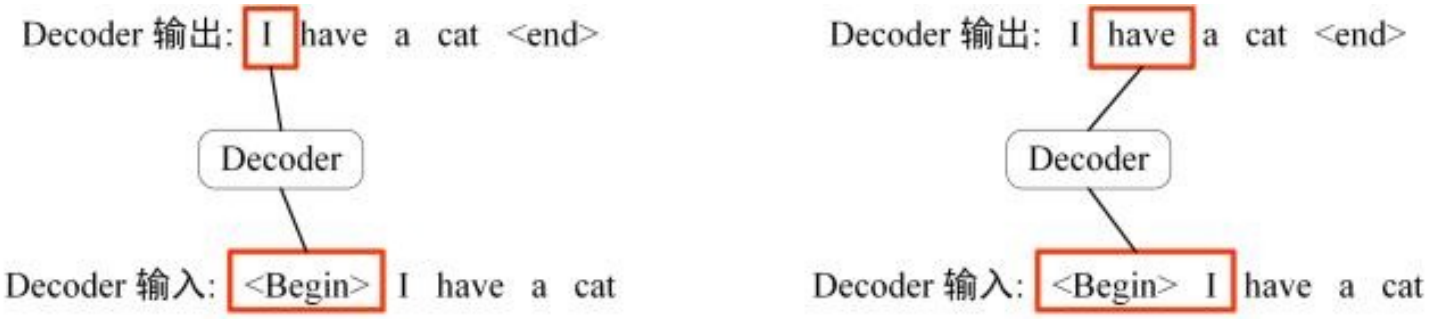
Transformer

- 解码块
  - 包含两个 Multi-Head Attention 层。
  - 第一个 Multi-Head Attention 层采用了 Masked 操作。
  - 第二个 Multi-Head Attention 层的**K, V**矩阵使用 Encoder 的**编码信息矩阵C**进行计算，而**Q**使用上一个 Decoder block 的输出计算。
  - 最后有一个 Softmax 层计算下一个翻译单词的概率。



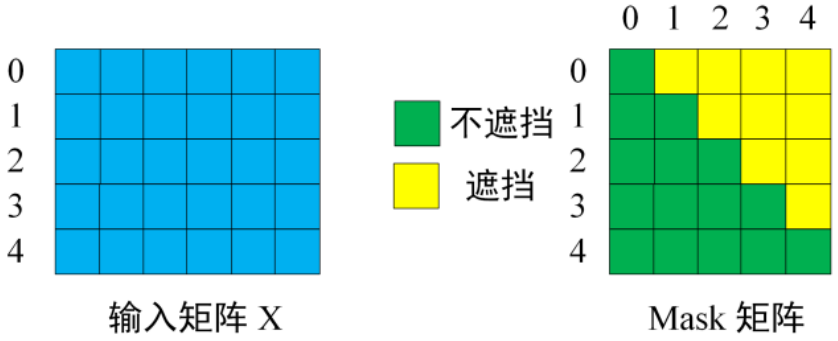
Transformer

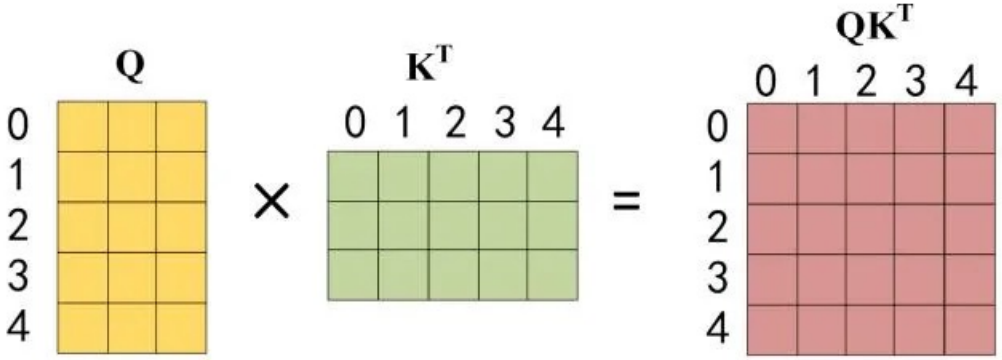
- 解码块—第一个 Multi-Head Attention 层



第一步：是 Decoder 的输入矩阵和 **Mask** 矩阵

第二步：接下来的操作和之前的 Self-Attention 一样，通过输入矩阵**X**计算得到**Q,K,V**矩阵。然后计算**Q**和$\mathbf{K}^T$的乘积



