

# 第五讲：神经网络模型优化

南京大学人工智能学院

申富饶

# 数据

- 简单神经网络难以在真实任务上发挥最大性能，因为数据和模型往往存在各种各样的问题，比较常见的**数据问题**有：

- 不同维度差异过大（数据中心偏置）

时间	1.1	2.1	3.1	4.1	5.1
用水量	2000	2300	2201	2102	2003
用电量	40	50	45	67	60

- 正负例样本不均衡

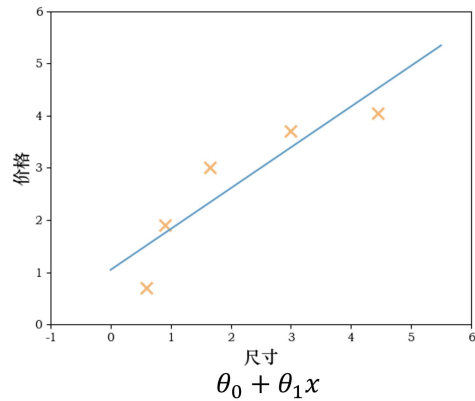
测试病人总数	患肺癌	不患肺癌
1520	18	1502

➤ .....

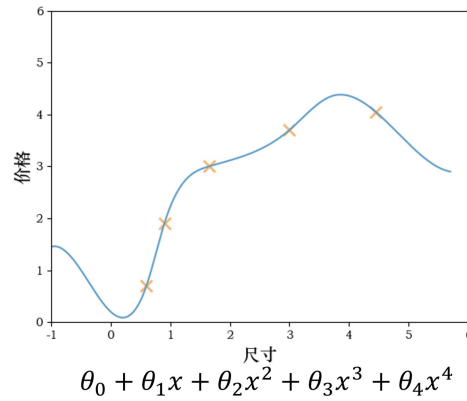
# 模型

- 比较常见的**模型问题**有：

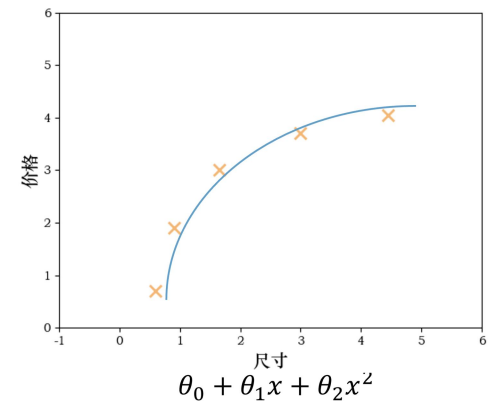
- 过/欠拟合



欠拟合



过拟合



正确拟合

- 梯度消失/梯度爆炸

- 模型过大，难以训练等

# 目录

CONTENTS

01. 学习率
02. 损失函数
03. 正则化
04. 归一化
05. 参数初始化
06. 网络预训练

01

# 学习率

# 学习率

在梯度下降法更新时，我们常常在计算的负梯度前乘以一个常数  $\eta$

$\eta$  在梯度下降算法中被称作为**学习率**或者**步长**，意味着我们可以通过  $\eta$  来控制每一步走的距离

- $\eta$  太小可能迟迟走不到最低点或无法跳出局部极小点
- $\eta$  太大的话会导致错过最低点，无法稳定收敛

# 学习率

## • 实验对比：以手写数据集为例

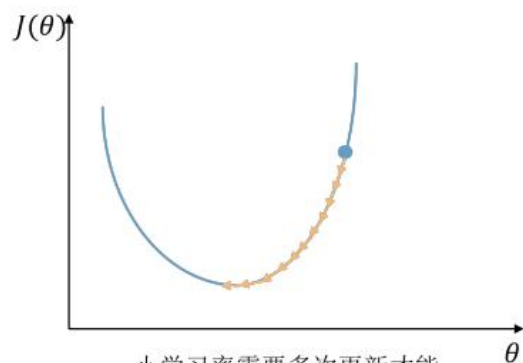
其他参数相同，使用随机梯度下降法迭代2000次，设置学习率为0.0001、0.001、0.01、0.1、0.2、0.5，对比实验效果为：

学习率	0.0001	0.001	0.01	0.1	0.2	0.5
错误数量	229	46	43	32	40	89
准确率	75.79%	95.13%	95.45%	96.62%	95.77%	90.59%

- 学习率较小时准确率不高，因为较小学习率无法在2000次迭代内完成收敛，因此，较小的学习率一般要配备较大的迭代次数以保证其收敛；
- 较大的学习率可以让MLP在2000次迭代内快速收敛到最优解；
- 但学习率过大时，可能会越过最优值，造成无法收敛；

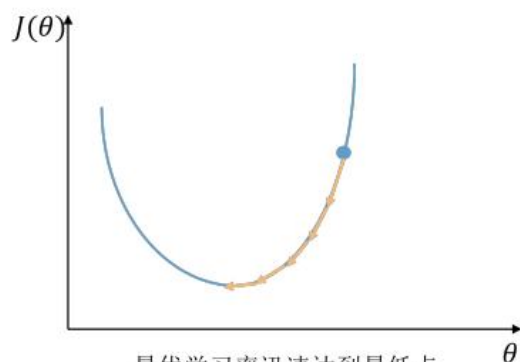
# 学习率

在梯度下降方法中，学习率如果过小则收敛速度太慢，如果过大可能难以收敛。



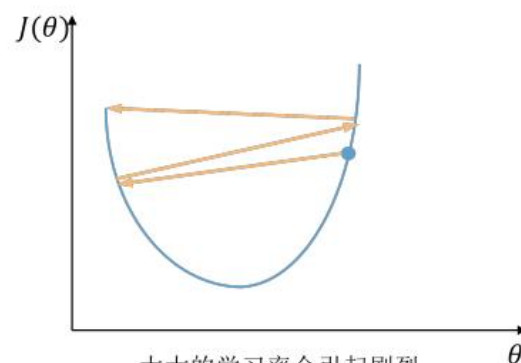
小学习率需要多次更新才能达到最低点

(a) 学习率过小



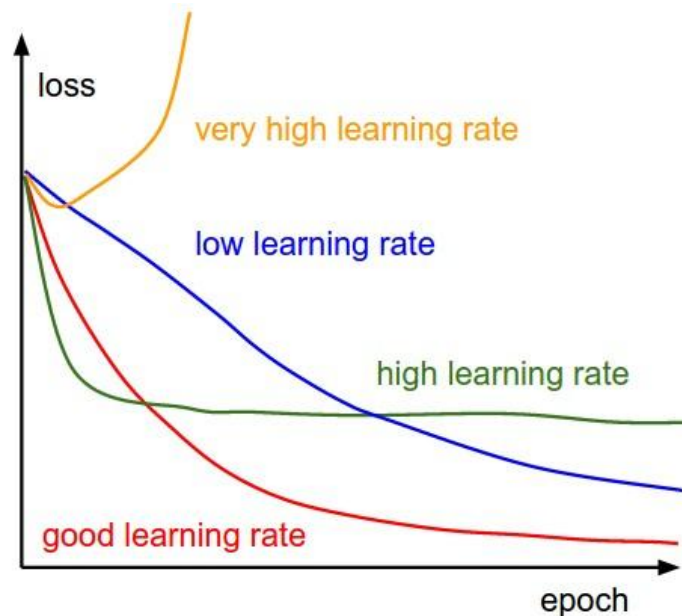
最优学习率迅速达到最低点

(b) 学习率适中



太大的学习率会引起剧烈的更新，从而导致发散

(c) 学习率过大





# 学习率调整

学习率  $\eta$  是神经网络优化时的重要超参数

在学习过程中维持恒定的学习率难以获取最优结果，需要动态的调整学习率的大小

## 常用的学习率调整方法：

- **固定衰减**      分段常数衰减、逆时衰减、指数衰减、余弦衰减
- **学习率预热**    逐渐预热
- **周期变化**      循环学习率、带热重启的随机梯度下降
- **自适应**        AdaGrad、RMSProp、Adam

# 学习率衰减

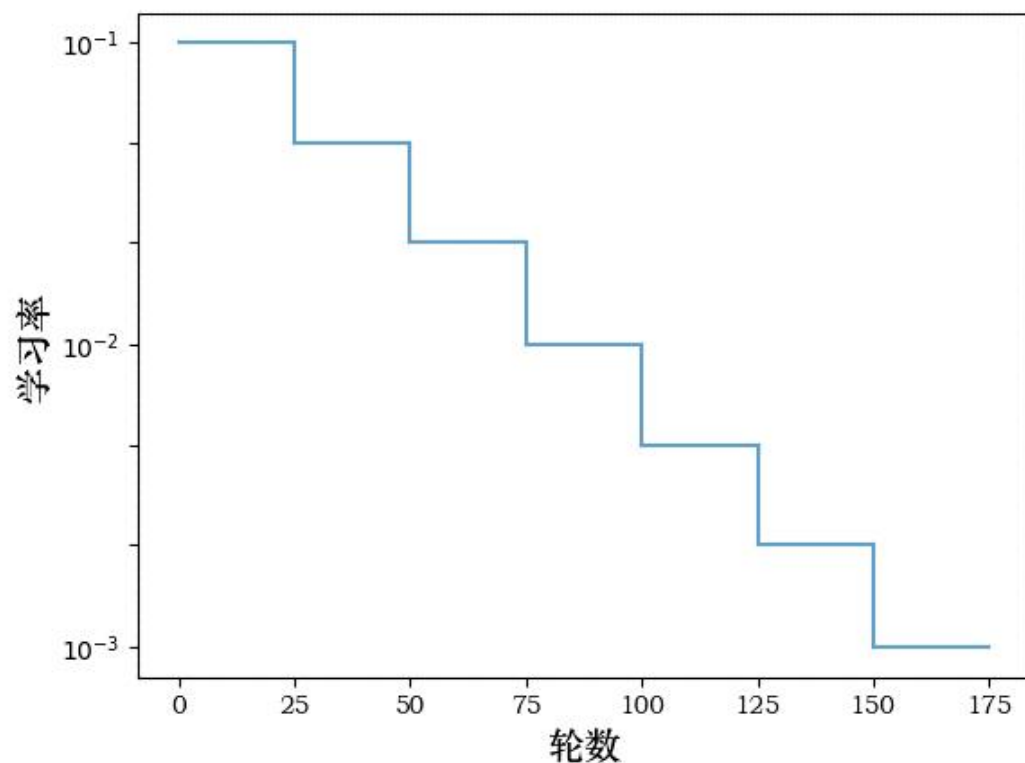
通常在一开始我们会把学习率设置的大一些来保证收敛速度，当收敛到最优  
点附近时，应该让学习率变小一些，避免震荡，这种学习率调整的方式称为  
**学习率衰减**（Learning Rate Decay）或者**学习率退火**（Learning Rate  
Annealing）。

假设初始的学习率为 $\eta_0$ ，在  $t$  次迭代时的学习率为 $\eta_t$ ，常见的固定衰减方  
式有以下几种：

（1）分段常数衰减 （2）逆时衰减 （3）指数衰减 （4）自然指数衰减

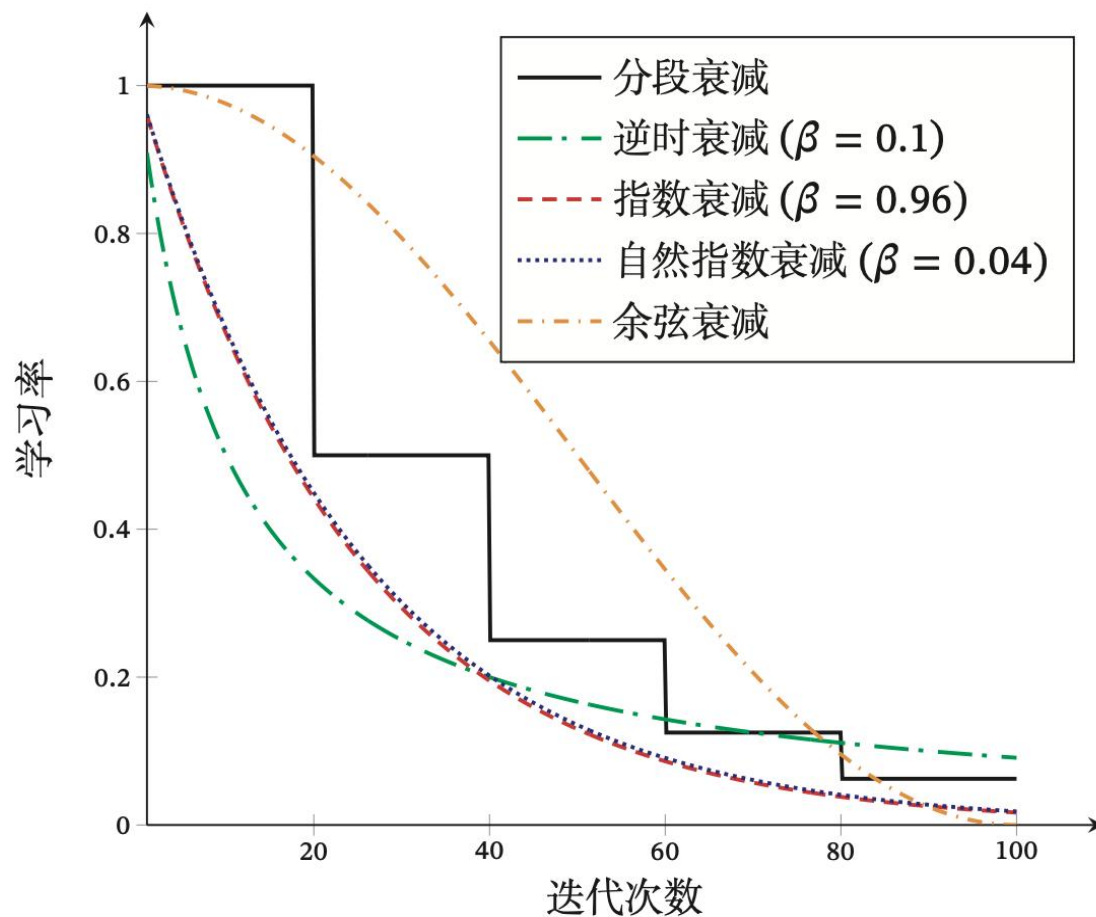
# 学习率衰减：分段常数衰减

**分段常数衰减 (Step decay):** 我们可以定义每经过  $T_1, T_2, \dots, T_n$  次迭代, 学习率将调整为  $\eta_1, \eta_2, \dots, \eta_n$ , 其中每个  $T_i$  和  $\eta_i$  根据经验设置, 并且  $\eta_1$  到  $\eta_n$  逐渐减小。



# 其它学习率固定衰减方法

- 逆时衰减:  $\eta_t = \eta_0 \frac{1}{1 + \beta \times t}$
- 指数衰减:  $\eta_t = \eta_0 \beta^t$
- 自然指数衰减:  $\eta_t = \eta_0 \exp(-\beta \times t)$
- 余弦衰减:  $\eta_t = \frac{1}{2} \eta_0 (1 + \cos(\frac{t\pi}{T}))$



# 学习率预热

由于刚开始训练时，模型的权重是随机初始化的，此时若选择一个较大的学习率，可能带来模型的不稳定(振荡)，为了让初始阶段的学习更加稳定，一种**学习率预热**（Learning Rate Warmup）的方法被提出。