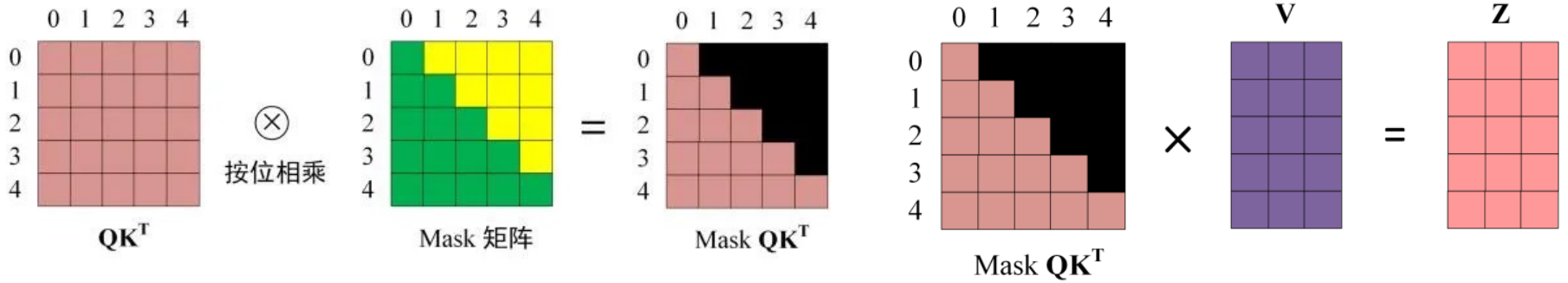


Transformer

- 解码块—第一个 Multi-Head Attention 层

第三步：在得到QK^T之后需要进行 Softmax，计算 attention score

第四步：使用 Mask QK^T与矩阵V相乘，得到输出 Z



第五步：通过上述步骤就可以得到一个 Mask Self-Attention 的输出矩阵Z_i，然后和 Encoder 类似，通过 Multi-Head Attention 拼接多个输出Z_i，然后计算得到第一个 Multi-Head Attention 的输出Z，Z与输入X维度一样。

Transformer

- 解码块—第二个 Multi-Head Attention 层

根据 Encoder 的输出 **C**计算得到 **K, V**，

根据上一个 Decoder block 的输出 **Z** 计算 **Q** (如果是第一个 Decoder block 则使用输入矩阵 **X** 进行计算)，