基本思路：

- 在初始的几轮，采用较小的学习率
- 梯度下降到一定程度之后，再恢复到初始设置的学习率

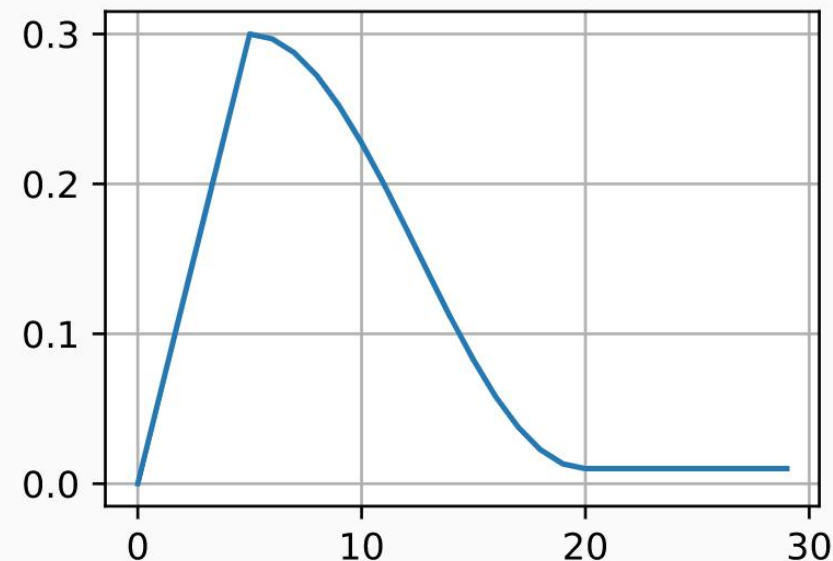
# 学习率预热： 逐渐预热

## 逐渐预热 (Gradual Warmup)

$$\eta_t = \frac{t}{T} \eta_0 \quad 1 \leq t \leq T.$$

其中  $T$  为预热的迭代数，初始学习率为  $\eta_0$ ；

在预热过程结束后，也可以再选择一种学习率衰减方法来逐渐降低学习率。



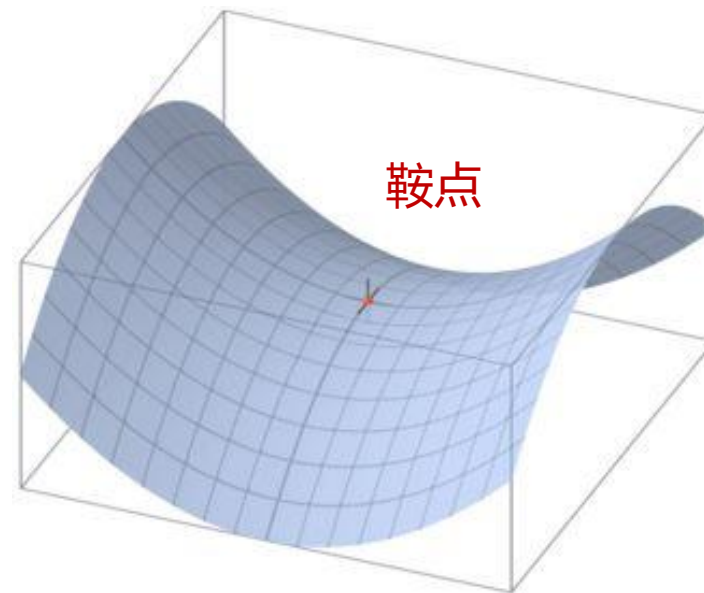
# 周期性学习率调整

梯度下降方法容易陷入局部最小值或鞍点，这个时候根据经验应当适当增加学习率来逃离。

虽然这有可能损失网络的收敛稳定性，但是长远来看能找到更好的局部最优解。

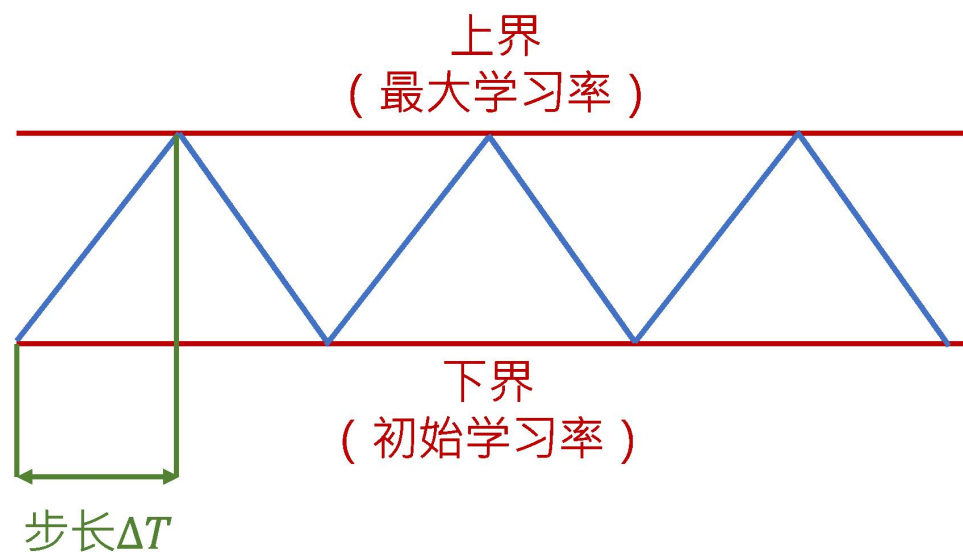
因此提出了周期性学习率调整的方法：

- 循环学习率(CLR)
- 带热重启的随机梯度下降(SGDR)



# 周期性学习率调整：CLR

**循环学习率(CLR):** 让学习率在一个区间内周期性的增大和缩小。

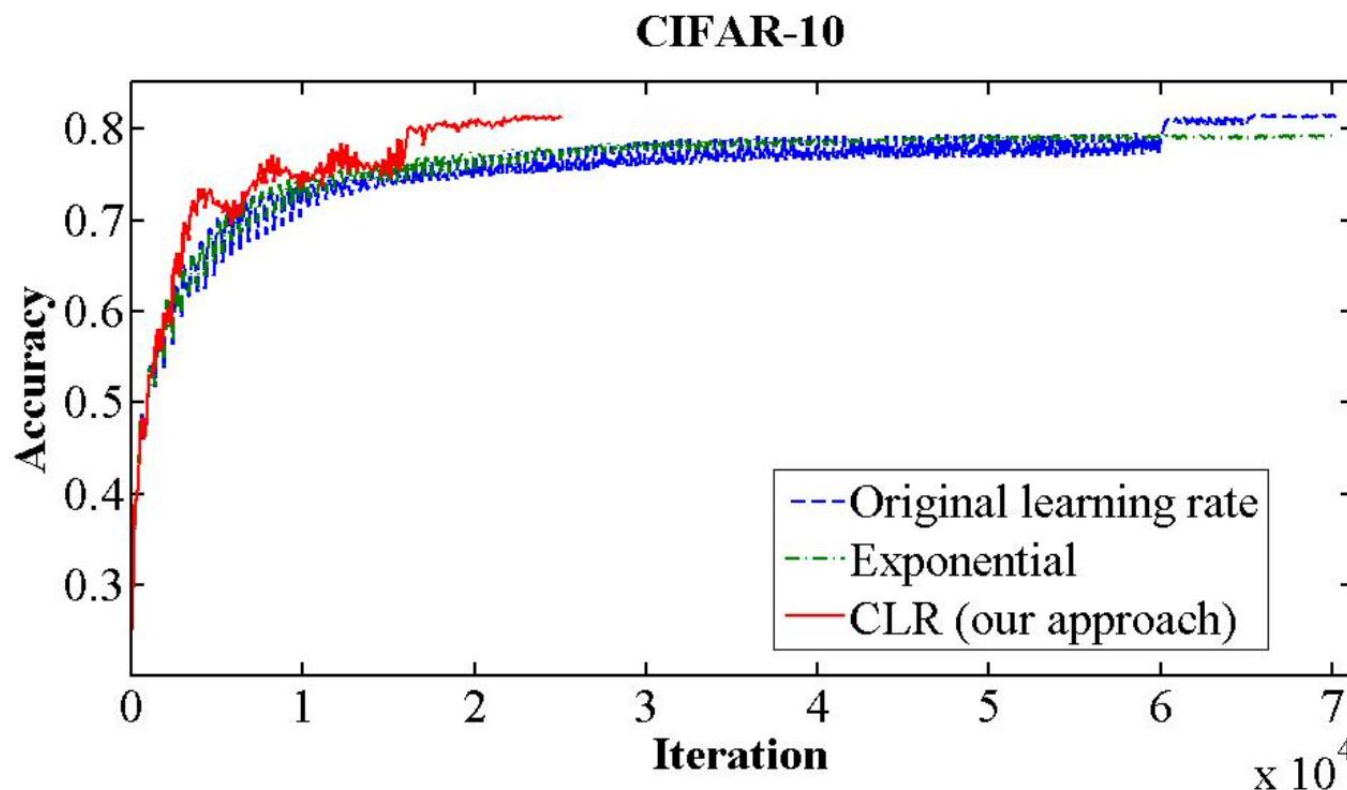


如图所示，通常可以使用线性缩放来调整学习率，称为三角循环学习率。假设每个循环周期的长度相等都为  $2\Delta T$ ，其中前  $\Delta T$  步为学习率线性增大阶段，后  $\Delta T$  步为学习率线性缩小阶段。



# 周期性学习率调整：CLR

**例：**对于CIFAR10数据集，传统的LR更新方法需要70000次迭代后才能达到81.4%的最终准确率，CLR在25000次迭代内就能达到同样的水平。

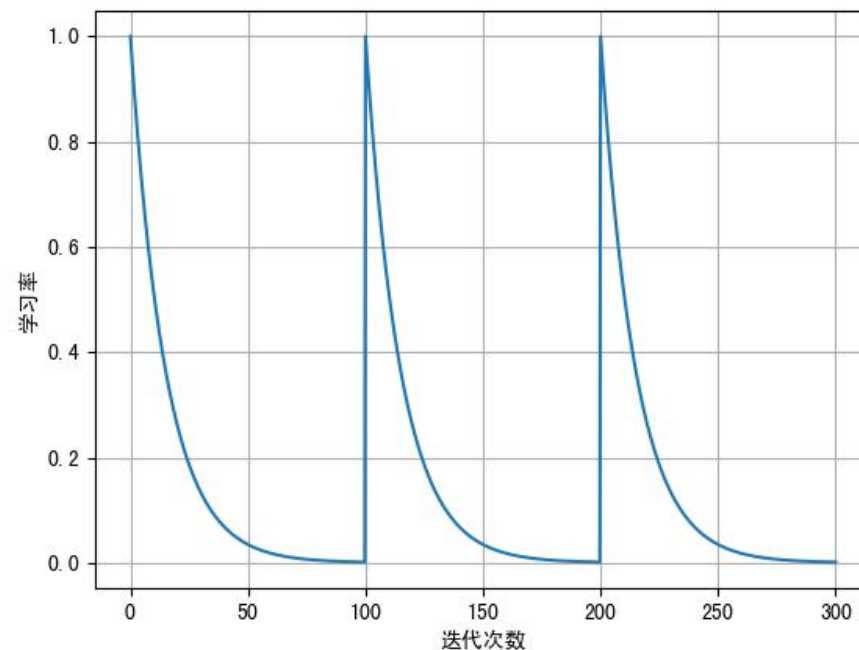


# 周期性学习率调整：SGDR

**带热重启的随机梯度下降(SGDR)**：学习率会每经过一定次数的迭代后重新初始化为预先设定好的值，并继续衰减。

$T_0$  代表每间隔  $T_0$  次迭代学习率重启一次，  
每次重启后的学习率回到  $\eta_0$ 。

比如右图设置  $T_0=100$ ，  
每100轮迭代后，学习率会重启；

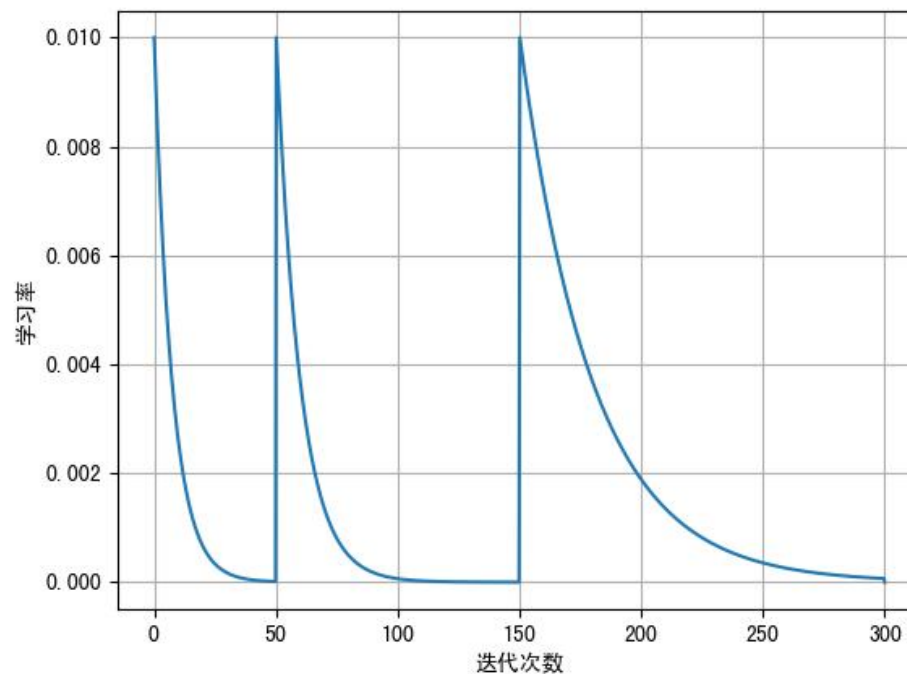


# 周期性学习率调整：SGDR

**带热重启的随机梯度下降(SGDR)**：学习率会每经过一定次数的迭代后重新初始化为预先设定好的值，并继续衰减。

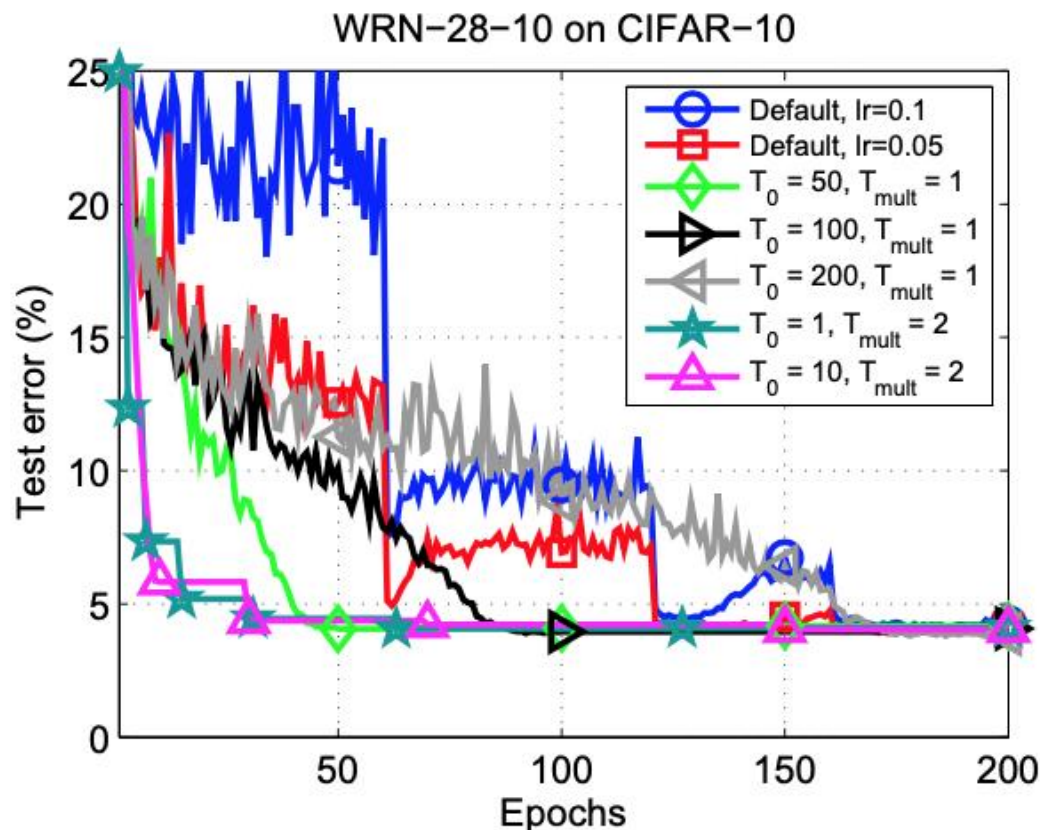
$T_{mult}$  是周期扩大因子，默认为1。如果不为1，则每次热重启周期都会扩大为前一次的  $T_{mult}$  倍，使得学习率衰减越来越缓慢。

比如右图设置  $T_{mult}=2$ ，  
每个循环的迭代次数都是上一个循环的 2 倍；



# 周期性学习率调整：SGDR

**例：**对于CIFAR10数据集，使用了SGDR方法比学习率衰减方法(Default)迭代次数减少了2~4倍，并能达到相似的或者更好的性能。



# 自适应学习率调整

- 如果参数过多，对每个参数设置不同的学习率未免太过繁琐且不切实际，但相同学习率又不一定适合所有参数，所以可以通过参数空间的不同而自适应的调整学习率。
- 经典的自适应学习率调整方法：
  - **自适应梯度** (Adaptive Gradient, AdaGrad)
  - **均方根传递** (Root Mean Square Propagation, RMSProp)
  - **自适应矩估计** (Adaptive Momentum Estimation, Adam)

# 自适应学习率调整：AdaGrad

## ● AdaGrad算法：

在训练过程中动态调整学习率，对不同参数根据**累计梯度平方**和更新不同学习率。

### AdaGrad 算法：

要求：全局学习率  $\varepsilon$

要求：初始参数  $\theta$

要求：小常数  $\delta$ ，通常设为  $10^{-7}$ （用于被小数除时的数值稳定）

初始化累积变量  $\gamma = 0$

如果满足条件停止，不满足继续下一步

第一步，从训练集中选取全部样本  $\{x_1, x_2, \dots, x_m\}$ ，对应目标为  $\{y_1, y_2, \dots, y_m\}$ 。

第二步，计算当前梯度：
$$g_t \leftarrow \frac{1}{m} \nabla_{\theta_t} \sum_i J(h_{\theta_t}(x_i), y_i)$$

第三步，计算累积平方梯度：
$$\gamma_t \leftarrow \gamma_{t-1} + g_t \odot g_t$$

第四步，计算参数更新：
$$\Delta \theta_t = -\frac{\varepsilon}{\delta + \sqrt{\gamma_t}} \odot g_t$$

第五步，应用参数更新：
$$\theta_t \leftarrow \theta_{t-1} + \Delta \theta_t$$

结束

# 自适应学习率调整：AdaGrad

- 因为  $\gamma$  是关于梯度平方和的累加项，所以：

$$\gamma \leftarrow \gamma + g \odot g, \quad \Delta \theta \leftarrow \frac{-\eta}{\delta + \sqrt{\gamma}} \odot g$$

- 梯度变化频率高、幅度大的参数，学习率下降较快，即高频特征使用较小学习率。
  - 梯度变化频率低、幅度小的参数，学习率下降较慢，即低频特征使用较大学习率。
  - 因为累加性，学习率的趋势是不断衰减的，这也符合迭代后期靠近极值点时需设置较小的学习率的直观想法。
- 
- **缺点：** 由于学习率的不断衰减，在迭代过程早期衰减过快可能直接导致后期收敛动力不足，使得 AdaGrad 无法获得满意的结果。