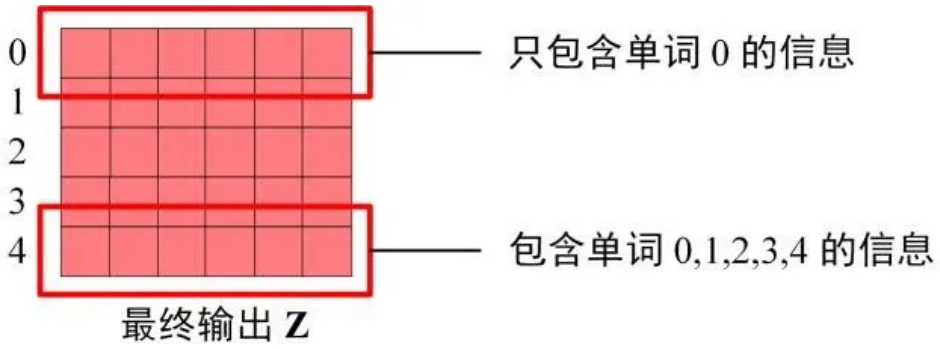
后续的计算方法与之前描述的一致。

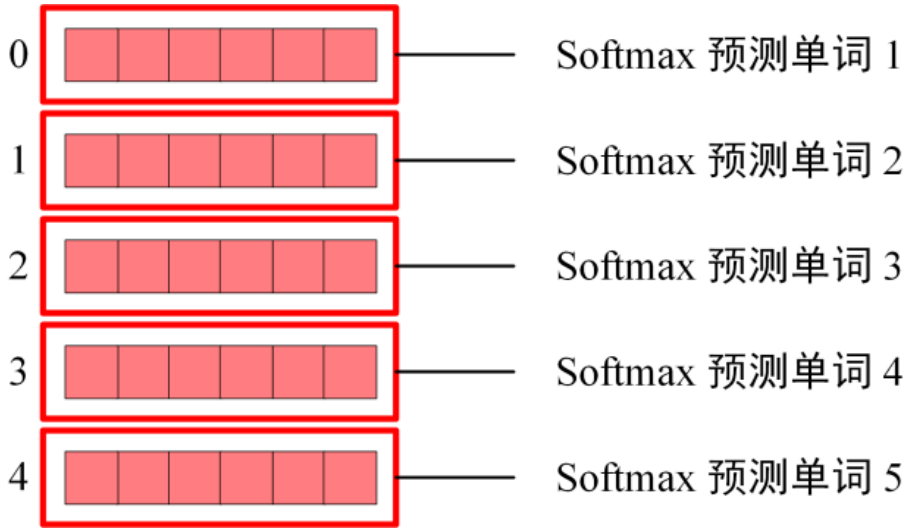
Transformer

- 解码块—**Softmax 预测输出单词**

因为 Mask 的存在，使得单词 0 的输出 Z_0 只包含单词 0 的信息

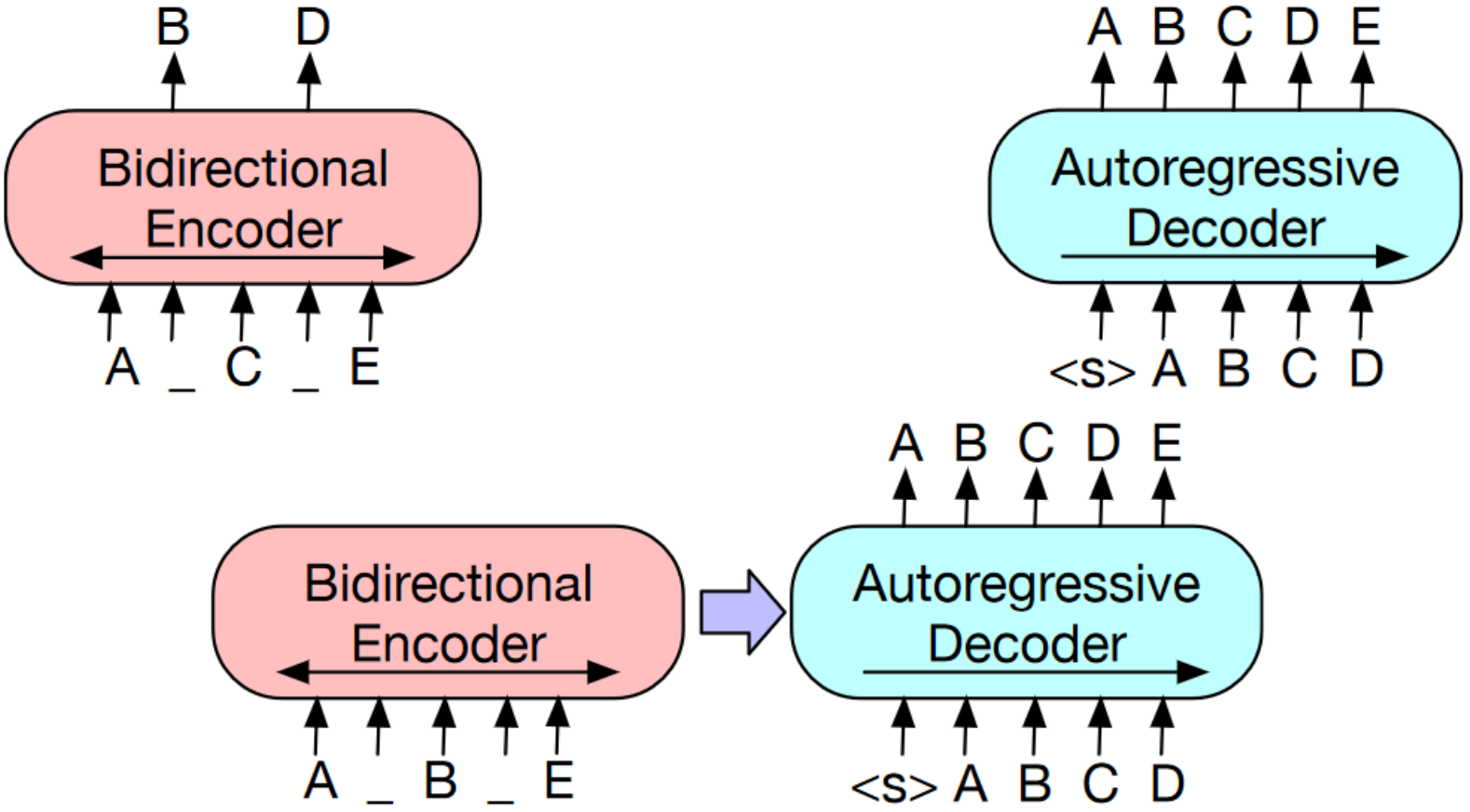
Softmax 根据输出矩阵的每一行预测下一个单词：





Transformer

- 变种： Encoder, Decoder, Encoder-Decoder



Transformer

小结：

- Transformer 与 RNN 不同，可以比较好地并行训练。
- Transformer 本身是不能利用单词的顺序信息的，因此需要在输入中添加位置 Embedding，否则 Transformer 就是一个词袋模型了。
- Transformer 的重点是 Self-Attention 结构，其中用到的 **Q, K, V** 矩阵通过输出进行线性变换得到。
- Transformer 中 Multi-Head Attention 中有多个 Self-Attention，可以捕获单词之间多种维度上的相关系数 attention score。

表示收敛

- 核心观点：不同大模型的表示趋于收敛。

The Platonic Representation Hypothesis

Minyoung Huh [*1] Brian Cheung [*1] Tongzhou Wang [*1] Phillip Isola [*1]

The Platonic Representation Hypothesis