# 自适应学习率调整：RMSProp

## ● RMSProp 算法：

通过指数加权移动平均替代  
累计平方梯度和，可以在有  
些情况下避免 AdaGrad 算法  
中学习率不断单调下降以至  
于过早衰减的缺点。

### RMSProp 算法过程：

要求：全局学习率  $\varepsilon$ ，衰减速率  $\rho$

要求：初始参数  $\theta$

要求：小常数  $\delta$ ，通常设为  $10^{-6}$ （用于被小数除时的数值稳定）

初始化累积变量  $\gamma = 0$

如果满足条件停止，不满足继续下一步

第一步，从训练集中选取全部样本  $\{x_1, x_2, \dots, x_m\}$ ，对应目标为  $\{y_1, y_2, \dots, y_m\}$ 。

第二步，计算当前梯度：
$$g_t \leftarrow \frac{1}{m} \nabla_{\theta_{t-1}} \sum_i J(h_{\theta_{t-1}}(x_i), y_i)$$

第三步，计算累积平方梯度：
$$\gamma_t \leftarrow \rho \gamma_{t-1} + (1 - \rho) g_t \odot g_t$$

第四步，计算参数更新：
$$\Delta \theta_t = -\frac{\varepsilon}{\sqrt{\gamma_t} + \delta} \odot g_t$$

第五步，应用参数更新：
$$\theta_t \leftarrow \theta_{t-1} + \Delta \theta_t$$

结束



# 自适应学习率调整：RMSProp

- RMSProp 算法添加衰减因子  $\rho$ ，在学习率更新过程中权衡过去与当前的梯度信息，减轻了因梯度不断累计导致学习率大幅降低的影响，防止学习过早结束。

$$\gamma \leftarrow \rho\gamma + (1 - \rho)g \odot g$$

$$\Delta\theta \leftarrow \frac{-\eta}{\delta + \sqrt{\gamma}} \odot g$$

# 自适应学习率调整：Adam

## ● Adam 算法：

Adam融合了动量法和 RMSProp的思想，使用动量作为参数更新的方向，并引入了偏差修正机制，在不同的参数更新阶段自适应地调整学习率。

---

### Algorithm 5 Adam 算法

---

**Require:** 全局学习率  $\varepsilon$ , 动量参数  $\beta_1, \beta_2$

初始化参数  $\theta$

小常数  $\delta$ , 通常设为  $10^{-8}$  (用于被小数除时的数值稳定)

初始化一阶动量  $m_0 = 0$ , 二阶动量  $v_0 = 0$

初始化时间步  $t = 0$

**Ensure:** 更新后的模型参数  $\theta_t$

- 1: **while** 误差大于设定值或未达到最大迭代次数 **do**
- 2: 从训练集中选取全部样本  $\{x_1, x_2, \dots, x_m\}$ , 对应目标为  $\{y_1, y_2, \dots, y_m\}$ 。
- 3: 计算当前梯度:  $g_t \leftarrow \frac{1}{m} \nabla_{\theta_1} \sum_i L(f_{\theta_i}(x_i), y_i)$
- 4: 更新一阶动量:  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
- 5: 更新二阶动量:  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t$
- 6: 计算偏差修正后的一阶动量:  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
- 7: 计算偏差修正后的二阶动量:  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
- 8: 计算参数更新:  $\Delta\theta_t = -\frac{\varepsilon \hat{m}_t}{\sqrt{\hat{v}_t} + \delta}$
- 9: 应用参数更新:  $\theta_t \leftarrow \theta_{t-1} + \Delta\theta_t$

10: **end while**

---

# 自适应学习率调整: Adam

- Adam (Adaptive Moment Estimation, 自适应矩估计) 是动量法和自适应学习率的结合, 使用向量 $\mathbf{m}$ 和 $\mathbf{v}$ 分别存储一阶动量与二阶动量

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \mathbf{g}_t$$

$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\delta + \sqrt{\mathbf{v}_{t+1}}} \odot \mathbf{m}_{t+1}$$

因为 $\mathbf{m}$ 和 $\mathbf{v}$ 初始化为0, 初始阶段的估计值存在偏差, 因此需要对偏差进行校正(debiasing)

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t}, \quad \widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t}$$

# 自适应学习率调整：Adam

- Adam是机器学习实践中最受欢迎的优化器之一，它有多个衍生算法：

Nadam = Adam + Nesterov

$$\mathbf{g}_t = \nabla f(\boldsymbol{\theta}_t - \frac{\eta}{\delta + \sqrt{\mathbf{v}_t}} \odot \mathbf{m}_t)$$

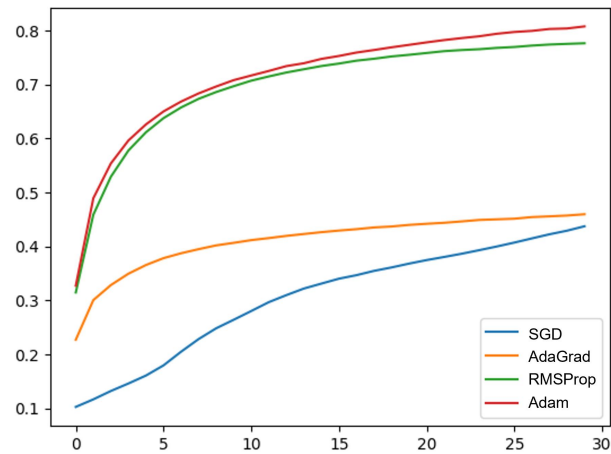
AdamW = Adam + Weight Decay

Rectified Adam (RAdam)

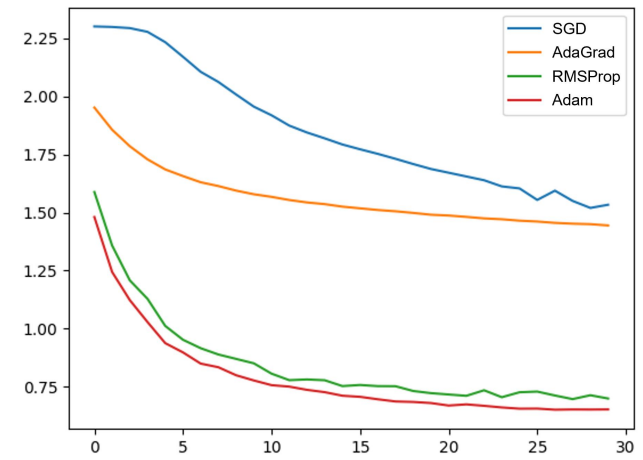
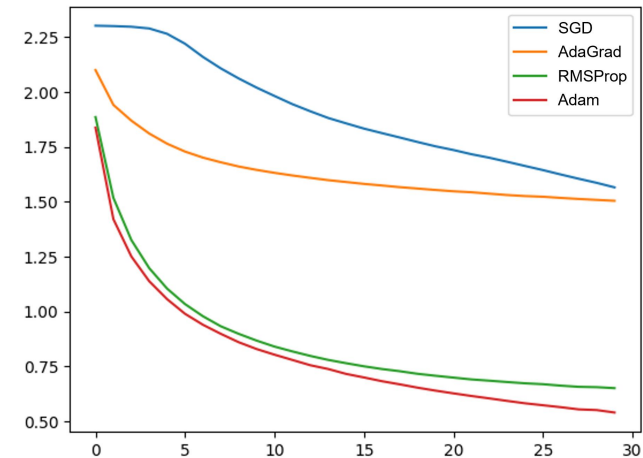
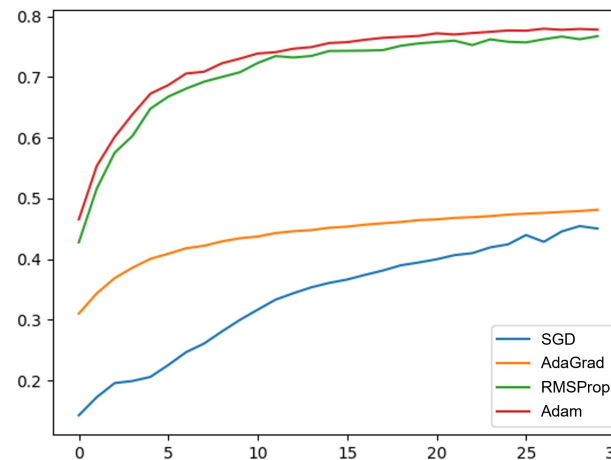
# 自适应学习率调整

**例：**三种自适应学习率调整方法在CIFAR10数据集上的实验效果：

训练集准确率  
与损失值



验证集准确率  
与损失值



02

# 损失函数

# 损失函数

- 上一讲，我们提到网络预测值与实际值的差值称为“误差”：

$$e = Y - O$$

- 本讲将详细介绍衡量网络预测值与实际值间差值的函数——损失函数  $F(e)$ 。

# 损失函数

机器学习问题真正关注的是测试时的某些性能度量 $P$ ，但只能被间接优化。

损失函数是度量 $P$ 在训练过程中的代理，用来指导模型的训练并度量模型的好坏。

通常模型对数据拟合的越好，损失函数值越小，反之亦然。

我们希望通过降低损失函数 $J(\theta)$ 来间接提高 $P$ 。神经网络的训练过程本质上就是**不断优化损失函数的过程**。



## 2.1 回归损失函数

# 回归损失函数

## 常用的损失函数：

1. 平方误差损失与绝对值误差损失
2. Huber损失

# 平方损失与绝对值损失

平方误差损失(MSE) :

$$L(Y, f(X)) = (f(X) - Y)^2$$

绝对值误差损失(MAE):

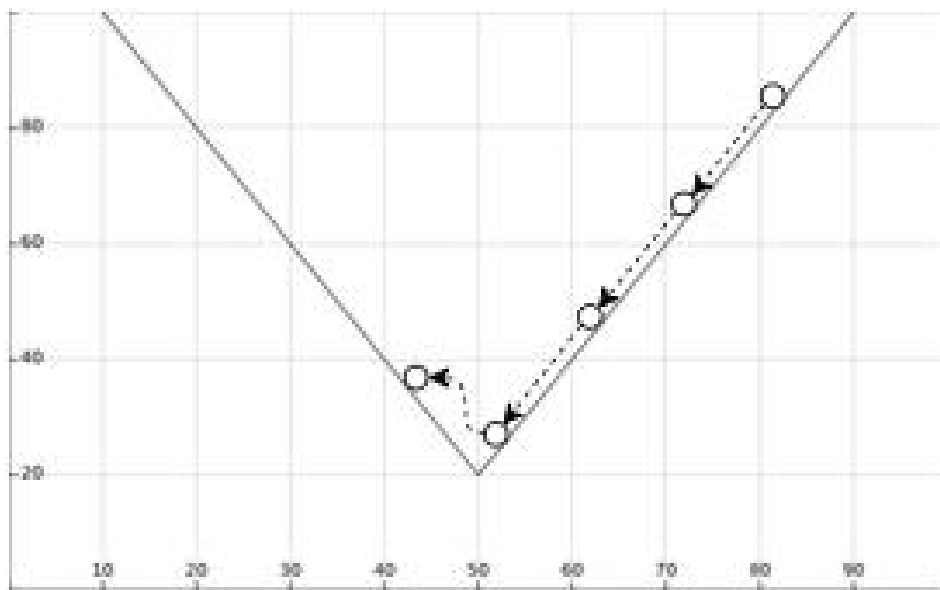
$$L(Y, f(X)) = |f(X) - Y|$$

其中  $f(X)$  是预测值,  $Y$  是实际值。

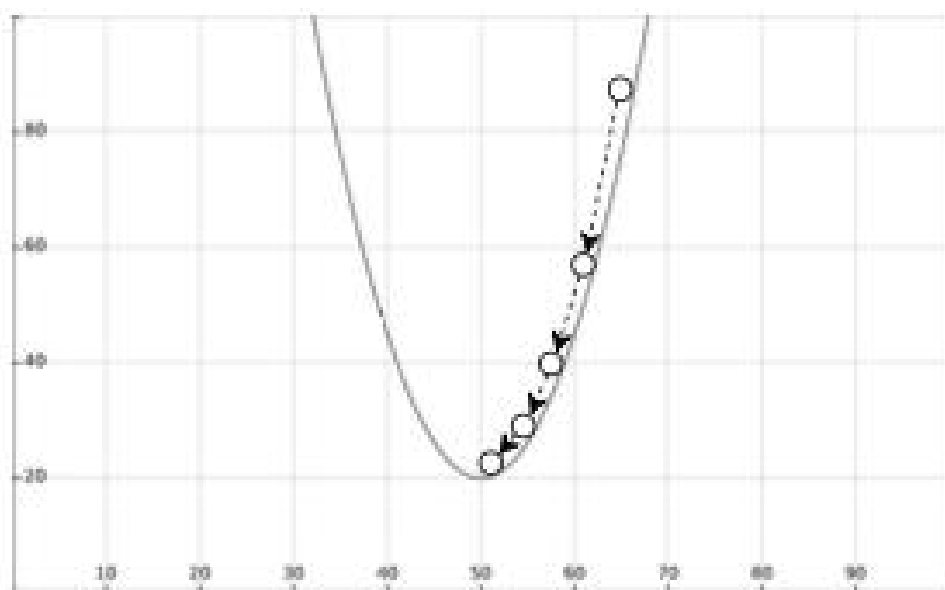
# 平方损失与绝对值损失

- **MAE 对于异常点更加鲁棒**：在有异常点的情况下，平方误差会扩大误差，因此平方误差损失函数对异常点更敏感。
- **MSE 通常比 MAE 可以更快地收敛**：绝对值误差损失函数更新梯度始终一样，不利于模型收敛。

绝对值误差损失



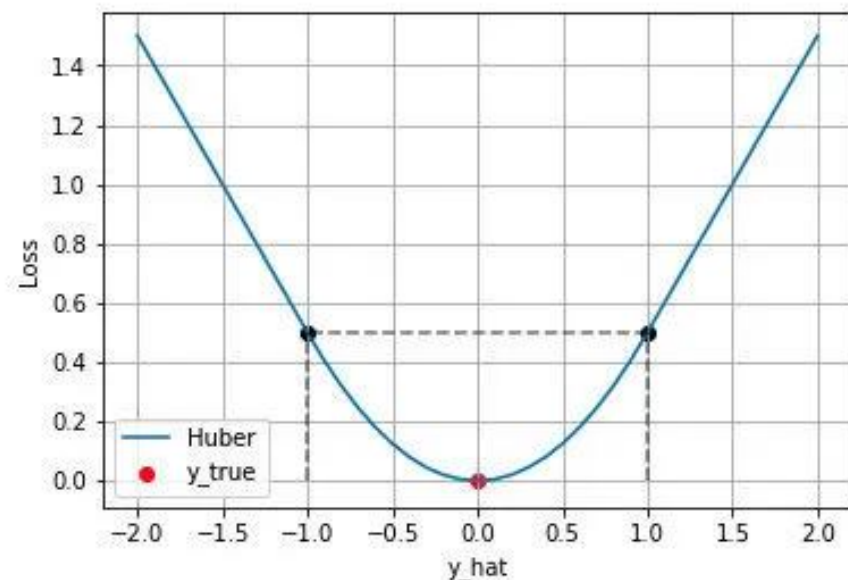
平方误差损失



# Huber损失

结合了平方误差损失和绝对值误差损失优点:

- 误差较小时: 平方误差损失
- 误差较大时: 绝对值误差损失



$$L(Y, f(X)) = \begin{cases} \frac{1}{2} (f(X) - Y)^2. & \text{if } |f(X) - Y| \leq \delta \\ \delta |f(X) - Y| - \frac{1}{2} \delta^2. & \text{else} \end{cases}$$

其中  $\delta$  为超参数, 是需要人工确定, 这也是 Huber 损失的缺点。

## 2.2 分类损失函数

# 分类损失函数

## 常用的损失函数：交叉熵损失

- 在神经网络中，通常在网络的最后一层经过 *softmax* 函数得到每个类别的预测概率，通过交叉熵损失函数公式计算交叉熵损失
- 交叉熵损失越小表示预测概率分布越接近真实概率分布
- *softmax* 函数：

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}, \quad i = 1, 2, \dots, K$$

# 交叉熵损失

假设样本数据总共有  $K$  个类别，则网络的输出为  $[\hat{y}_1, \dots, \hat{y}_n]$ ，*label* 采用 one-hot 形式表示，即 *label* 为  $K$  维向量，其中目标类别为  $i$ ，则第  $i$  维的值为 1，其余维度值为 0，由此我们计算得到一个样本的交叉熵损失为：

$$loss = - (y_1 \log \hat{y}_1 + \dots + y_K \log \hat{y}_K) = - y_i \log \hat{y}_i = - \log \hat{y}_i$$

样本数为  $N$ ，类别数为  $K$  的数据，总的交叉熵损失为：

$$L = - \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^K y_{ij} \log \hat{y}_{ij}$$



# 分类损失函数

**例1：**当前有三种可预测类别：猫、狗、猪。

假设我们有两个模型（参数不同），可以通过定义损失函数来判断模型在样本上的表现。

模型一		
预测	真实	是否正确
0.3 0.3 0.4	0 0 1 (猪)	正确
0.3 0.4 0.3	0 1 0 (狗)	正确
0.1 0.2 0.7	1 0 0 (猫)	错误
模型二		
预测	真实	是否正确
0.1 0.2 0.7	0 0 1 (猪)	正确
0.1 0.7 0.2	0 1 0 (狗)	正确
0.3 0.4 0.3	1 0 0 (猫)	错误

# 分类损失函数

- 分类错误率

损失函数定义为:  $\text{classification error} = \frac{\text{count of error items}}{\text{count of all items}}$