Neural networks, trained with different objectives on different data and modalities, are converging to a shared statistical model of reality in their representation spaces.

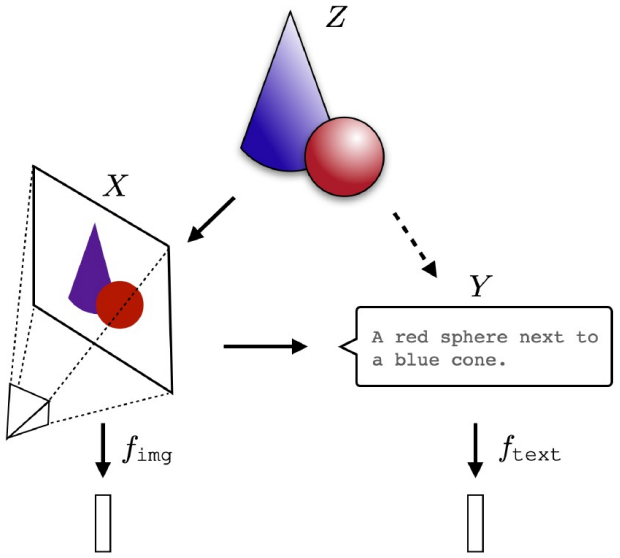


Figure 1. **The Platonic Representation Hypothesis:** Images (X) and text (Y) are projections of a common underlying reality (Z). We conjecture that representation learning algorithms will converge on a shared representation of Z, and scaling model size, as well as data and task diversity, drives this convergence.

https://arxiv.org/abs/2405.07987

https://phillipi.github.io/prh/

表示收敛

- 假设

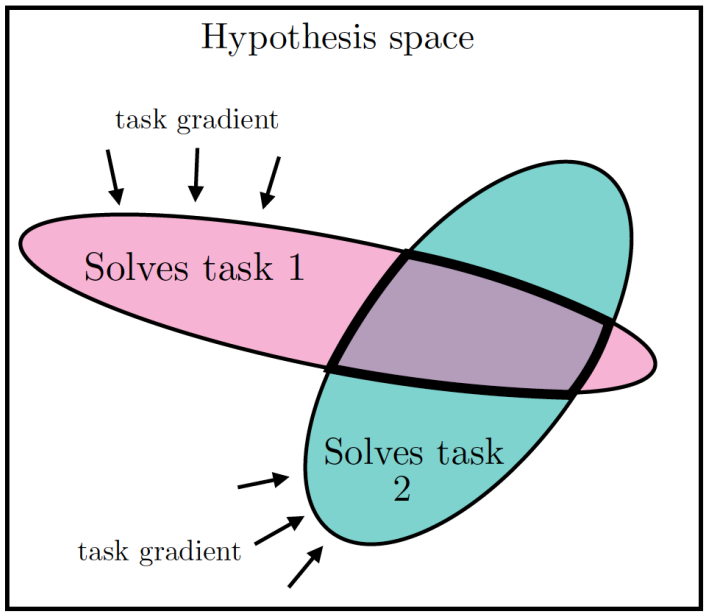


Figure 6. **The Multitask Scaling Hypothesis:** Models trained with an increasing number of tasks are subjected to pressure to learn a representation that can solve all the tasks.