模型1:  $\text{classification error} = \frac{1}{3}$

模型2:  $\text{classification error} = \frac{1}{3}$

虽然模型1和模型2都预测错误了一个，但是模型2实际上效果要好于模型1，只是通过这样的损失函数不能很好的反应当前模型的效果。

# 分类损失函数

## • 均方误差

损失函数定义为： $MSE = \frac{1}{n} \sum_i^n (\hat{y}_i - y_i)^2$

模型1：  
sample 1 loss =  $(0.3 - 0)^2 + (0.3 - 0)^2 + (0.4 - 1)^2 = 0.54$   
sample 2 loss =  $(0.3 - 0)^2 + (0.4 - 1)^2 + (0.3 - 0)^2 = 0.54$   
sample 3 loss =  $(0.1 - 1)^2 + (0.2 - 0)^2 + (0.7 - 0)^2 = 1.34$

$$MSE = \frac{0.54 + 0.54 + 1.34}{3} = 0.81$$

模型2：  
sample 1 loss =  $(0.1 - 0)^2 + (0.2 - 0)^2 + (0.7 - 1)^2 = 0.14$   
sample 2 loss =  $(0.1 - 0)^2 + (0.7 - 1)^2 + (0.2 - 0)^2 = 0.14$   
sample 3 loss =  $(0.3 - 1)^2 + (0.4 - 0)^2 + (0.3 - 0)^2 = 0.74$

$$MSE = \frac{0.14 + 0.14 + 0.74}{3} = 0.34$$

通过MSE可以判断当前模型2的参数优于模型1。

# 分类损失函数

- 交叉熵

损失函数定义为：

$$loss = -(y_1 \log \hat{y}_1 + \dots + y_K \log \hat{y}_K)$$

$$L = \frac{1}{N} \sum_i^N loss_i$$

模型1：

$$\begin{aligned} \text{sample 1 loss} &= -(0 \times \log 0.3 + 0 \times \log 0.3 + 1 \times \log 0.4) = 0.91 \\ \text{sample 2 loss} &= -(0 \times \log 0.3 + 1 \times \log 0.4 + 0 \times \log 0.3) = 0.91 \\ \text{sample 3 loss} &= -(1 \times \log 0.1 + 0 \times \log 0.2 + 0 \times \log 0.7) = 2.30 \end{aligned}$$

$$L = \frac{0.91 + 0.91 + 2.3}{3} = 1.37$$

模型2：

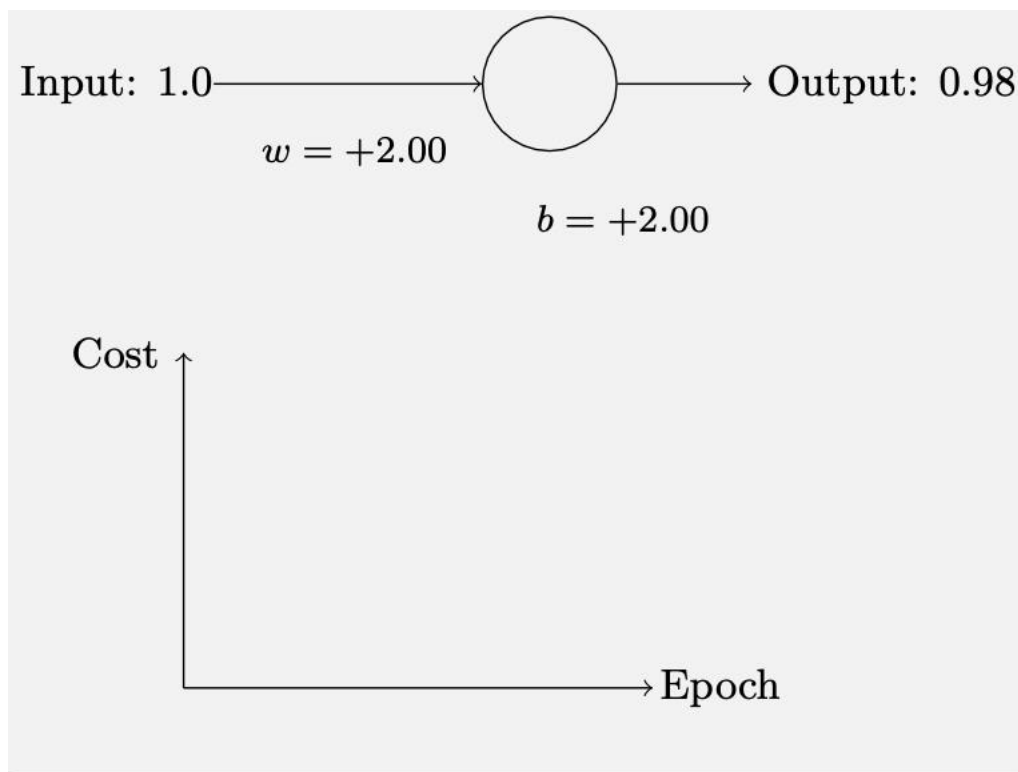
$$\begin{aligned} \text{sample 1 loss} &= -(0 \times \log 0.1 + 0 \times \log 0.2 + 1 \times \log 0.7) = 0.35 \\ \text{sample 2 loss} &= -(0 \times \log 0.1 + 1 \times \log 0.7 + 0 \times \log 0.2) = 0.35 \\ \text{sample 3 loss} &= -(1 \times \log 0.3 + 0 \times \log 0.4 + 0 \times \log 0.4) = 1.20 \end{aligned}$$

$$L = \frac{0.35 + 0.35 + 1.2}{3} = 0.63$$

交叉熵损失函数也可以捕捉到模型1和模型2预测效果的差异。

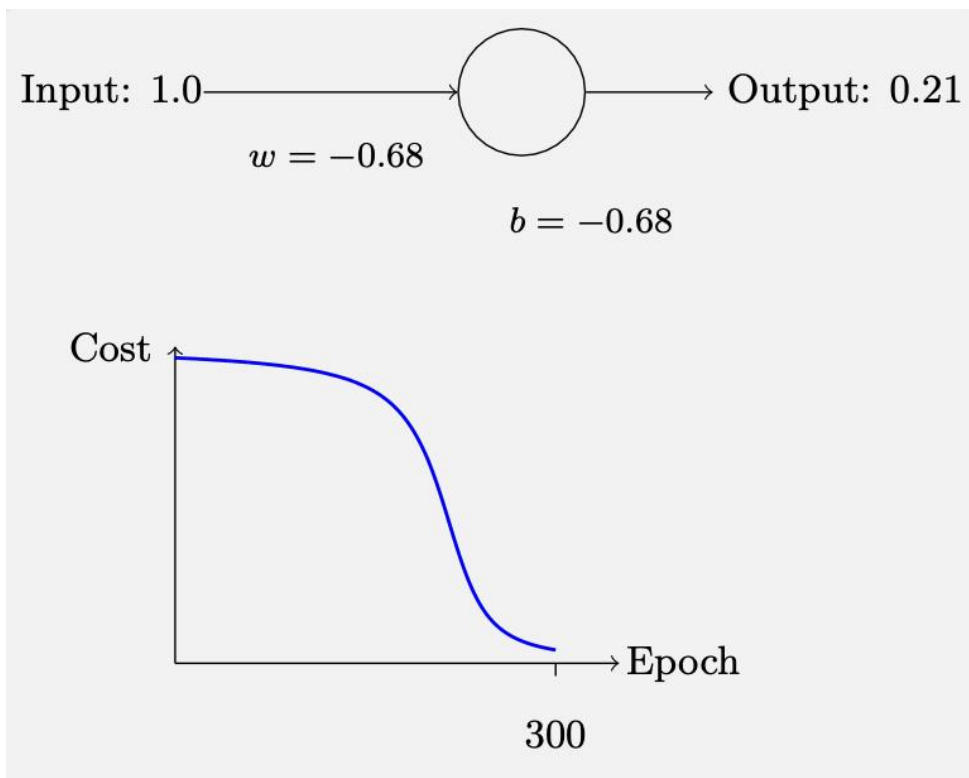
# 分类损失函数

**例2：**假设有一个单输入单输出的神经元，输入为1，目标输出为0。将初始权重和偏置都设置为 2.0，此时初始输出为 0.98。

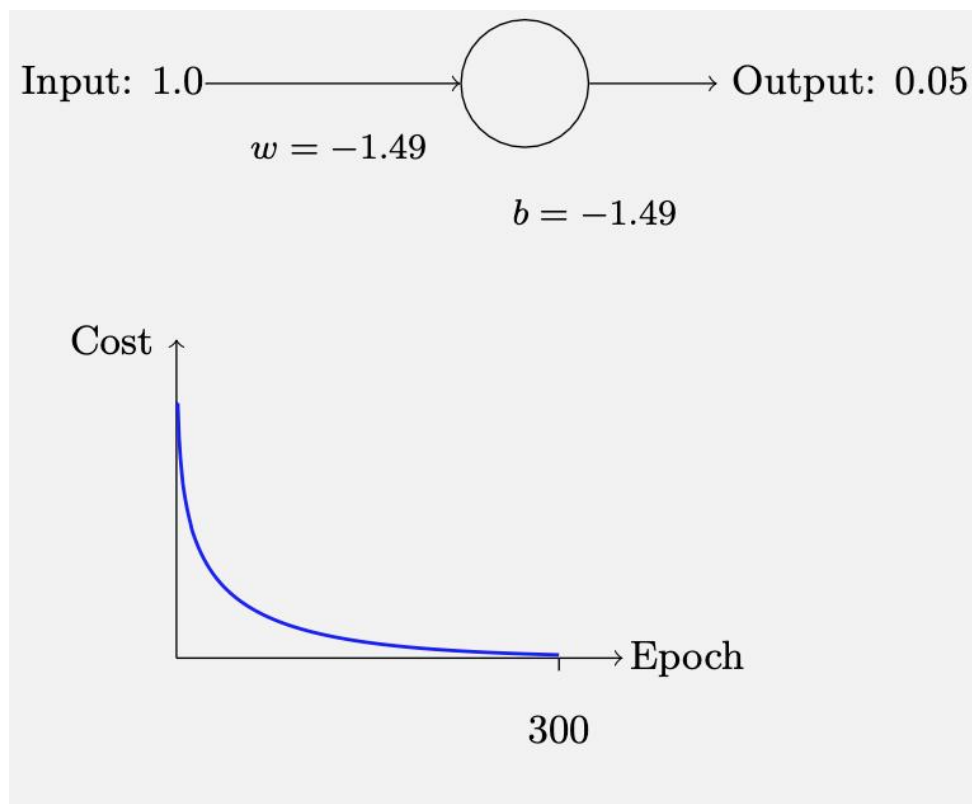


# 分类损失函数

## 均方误差的训练过程



## 交叉熵的训练过程



神经元权重和偏置的学习速度基本上是由损失函数的偏导数  $\frac{\partial C}{\partial w}$  和  $\frac{\partial C}{\partial b}$  决定的。

所以如果“学习缓慢”，可能是由这些偏导数很小所导致的。

# 分类损失函数

均方误差损失函数为：
$$C = \frac{(y - a)^2}{2}$$

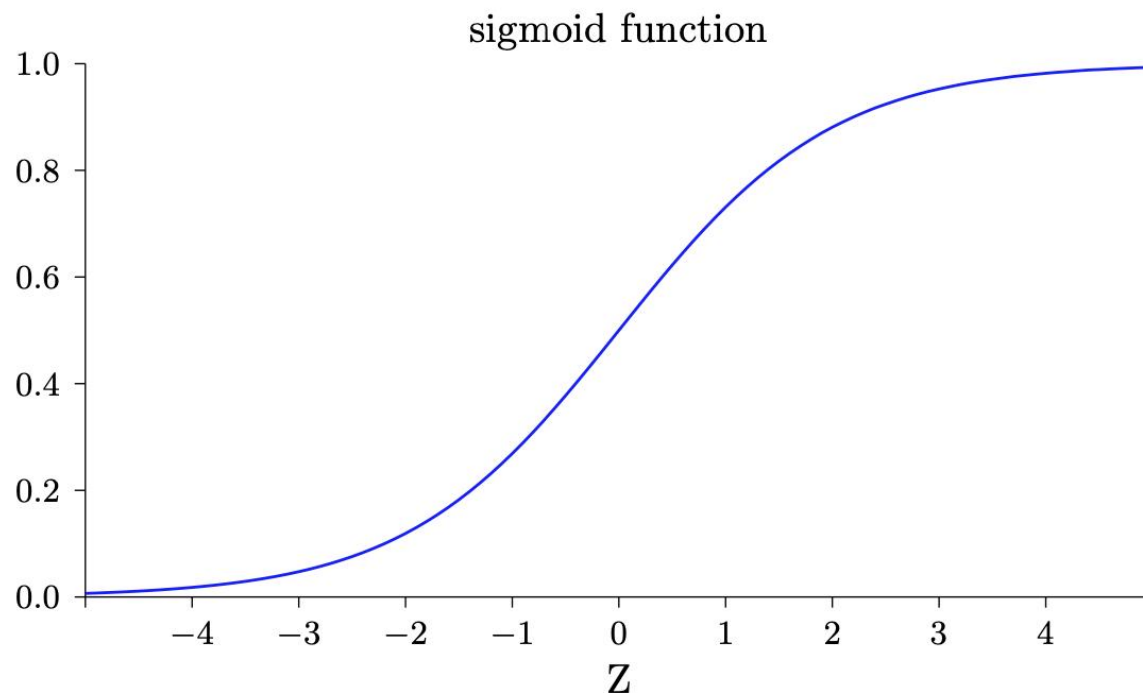
其中  $a$  是神经元的输出， $x = 1$  是训练输入， $y = 0$  则是目标输出。

有  $a = \sigma(z)$ ，其中  $z = wx + b$ ，使用链式法则来求权重和偏置的偏导数，并将  $x = 1$  和  $y = 0$  代入有：

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z)$$

当  $a \rightarrow 1$  时， $\sigma'(z)$  就很小了



# 分类损失函数

交叉熵损失函数为：
$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \\ &= -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j \\ &= \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y) \\ &= \frac{1}{n} \sum_x x_j (\sigma(z) - y) \end{aligned}$$

因为  $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ ,

这里，权重学习的速度受到  $\sigma(z) - y$ ，也就是输出值和目标值的误差的控制。更大的误差带来更快的学习速度，避免了均方误差“学习缓慢”的问题。



03

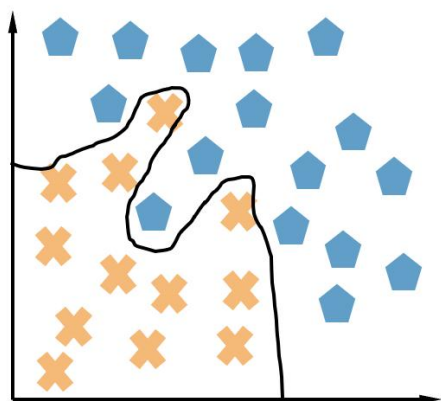
## 正则化

# 泛化能力

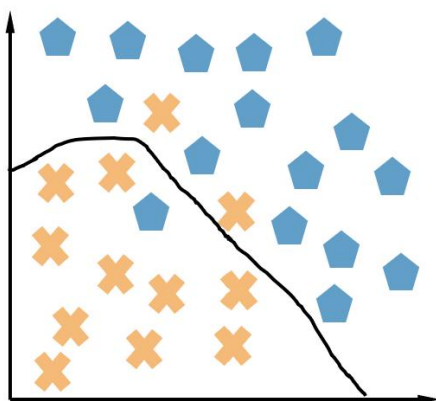
数据中总是包含随机变量或者噪声。一个模型能从数据的噪声中学习分辨出总体趋势，这种特性叫做“模型的泛化能力”。

**欠拟合：**模型比较简单，特征维度过少。欠拟合的模型不能充分拟合数据。

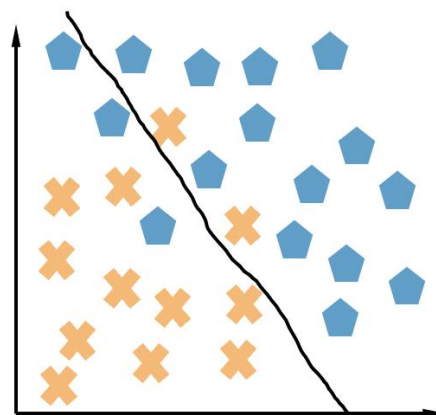
**过拟合：**模型过于复杂，参数过多或训练数据过少，噪声过多。过拟合的模型会拟合噪声弱化泛化能力。



(a)过拟合



(b)好拟合

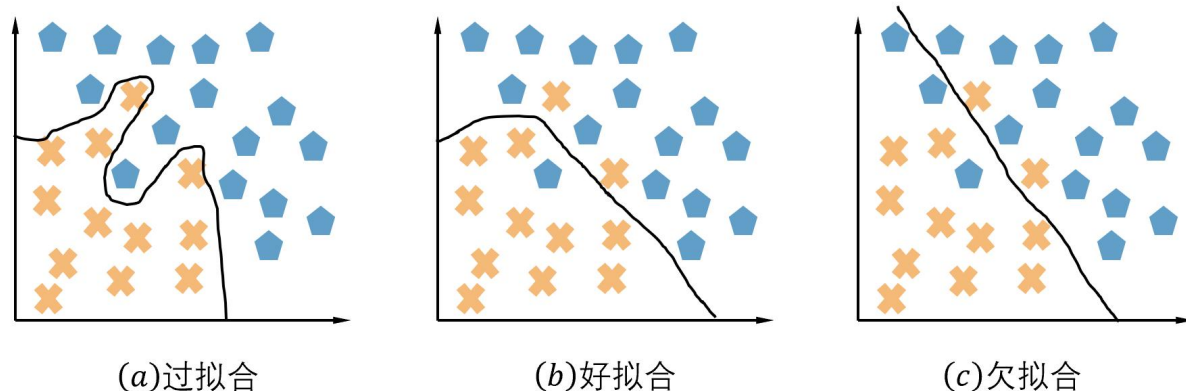


(c)欠拟合

# 泛化能力

**偏差：** 衡量模型预测值和实际值之间的偏离关系，即模型在样本上拟合得好不好。

**方差：** 描述模型在整体数据上表现的稳定情况，在训练集和验证集/测试集上表现是否一致。

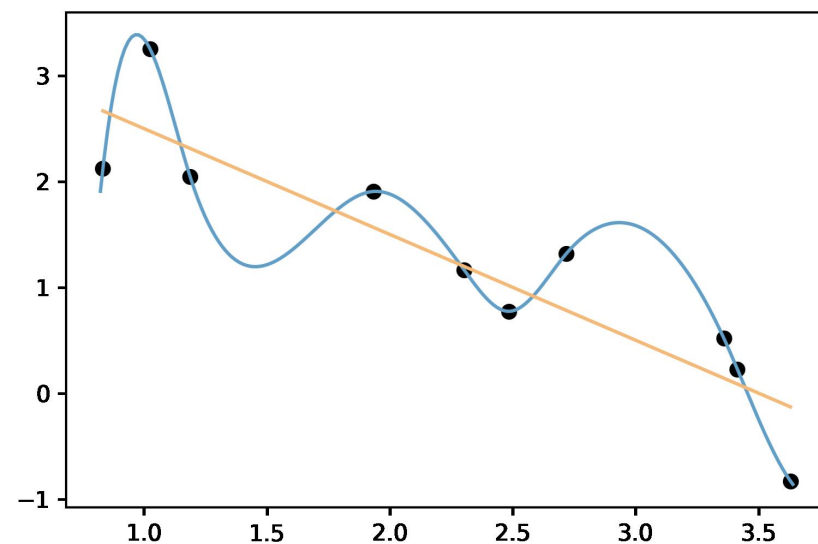


情况	1	2	3	4
训练集错误率	16%	13%	3%	1%
验证集错误率	35%	17%	13%	2%
情况 1：高偏差，高方差（差模型）				
情况 2：高偏差，低方差（欠拟合）				
情况 3：低偏差，高方差（过拟合）				
情况 4：低偏差，低方差（好模型）				

# 泛化能力

**思考题：** 过拟合的时候，拟合函数的系数（ $W$ ）往往非常大，为什么？

通过减小权重 $w$ 的值来降低网络的复杂度。  
以此缓解过拟合现象。



# 泛化能力

神经网络的灵活性来自于隐藏神经元，随着隐藏神经元数量的增多，网络的权值也变化增多了，网络灵活性变强。

**为了避免过拟合，必须减小神经网络的灵活性。**

**为了避免欠拟合，又必须有足够多的神经元。**

HOW?

# 正则化

使用正则化方法来降低模型的复杂程度，从而解决过拟合问题，提升泛化能力。

**正则化**(Regularization)字面本身的意思是使得事情变得规则，也就是**规则化**，一般表示一类通过限制模型复杂度，从而避免过拟合，提高泛化能力的方法。

只要能限制模型复杂度，避免过拟合的方法，我们都可以归为正则化方法。

# 正则化

## 常见正则化方法：

1. L1和L2正则化
2. 丢弃法
3. 提前停止

# L1和L2正则化

●L<sub>1</sub>范数:

$$||\mathbf{x}||_1 = \sum_i |x_i|$$

●L<sub>2</sub>范数:

$$||\mathbf{x}||_2 = \left( \sum_i |x_i|^2 \right)^{1/2}$$

优化目标:

$$\arg \min_{\theta} \frac{1}{N} \left( \sum_i^N L(y_i, f(x_i, \theta)) \right)$$

其中 $\mathcal{L}(\cdot)$ 为损失函数,  $f(\cdot)$ 为待学习的神经网络,  $\theta$ 为参数,  $N$ 为样本数量。



# L1和L2正则化

- 加入**L<sub>1</sub>正则化**后的优化目标， $\lambda$ 为正则化系数：

$$\arg \min_{\theta} \frac{1}{N} \left( \sum_i^N L(y_i, f(x_i, \theta)) + \lambda \|\theta\| \right)$$

每次更新 $\theta$ ，假设学习率为 $\alpha$

$$\begin{aligned} \theta &:= \theta - \alpha d\theta \\ &= \theta - \frac{\alpha}{N} \left( \frac{\partial L}{\partial \theta} + \lambda \text{Sgn}(\theta) \right) \\ &= \theta - \frac{\alpha}{N} \frac{\partial L}{\partial \theta} - \frac{\alpha \lambda}{N} \text{Sgn}(\theta) \end{aligned} \quad \text{Sgn}(\theta) = \begin{cases} 1, & \theta > 0; \\ 0, & \theta = 0; \\ -1, & \theta < 0. \end{cases}$$

目的：使 $\theta$ 更容易为0，整体权重矩阵更为稀疏，抑制过拟合。

# L1和L2正则化

- 加入**L<sub>2</sub>正则化**后的优化目标， $\lambda$ 为正则化系数：

$$\arg \min_{\theta} \frac{1}{N} \left( \sum_i^N L(y_i, f(x_i, \theta)) + \lambda \theta^2 \right)$$

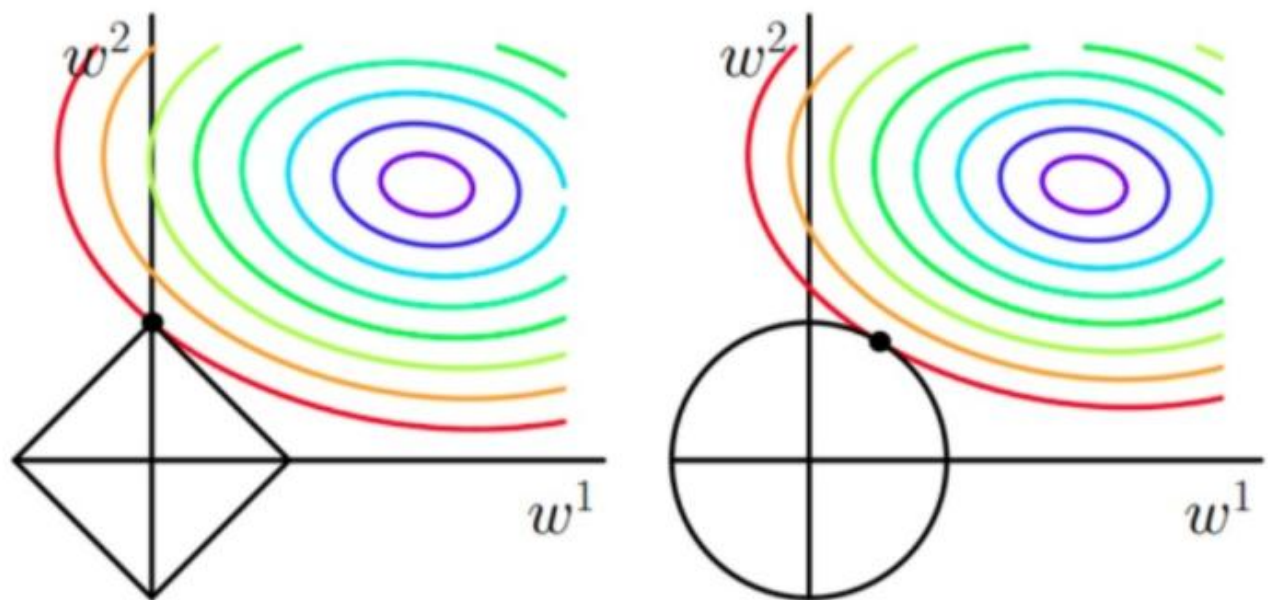
每次更新 $\theta$ ，假设学习率为 $\alpha$

$$\begin{aligned} \theta &:= \theta - \alpha d\theta \\ &= \theta - \frac{\alpha}{N} \left( \frac{\partial L}{\partial \theta} + 2\lambda\theta \right) \\ &= \left( 1 - \frac{2\alpha\lambda}{N} \right) \theta - \alpha \frac{\partial L}{N \partial \theta}. \end{aligned}$$

$0 < (1 - 2\alpha\lambda/N) < 1$   
使得权重变的更小

# L1和L2正则化

## L1和L2范数原理



- L1正则化使参数趋近坐标系（零值）。
- L2正则化使参数减小

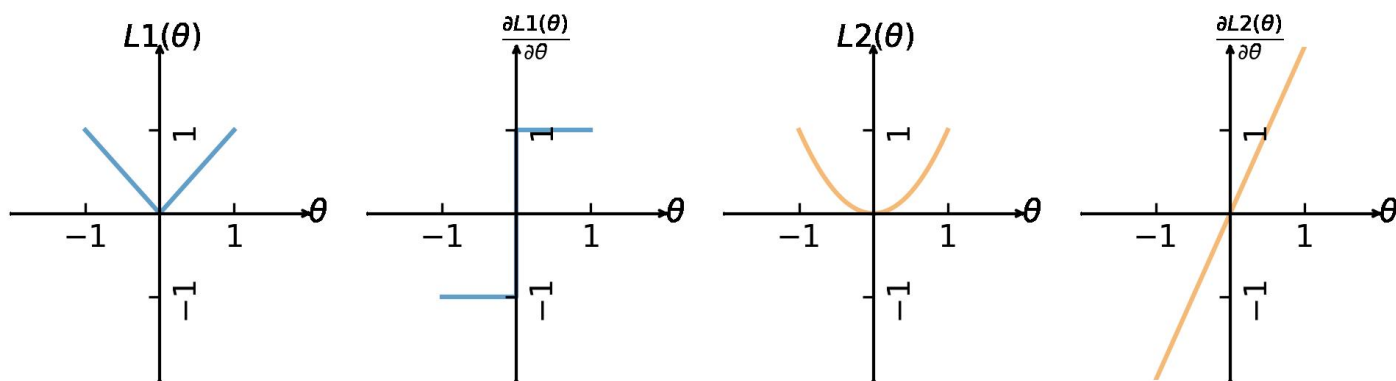
# L1和L2正则化

**例1：**  $L_1 = |\theta|$ ,  $L_2 = \theta^2$ ，分别画出函数及其导函数：

对于 $L_1$ 正则，无论 $L_1$ 大小（0除外），每次梯度更新时，梯度为1或-1，导致 $L_1$ 稳定向 0靠近；

对于 $L_2$ 正则，随着变量靠近0，其梯度逐渐减小。

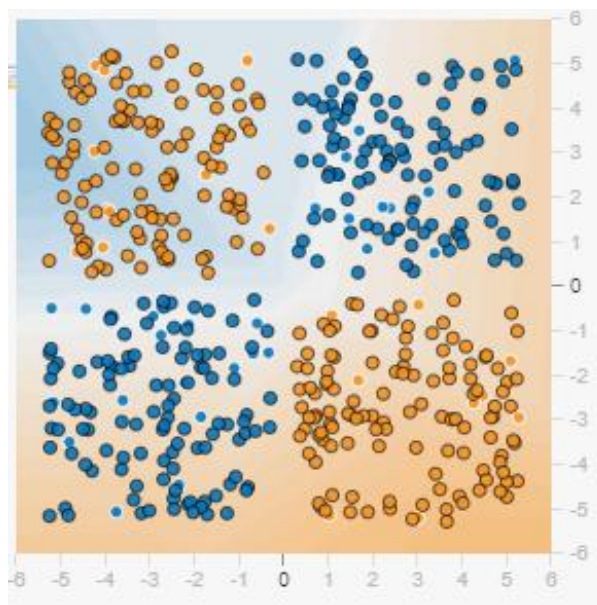
最终， $L_1$ 正则可能为0， $L_2$ 正则却几乎不可能，所以带 $L_1$ 正则项训练的模型更容易得到稀疏解。



# L1和L2正则化

**例2：**二维空间中，点被分成了四个点集，代表了两类：

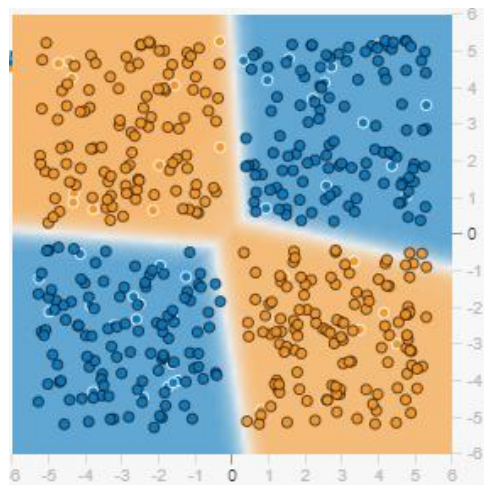
- 其中浅色小点为训练集，深色粗点为验证集，使用模型为具有一个隐含层，隐含单元为5的神经网络，激活函数为ReLU函数。
- 该任务中训练数据偏少，网络学到的分布容易偏离正确的分布。



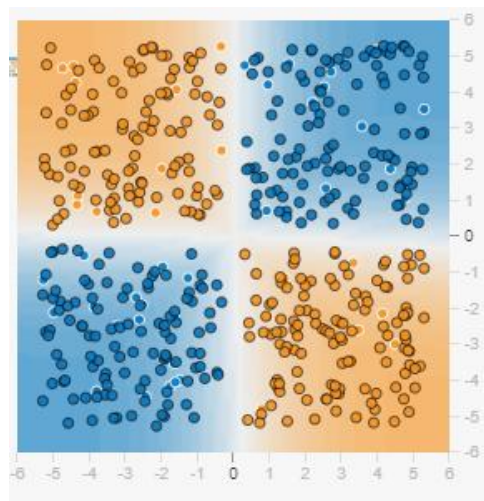


# L1和L2正则化

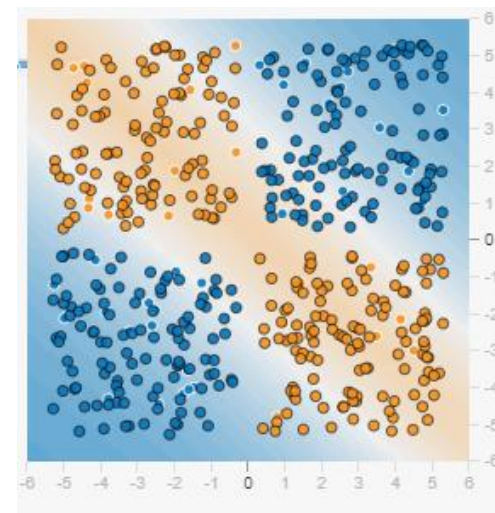
训练结果：



不采用正则化



L2正则化



L1正则化

训练后的权重（部分）：

None	-0.75	-0.85	0.87	0.34	0.75
L2	0.33	-0.25	0.33	-0.24	0.44
L1	0	0	-0.31	0.35	0

该实例可通过<https://playground.tensorflow.org/>自行验证和探讨。

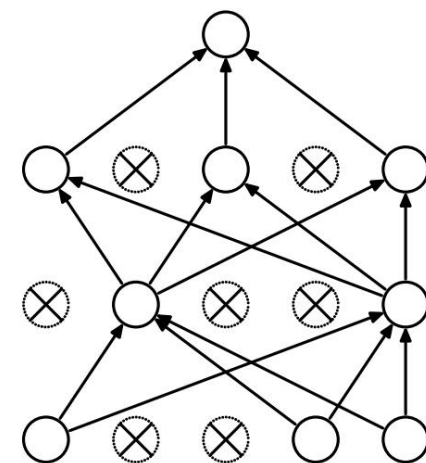
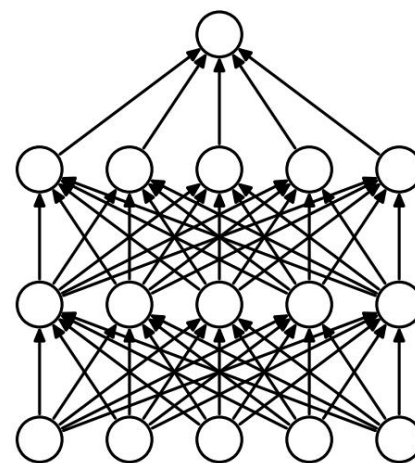
# 丢弃法 (Dropout)

- 思路：在训练时，以概率 $p$ 随机保留部分神经元。

对一神经元层 $y = f(Wx + b)$ 进行**Dropout**操作，

得到 $y = f(W \text{Dropout}(x) + b)$ ：

$$\text{Dropout}(x) = \begin{cases} m \odot x, & \text{训练时} \\ px, & \text{测试时} \end{cases}$$