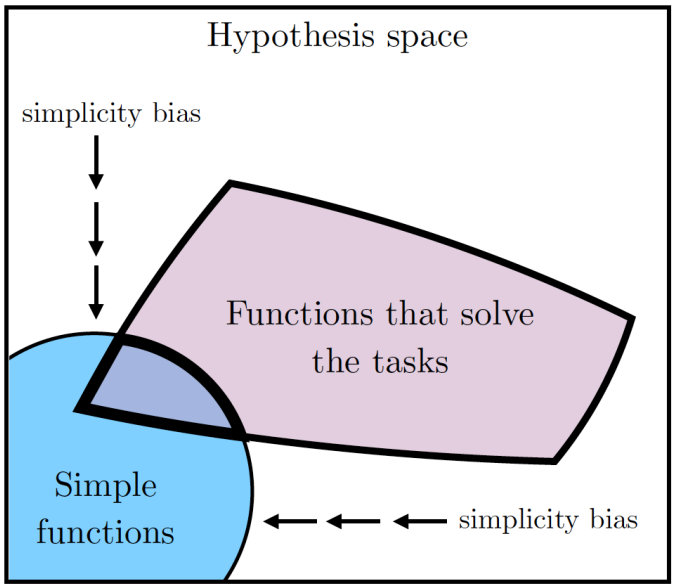


Figure 7. **The Simplicity Bias Hypothesis:** Larger models have larger coverage of all possible ways to fit the same data. However, the implicit simplicity biases of deep networks encourage larger models to find the simplest of these solutions.

OpenAI 工程师 James Betker(DALL-E 一作)

“我在 OpenAI 工作已经快一年了。这段时间里，我训练了很多生成式 AI 模型，比任何人能想到的还要多。

每当我花了几个小时，观察和调整各种模型配置和参数时，有一件事让我印象深刻，那就是所有训练结果之间的相似性。

我越来越发现，**这些模型以令人难以置信的程度，向它们的语料集靠近。**

这表明在相同的语料集上训练足够长的时间，几乎每个具有足够权重和训练时间的模型都会收敛到同一点。足够大的扩散卷积网络会产生相同的结果。

这是一个令人惊讶的观察！

这意味着模型行为不是由架构、参数或优化器决定的。它由你的语料集决定，没有其他决定因素。其他一切因素都不过是为了有效计算以近似该语料集的手段。

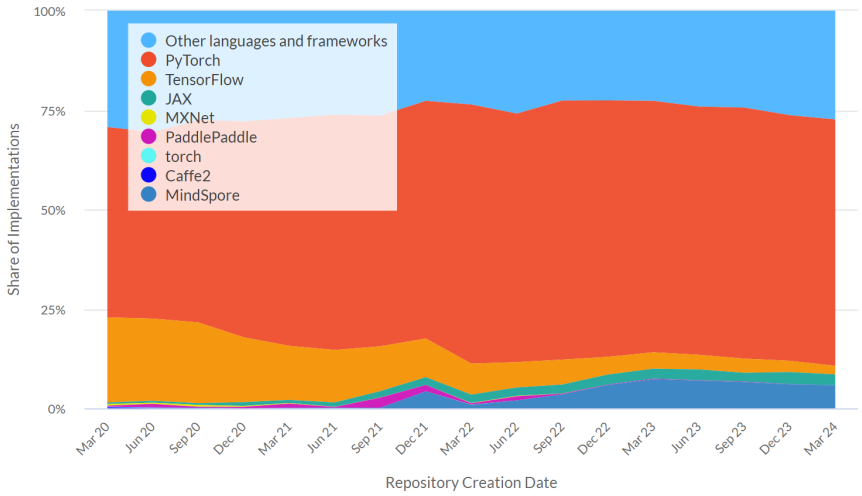
当你谈论 Lambda、ChatGPT、Bard 或Claude 时，指的并不是它们的模型，而是它们的语料集。”