$m \sim \text{Bernoulli}(p)$ 表示以概率 $p$ 的伯努利分布生成0,1向量。

# 丢弃法 (Dropout)

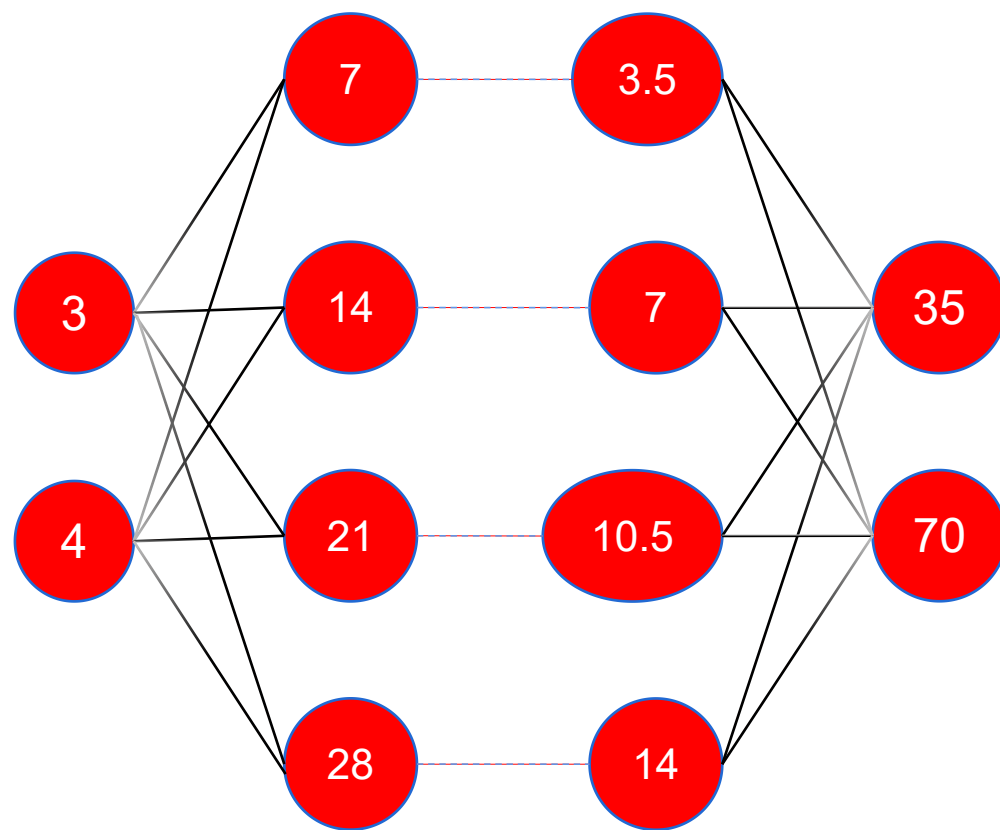
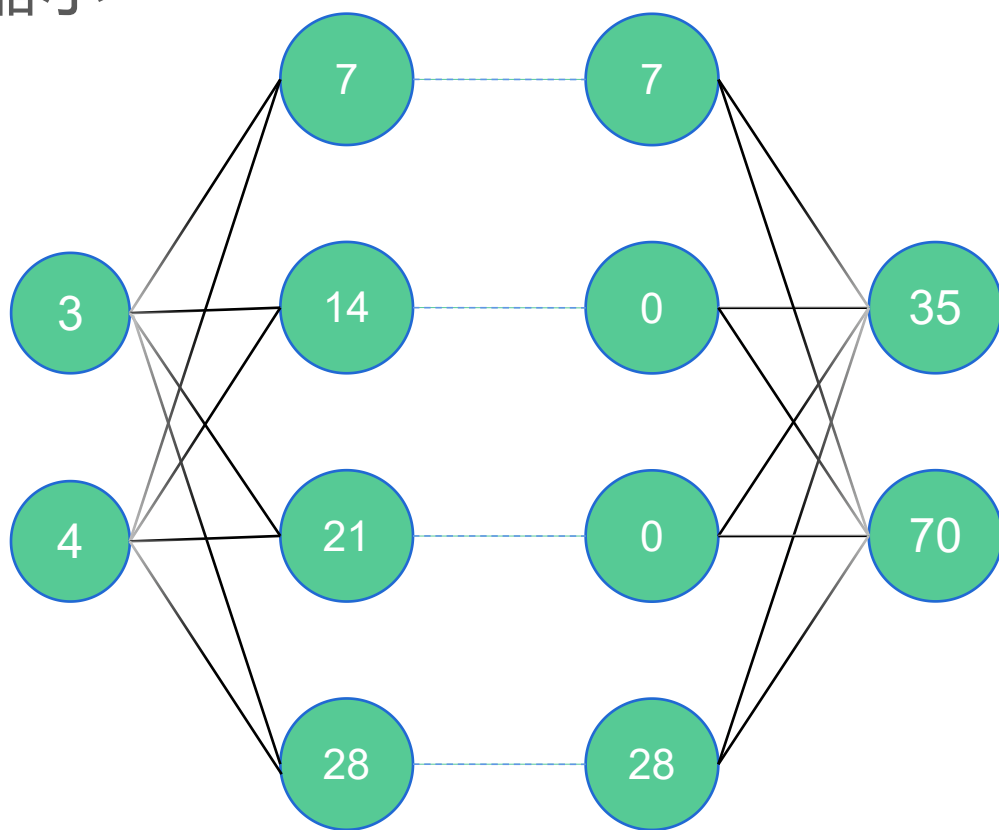
- 避免过拟合原因：

- 1.相当于取平均的作用，取每次丢弃后子网络的平均结果。
- 2.降低神经元之间的敏感度，增加整体鲁棒性。



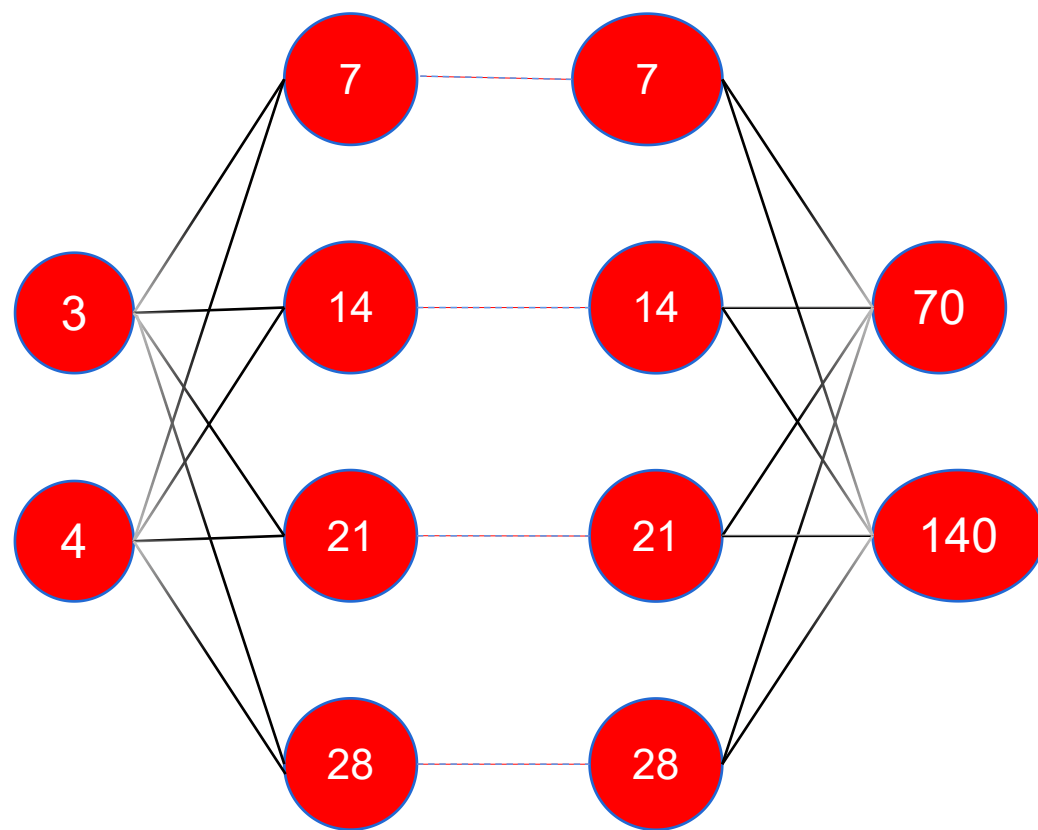
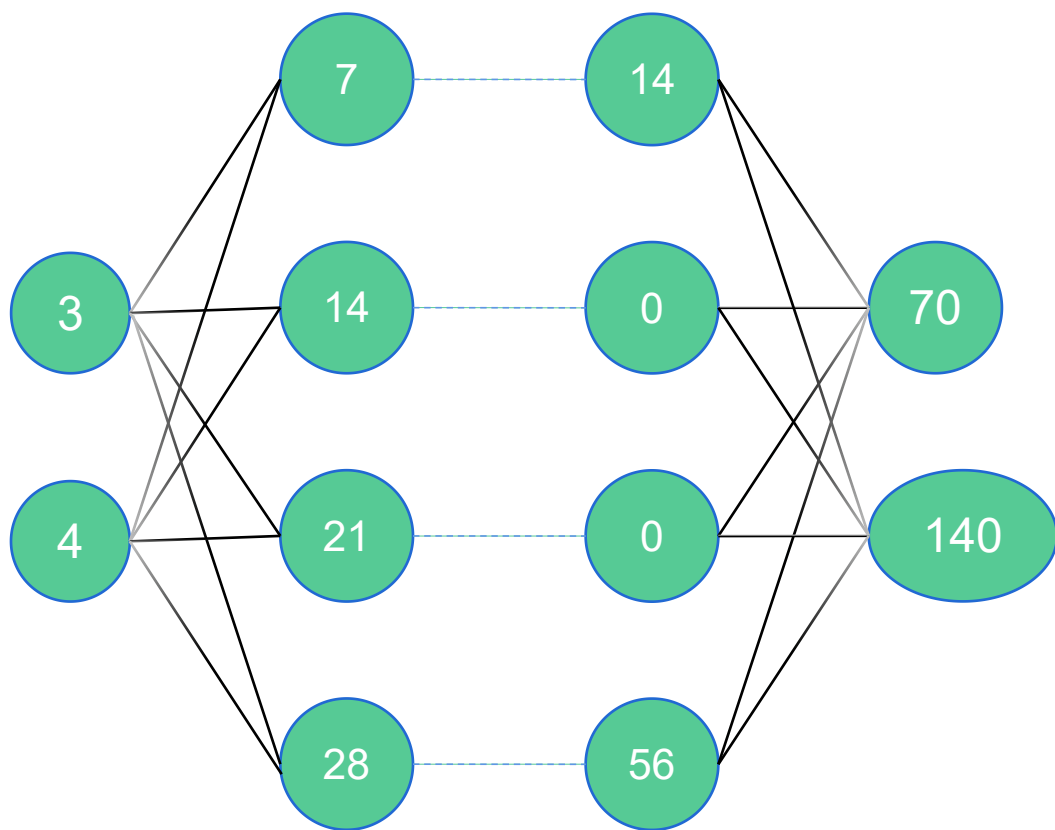
# Dropout

**例：** 设  $p = 0.5$ ，在全连接层后接dropout层，训练阶段，因为丢掉了一部分神经元，导致输出期望降低；预测阶段使用了全部神经元，也需要按比例对数值进行缩小



# Inverted Dropout

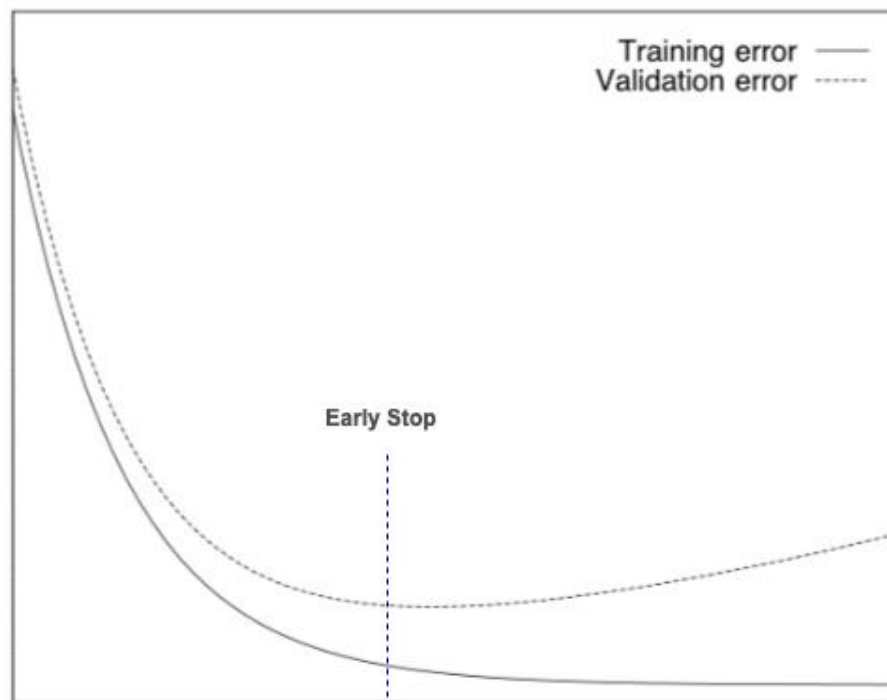
**例：** Inverted Dropout同样丢掉了一部分神经元，导致输出期望降低，但把剩余神经元的输出值按比例放大来抵消期望降低；测试阶段无需任何改动



# 提前停止

## ● 提早结束训练

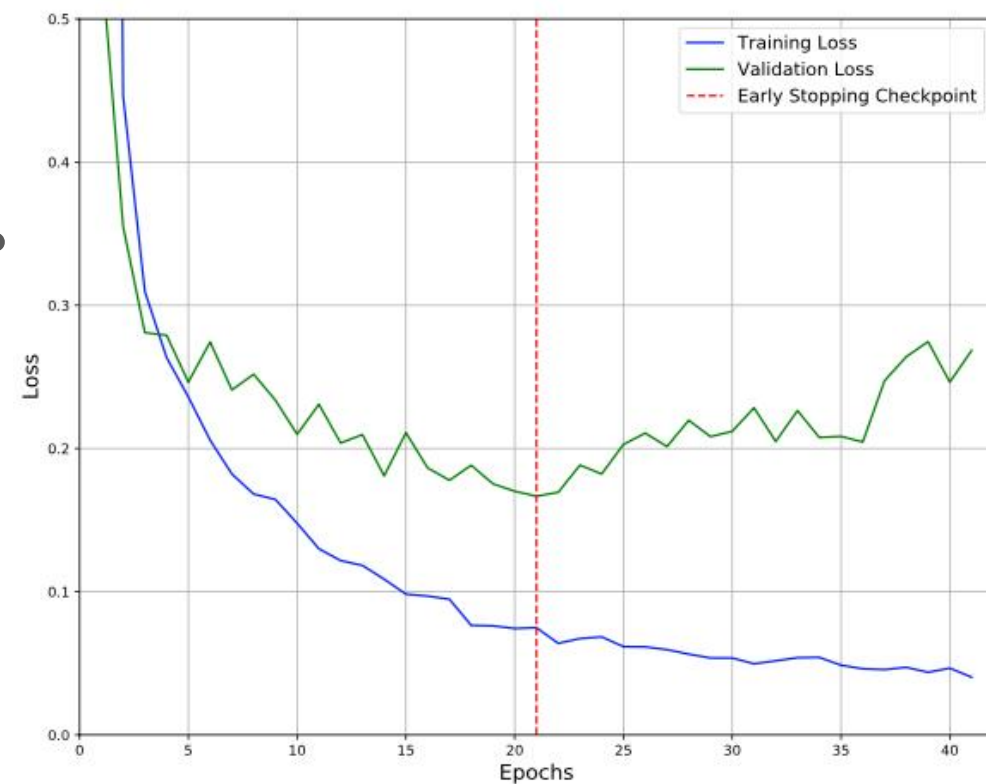
验证集错误率基本不下降时或有反增趋势时，可以提前停止训练。



# 提前停止

**例：**以一个两层全连接层，一层输出层的网络为例，进行Mnist数据集分类任务。

若设置提前停止策略的容忍度为20，验证集的损失在20轮内不再下降则停止训练。



## 04 归一化



# 归一化

## •为什么要归一化？

**实例：**从给定的房子面积 $x_1$  ( $0-1000m^2$ )和房间数量 $x_2$  ( $0-10$ )，来预测房子的总价格。

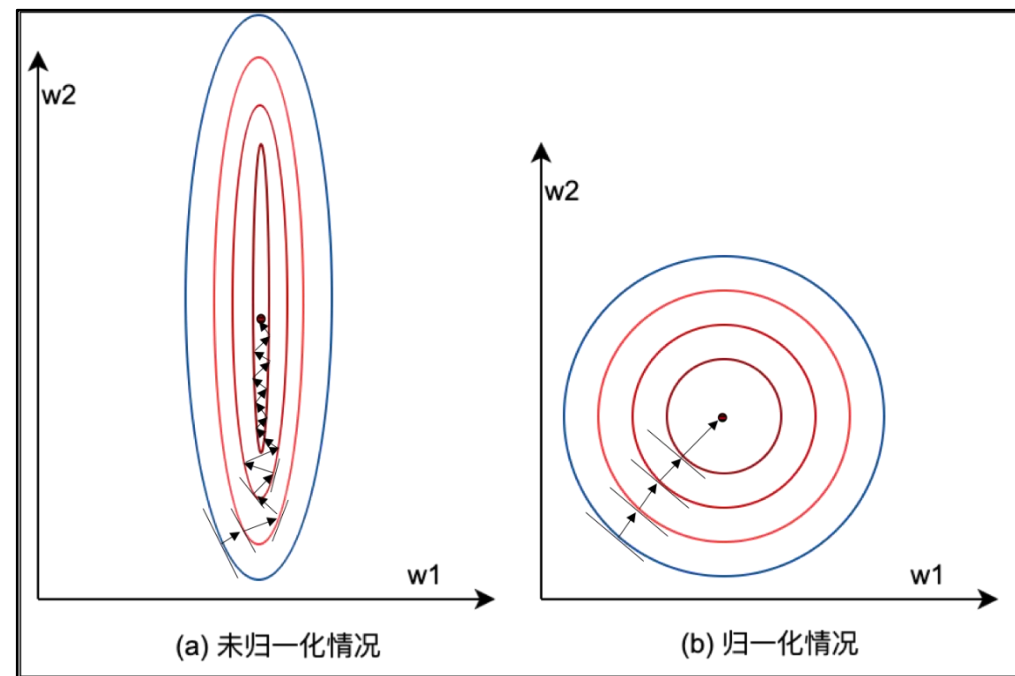
$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2.$$

优化的目标函数：

$$L(\theta) = (\hat{y} - y)^2 = (\theta_0 + \theta_1 x_1 + \theta_2 x_2 - y)^2$$

其中 $y$ 为真实的房子总价。

$x_1$ 和 $x_2$ 取值范围差别比较大。



数据归一化对梯度的影响

# 数据归一化

最常用的数据归一化方法

1、**Min-max归一化**：将结果映射到[0,1]之间。

$$\hat{x} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

2、**Z-score归一化**（标准归一化）：

$$\hat{x} = \frac{x - \mu}{\sigma}$$

其中 $\mu$ 为均值， $\sigma$ 为标准差

# 归一化

- 在深度神经网络中，由于神经层数比较多，在通过梯度下降训练时，前面层参数变化会导致输入到下一层的分布发生变化，且越往后这个变化越大。
- 如果前面发生了一个小偏移，到了越深的层这个偏移就会被扩大。这样是不利于网络的训练。
- 因此后面介绍的归一化方法主要为了**稳定层输入的分布**，来改善神经网络训练。



# 归一化

## 常见归一化方法：

1. 批归一化 (Batch Normalization, **BN**)
2. 层归一化 (Layer Normalization, **LN**)
3. 实例归一化 (Instance Normalization, **IN**)
4. 组归一化 (Group Normalization, **GN**)

# 批归一化 (BN)

**动机:** Internal Covariate Shift, 神经网络中每层的输入分布不一致

**思路:** 对每一批数据进行归一化

**Require:** 每个 Mini-Batch 上的  $x : B = \{x_1 \dots x_m\}$ ;

需学习的参数:  $\gamma, \beta$

**Ensure:**  $\{y_i = \text{BN}_{\gamma\beta}(x_i)\}$

- 1: Mini-Batch 均值:  $\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$
- 2: Mini-Batch 方差:  $\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$
- 3: 进行归一化:  $\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
- 4: 缩放参数  $\gamma$ , 偏移参数  $\beta$ :  $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$

Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C]//International conference on machine learning. pmlr, 2015: 448-456.

# 批归一化 (BN)

- BN算法中引入了两个新参数, $\gamma$ 和 $\beta$ .
- $\gamma$ 和 $\beta$ 的值通过梯度下降进行优化  
初始值通常设为 $\gamma = 1, \beta = 0$

$$\frac{\partial \ell}{\partial \hat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_B^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_B} = \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\partial \ell}{\partial \mu_B} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

# 批归一化 (BN)

训练阶段，均值 $\mu$ 和方差 $\sigma$ 是在每个mini-batch上计算的，在测试时则使用整个训练集上的统计量

$$E[x] \leftarrow E_B[\mu_B]$$

$$Var[x] \leftarrow \frac{m}{m-1} E_B[\sigma_B^2]$$

$$y = \frac{\gamma}{\sqrt{Var[x] + \varepsilon}} \cdot x + \left( \beta - \frac{\gamma \cdot E[x]}{\sqrt{Var[x] + \varepsilon}} \right)$$

# 批归一化 (BN)

## 例1: 展示BatchNorm在MINIST数据集上应用效果

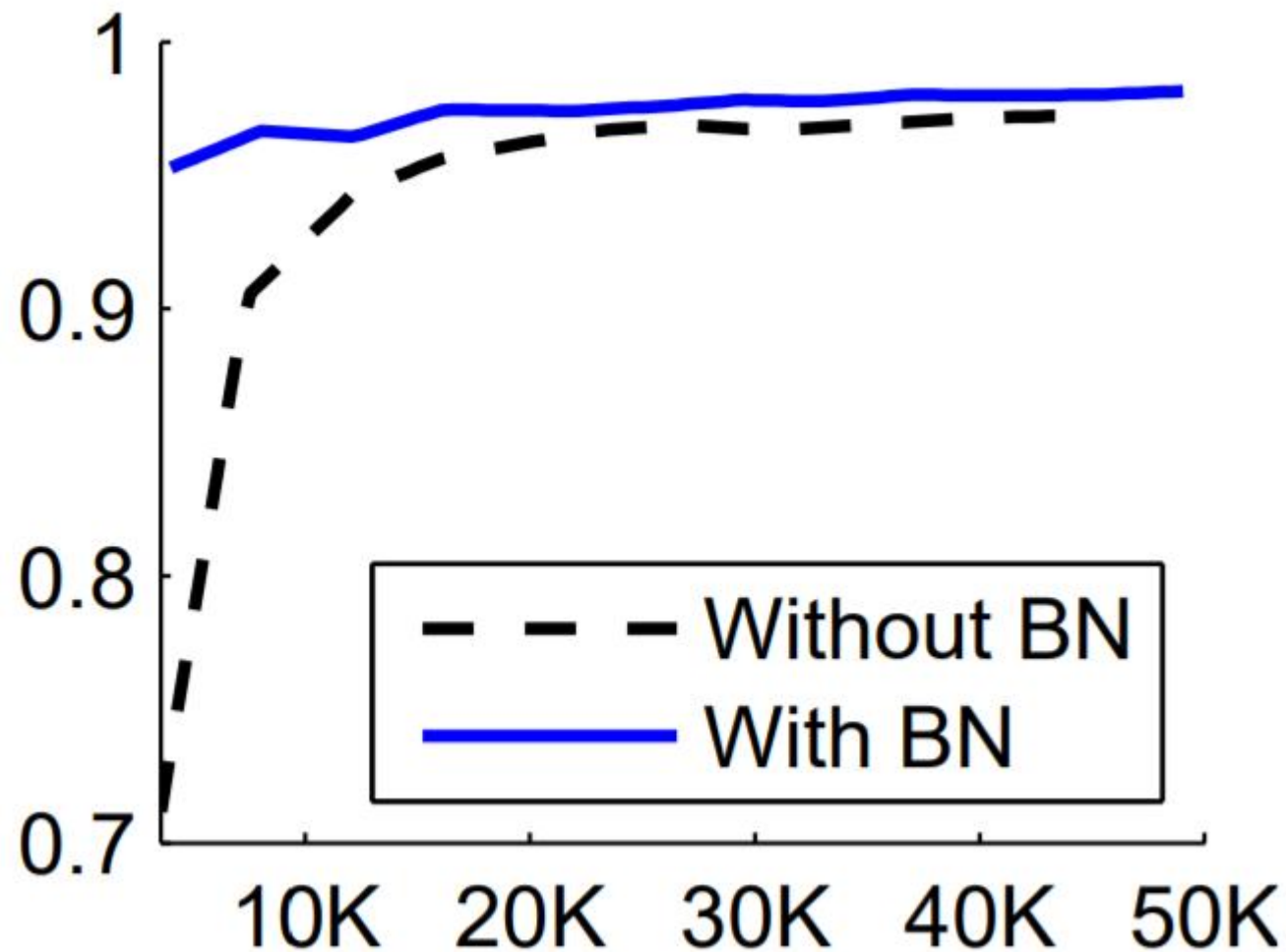
构建神经网络结构如下:

1. **基础版本**: 输入为28x28的图片。隐藏层为3个全连接层, 每层包含100个神经元, 使用sigmoid激活函数。输出层是个包含10个神经元的全连接层, 使用交叉熵作为损失函数
2. **BatchNorm版本**: 在基础版本的基础上, 每个隐藏层后面添加一个BatchNorm层。

# 批归一化 (BN)

图中展示了测试准确率和训练步数之间的关系

可以看到, BatchNorm能够有效加快网络收敛速度,并能够提高网络泛化性能。

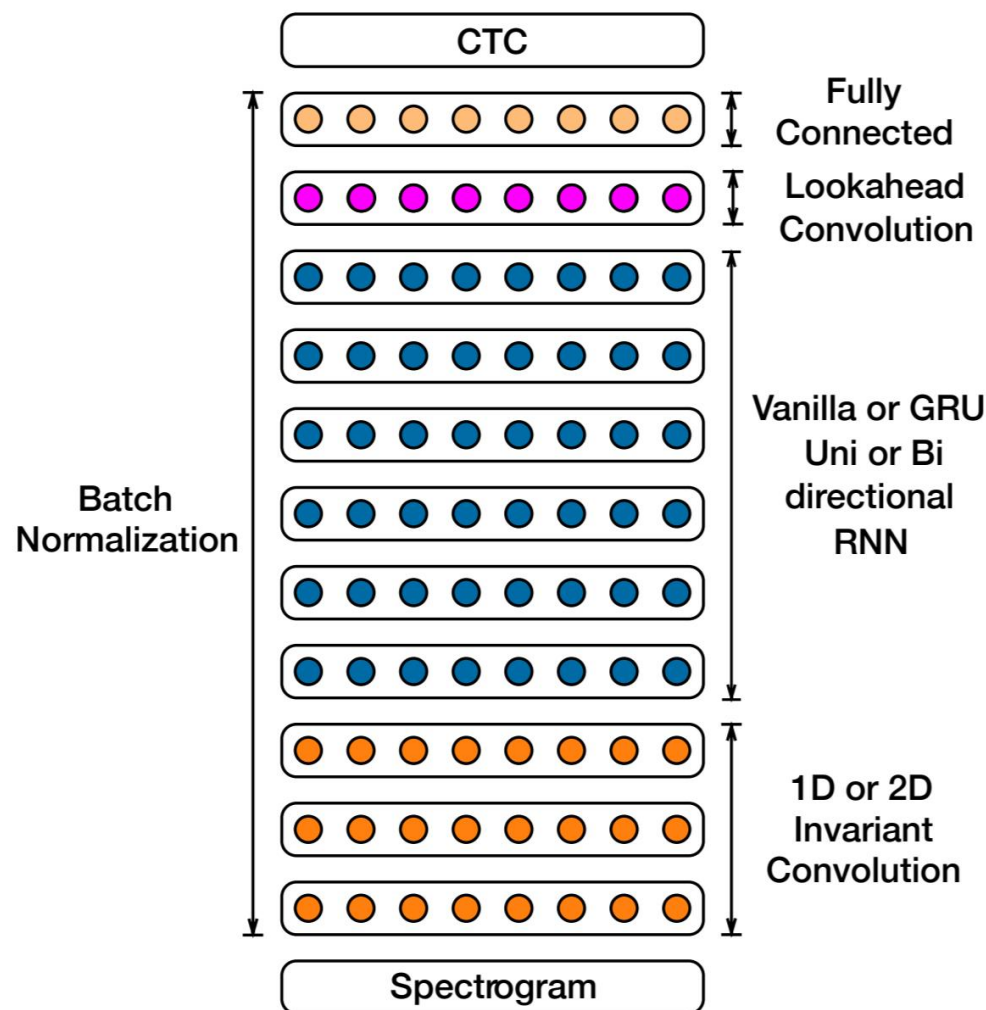


# 批归一化 (BN)

**例2:** 展示BatchNorm在语音识别任务上的效果搭建了如图所示的神经网络:

1. **基础版本:** 网络的输入在最下层, 为语音的频谱特征。网络中间包含了可变数量的双向RNN层 (标为蓝色)。输出层采用CTC损失函数。

2. **BatchNorm版本:** 隐藏层后面都接上一BN层





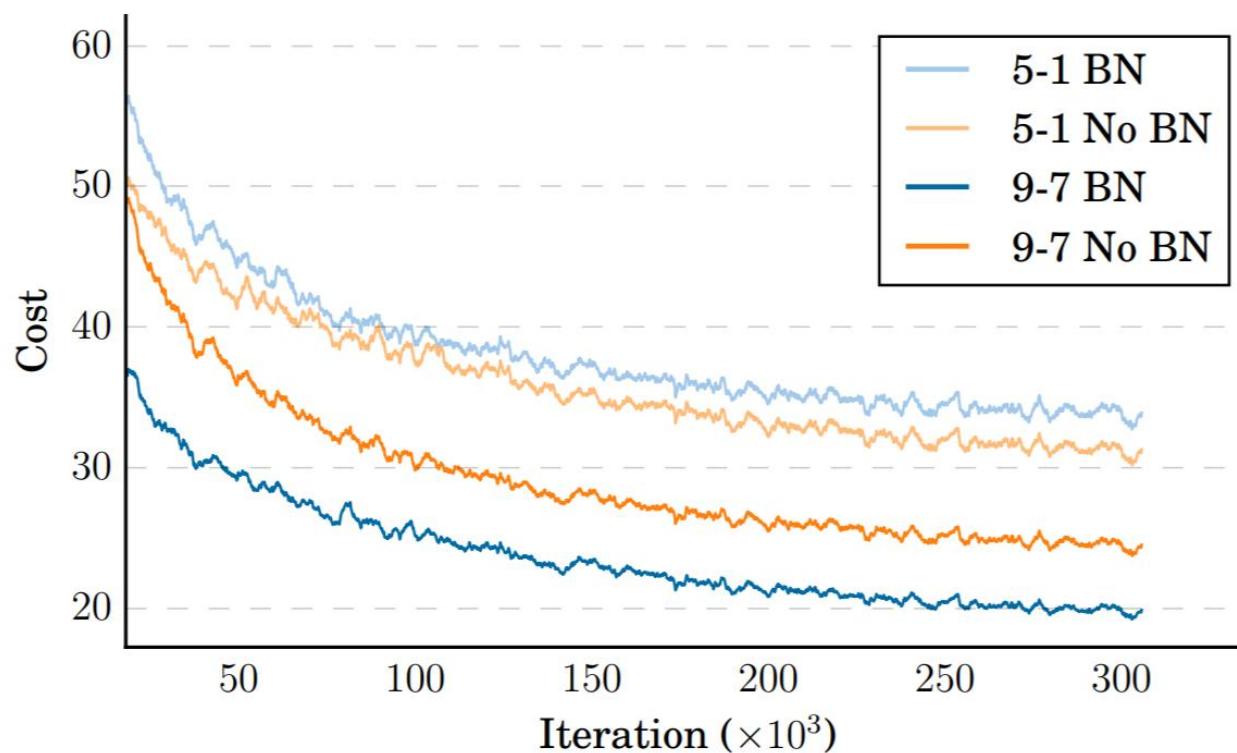
# 批归一化 (BN)

图中展示网络的训练损失函数和训练步数的关系

这里5-1指隐藏层总层数为5，其中双向RNN层数为1

9-7同理

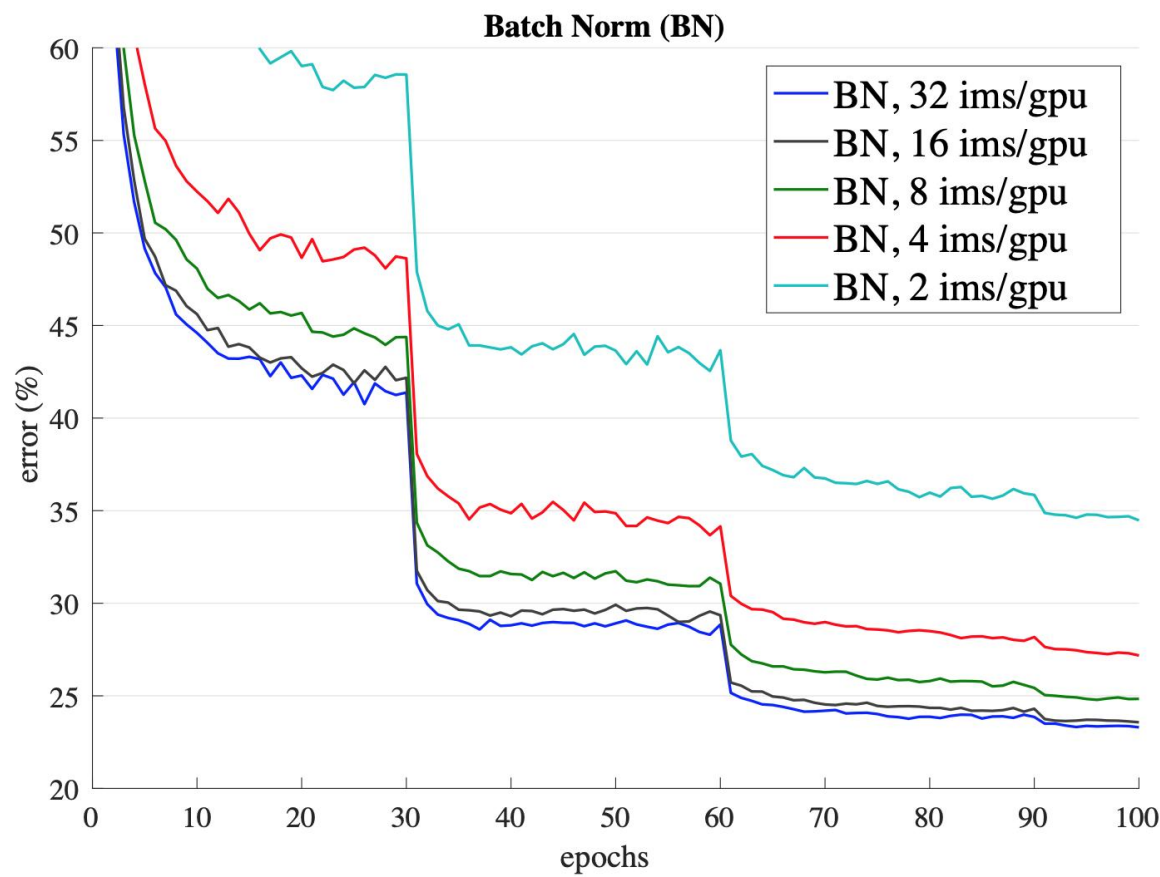
可以看到：BatchNorm可以有效加快收敛速度并进一步降低训练损失。





# 批归一化 (BN)

缺点：训练与测试时统计量不一致，使用小的batch sizes时统计量不稳定；  
不适用于RNN等动态网络



# 层归一化 (LN)

思路：对中间层的所有神经元进行归一化

算法流程：

1. 计算网络层的均值和方差

$$\mu^{(l)} \leftarrow \frac{1}{n^{(l)}} \sum_{i=1}^{n^{(l)}} x_i^{(l)} \quad \text{求均值}$$

$$\sigma^{(l)2} \leftarrow \frac{1}{n^{(l)}} \sum_{i=1}^{n^{(l)}} (x_i^{(l)} - \mu^{(l)})^2 \quad \text{求方差}$$