课程大纲

- 神经网络及训练
- 深度学习网络架构
- 大模型和表示收敛
- **PyTorch使用**

PyTorch

PyTorch是由Meta AI (Facebook)人工智能研究小组开发的一种基于Python实现的深度学习库，目前被广泛应用于学术界和工业界。



Pytorch的优势

- **更加简洁：**相比于其他的框架，PyTorch的框架更加简洁，易于理解。
- **上手快：**掌握numpy和基本的深度学习知识就可以上手。
- **良好的文档和社区支持**
- **便于调试**

安装地址：https://pytorch.org/get-started/locally/

PyTorch—Tensor

```
import torch
```

```
# 从列表初始化
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

```
# 从numpy数组初始化
import numpy as np

np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
# 从另一个Tensor初始化
x_ones = torch.ones_like(x_data)
x_rand = torch.rand_like(x_data,
                         dtype=torch.float)
```

Tensor(张量)是PyTorch中数据处理的基本单位。Tensor可以通过列表、numpy数组以及另一个Tensor初始化。Tensor具有以下的属性:

- **Tensor.shape:** 张量的形状
- **Tensor.dtype:** 张量的数值类型
- **Tensor.device:** 存储张量的设备（cpu/cuda/…）

Tensor的索引同numpy数组类似。

X[2, 1], y[:, :2], …

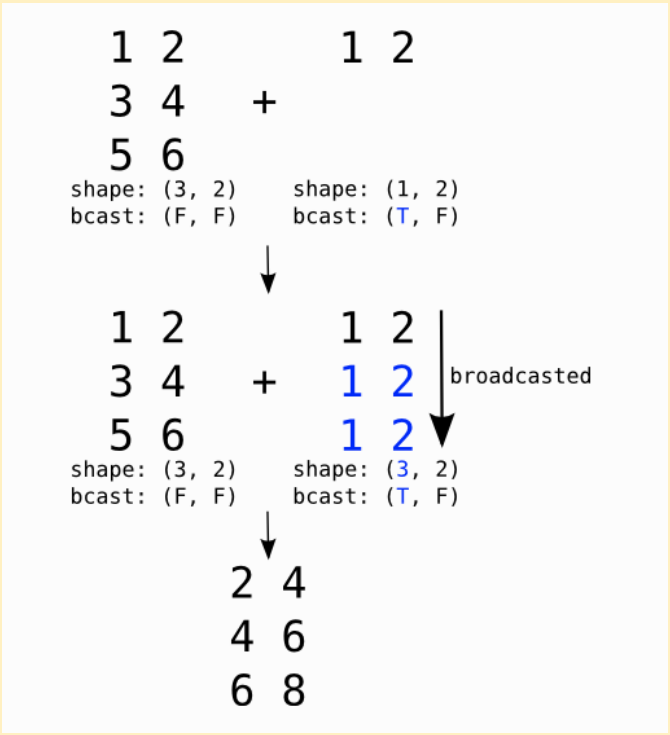
PyTorch—Tensor

```
# Tensor加法 四个值结果相同
y1 = a + b
y2 = torch.add(a, b)

y3 = a.copy()
y3.add_(b)

y4 = torch.rand_like(a)
torch.add(a, b, out=y4)
```

```
# Tensor乘法 四个值结果相同 a,b均为方阵
y1 = a @ b
y2 = a.matmul(b)

y3 = torch.rand_like(a)
torch.matmul(a, b, out=y3)
```

Tensor的广播机制(Boardcasting)

当对两个形状不同的 Tensor 按元素运算时，可能会触发广播机制：先适当复制元素使这两个 Tensor 形状相同后再按元素运算。



PyTorch—Dataset

- PyTorch中的数据加载通过Dataset+DataLoader的方式完成
- 自定义Dataset要求实现 **__init__**, **__getitem__** 和 **__len__**三个方法

```
import pandas as pd
from torch.utils.data import Dataset

class CustomDatasetFromCsv(Dataset):
    def __init__(self, csv_path):
        # 使用pandas读入csv
        self.full_data = pd.read_csv(csv_path, header=None)
        # 拆分输入与标签
        self.input_arr = np.asarray(self.full_data.iloc[:, :-1])
        self.label_arr = np.asarray(self.full_data.iloc[:, -1])
        self.data_len = len(self.data_full.index)

    def __getitem__(self, index):
        return (self.input_arr[index], self.label_arr[index])

    def __len__(self):
        return self.data_len
```

PyTorch—Dataset

通过DataLoader生成数据集的iterator

```
from torch.utils.data import DataLoader, random_split

dataset = CustomDatasetFromCsv("test.csv")

train_set, valid_set, test_set = random_split(dataset, [0.7, 0.1, 0.2])
# 设置批大小和数据读取进程数
batch_size = 64
num_workers = 4
train_loader = DataLoader(train_set, batch_size=batch_size,
                    num_workers=num_workers, shuffle=True, drop_last=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size,
                                    num_workers= num_workers, shuffle=True)
test_loader = DataLoader(test_set, batch_size=batch_size,
                                     num_workers=num_workers, shuffle=True)

example_input, example_label = next(iter(train_loader))
```

PyTorch—Module

```
import torch
from torch import nn

class MLP(nn.Module):
  # 声明带有模型参数的层，这里声明了两个全连接层
  def __init__(self, **kwargs):
    # 父类初始化
    super(MLP, self).__init__(**kwargs)
    self.hidden = nn.Linear(32, 64)
    self.activation = nn.ReLU()
    self.output = nn.Linear(64, 1)

  # 定义模型的前向计算，即如何根据输入x计算返回所需要的模型输出
  def forward(self, x):
    o = self.activation(self.hidden(x))
    return self.output(o)

net = MLP() # 实例化模型
print(net) # 打印模型
```

- Module 类是 torch.nn 模块里提供的一个模型构造类，是所有神经网络模块的基类。通过继承它来定义想要的模型。
- 反向传播函数**backward**是自动生成的

输出：

```
MLP(
  (hidden): Linear(in_features=32, out_features=64, bias=True)
  (activation): ReLU()
  (output): Linear(in_features=64, out_features=1, bias=True)
)
```

PyTorch—Module

可以通过Module自定义含模型参数的自定义层

```
class MyDictDense(nn.Module):
    def __init__(self):
        super(MyDictDense, self).__init__()
        self.params = nn.ParameterDict({
                'linear1': nn.Parameter(torch.randn(4, 4)),
                'linear2': nn.Parameter(torch.randn(4, 1))
        })
        # 新增
        self.params.update({'linear3': nn.Parameter(torch.randn(4, 2))})
    def forward(self, x, choice='linear1'):
        return torch.mm(x, self.params[choice])
```

对于每个Module，PyTorch会自动选择合适的参数进行初始化，也可选择使用torch.nn.init手动初始化

PyTorch—模型训练

PyTorch提供简便方法改变模型的状态

```
net.train()   # 训练状态，参数支持反向传播修改
net.eval()   # 验证/测试状态，参数冻结
```

torch.nn中提供了常用的损失函数

```
torch.nn.MSELoss()  # 均方误差
torch.nn.BCELoss()  # 二分类交叉熵
torch.nn.CrossEntropyLoss()   # 多分类交叉熵
```

torch.optim中提供了常用的优化器

```
torch.nn.SGD
torch.nn.Adam
torch.nn.RMSprop
...
```

PyTorch—模型训练

```
loss_fn = torch.nn.BCELoss()
optimizer = torch.optim.adam(net.parameters(), lr=0.001)

def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.
    for i, data in enumerate(train_loader):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % batch_size == batch_size - 1:
            last_loss = running_loss / batch_size
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.
    return last_loss
```

定义单个epoch的动作

PyTorch—模型训练

```
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
EPOCHS = 5
best_vloss = 1_000_000.
for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch_number + 1))
    net.train(True)
    avg_loss = train_one_epoch(epoch_number, writer)
    running_vloss = 0.0
    net.eval()
    with torch.no_grad():
        for i, vdata in enumerate(valid_loader):
            vinputs, vlabels = vdata
            voutputs = net(vinputs)
            vloss = loss_fn(voutputs, vlabels)
            running_vloss += vloss
    avg_vloss = running_vloss / (i + 1)
    print('LOSS train {} valid {}'.format(avg_loss, avg_vloss))
    # 保存效果最好的模型
    if avg_vloss < best_vloss:
        best_vloss = avg_vloss
        model_path = 'model_{}_{}'.format(timestamp, epoch_number)
        torch.save(net.state_dict(), model_path)
    epoch_number += 1
```

在每个epoch结束后冻结参数在验证集上进行测试（早停）

下午实验课

需要完成：

- 安装PyTorch
- 完成一个简单的RNN分类任务

谢 谢