$n^{(l)}$ : 第 $l$ 层神经元数量

$x_i^{(l)}$ : 第 $l$ 层第 $i$ 个神经元的输入

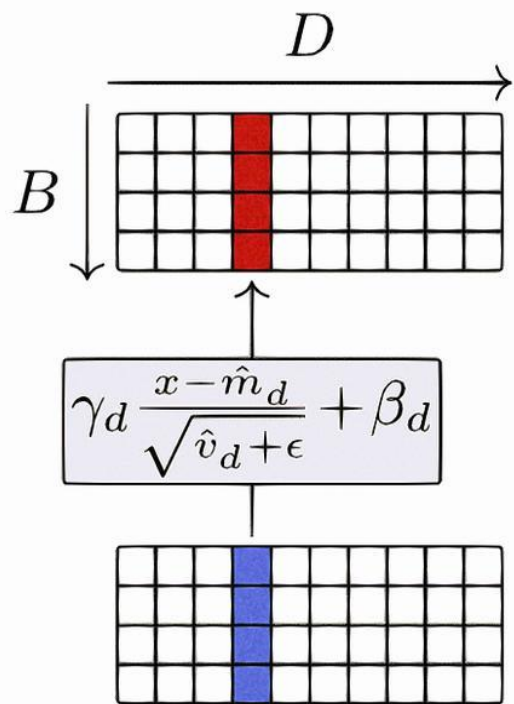
2. 进行归一化

$$\hat{x}_i^{(l)} \leftarrow \frac{x_i^{(l)} - \mu^{(l)}}{\sqrt{\sigma^{(l)2} + \epsilon}}$$

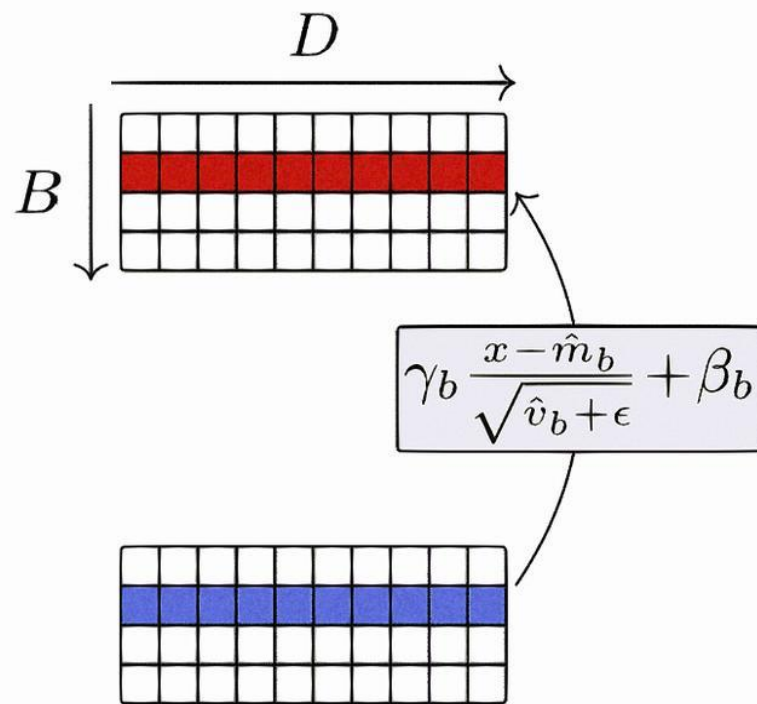
3. 设置可训练的缩放和偏移参数，映射数据

$$y_i^{(l)} \leftarrow \gamma \hat{x}_i^{(l)} + \beta \equiv LN_{\gamma, \beta}(x_i^{(l)})$$

# 批归一化和层归一化的对比



*batchnorm 1d*



*layernorm 1d*

# 实例归一化 (IN)

- 主要用于依赖于某个图像实例的任务。
- 假设输入的特征图为  $x \in R^{N \times C \times H \times W}$ ;

N: batch维度, C: 特征通道维度, H、W: 特征图高和宽维度。

$$\begin{aligned}\mu_{nc} &\leftarrow \frac{1}{HW} \sum_{w=1}^W \sum_{h=1}^H x_{nchw}, & \hat{x}_{nchw} &\leftarrow \frac{x_{nchw} - \mu_{nc}}{\sqrt{\sigma_{nc}^2 + \epsilon}}, \\ \sigma_{nc}^2 &\leftarrow \frac{1}{HW} \sum_{w=1}^W \sum_{h=1}^H (x_{nchw} - \mu_{nc})^2, & y_{nchw} &\leftarrow \gamma \hat{x}_{nchw} + \beta \equiv \text{IN}_{\gamma, \beta}(x_{nchw}).\end{aligned}$$

# 实例归一化 (IN)

- 思路：对每个样本的H和W的数据求均值和标准化，保留N、C维度。



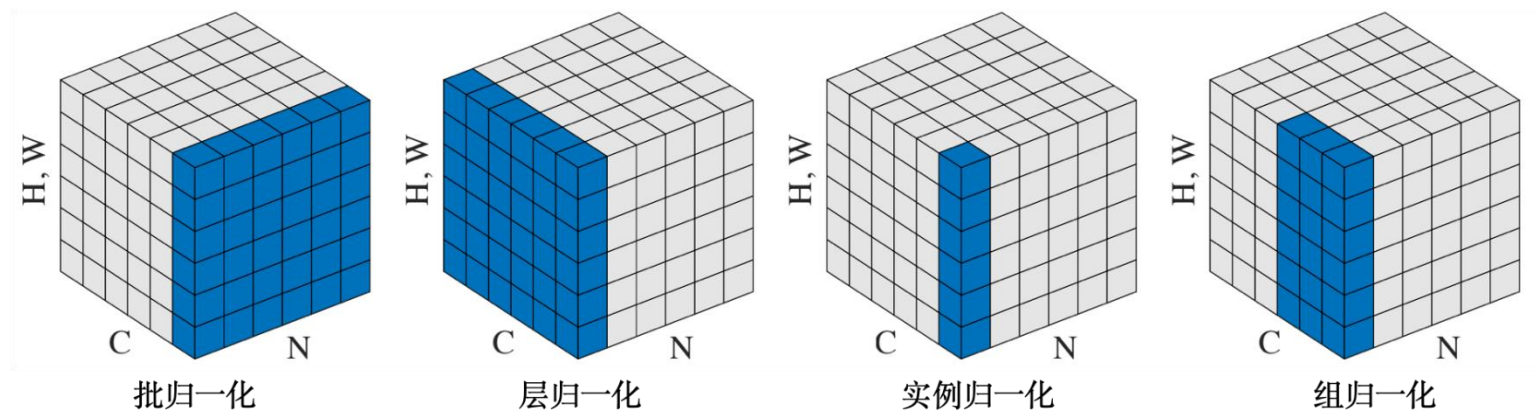
# 组归一化 (GN)

- 主要用于依赖于某个图像实例的任务。

- 输入的特征图为  $x \in R^{N \times C \times H \times W}$ ;

N: batch维度, C: 特征通道维度, H、W: 特征图高和宽维度。

- **思路**: 把特征通道分为G组, 每组有C/G个特征通道, 在组内进行归一。



- **组归一化是层归一化和实例归一化的折中。**



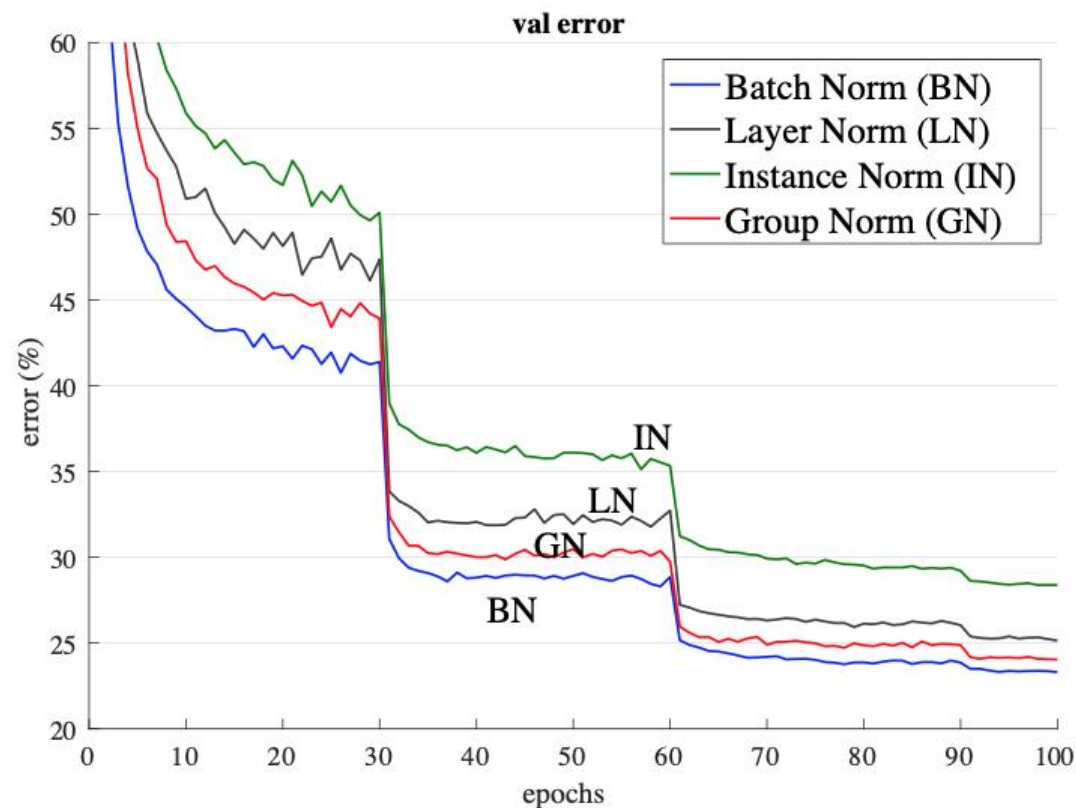
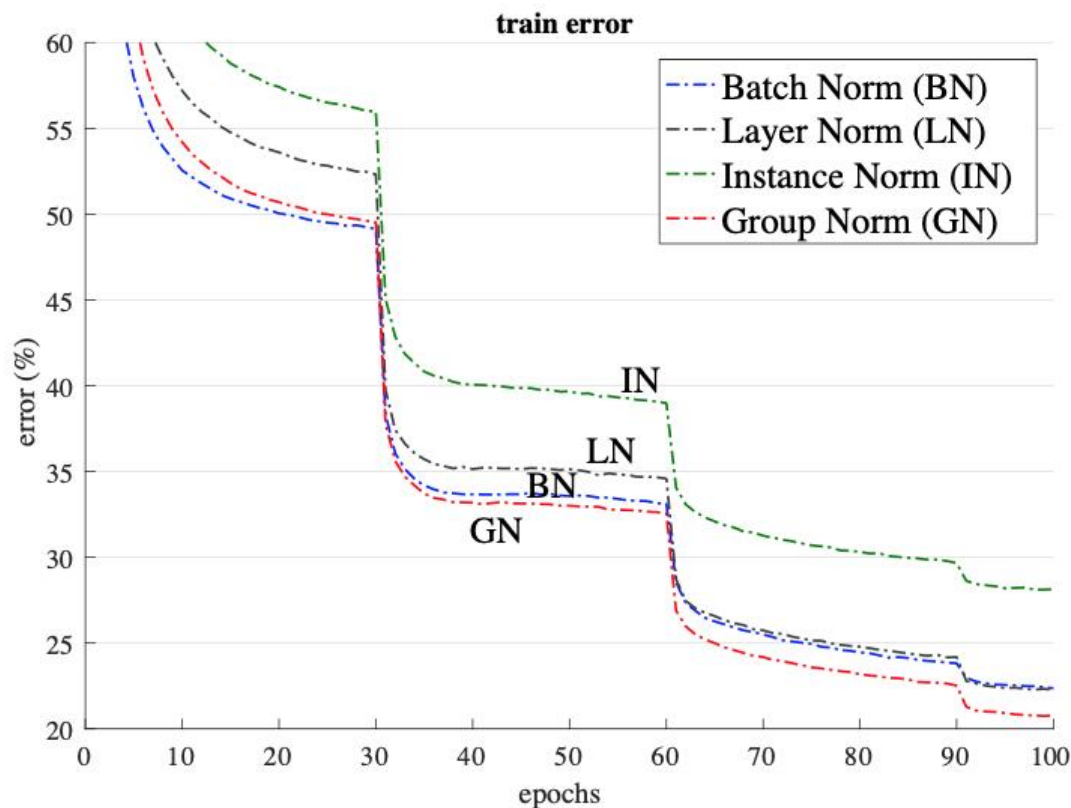
# 归一化：比较

- 神经网络内部的归一化为参数提供了缩放不变性，有助于缓解梯度消失与梯度爆炸

	权重矩阵缩放	权重向量缩放	输入向量缩放
Batch Normalization	不变	不变	不变
Layer Normalization	不变	变化	不变
Instance Normalization	不变	不变	不变
Group Normalization	不变	变化	不变

# 归一化：比较

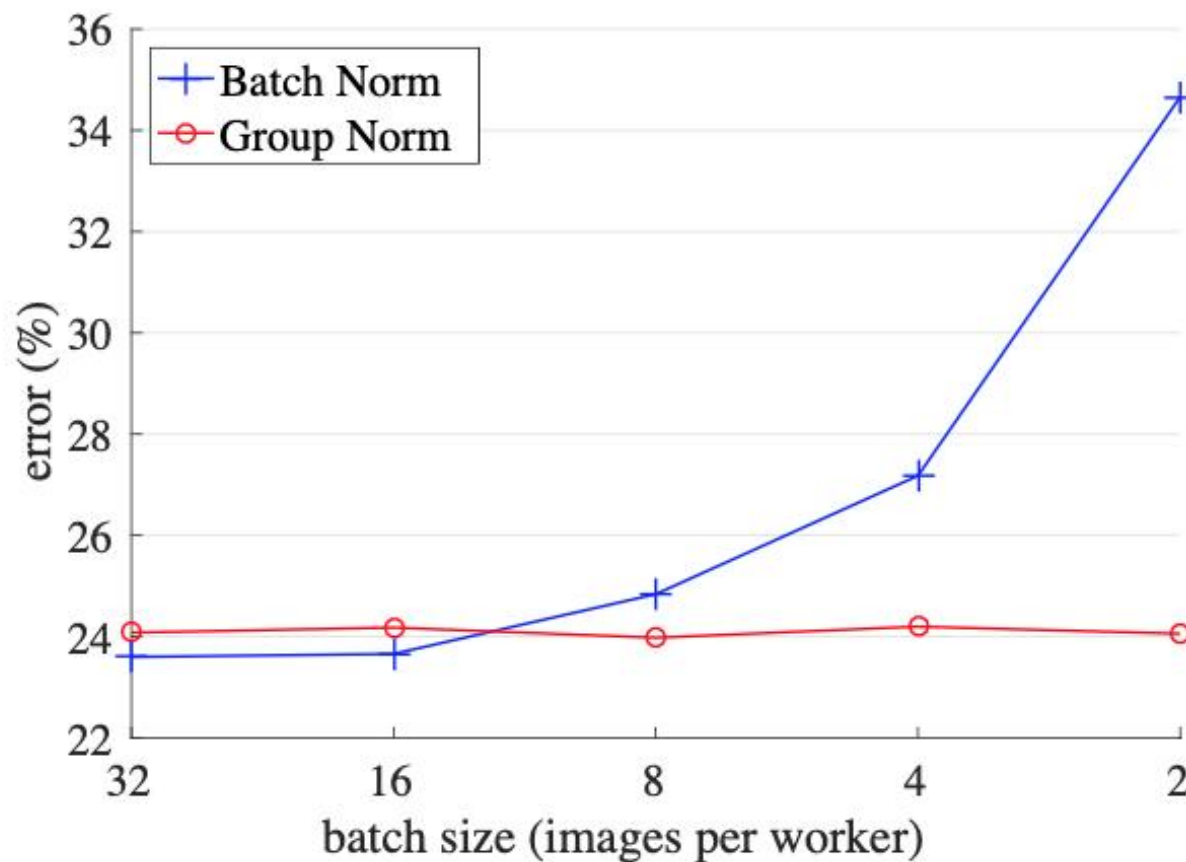
**例1：**对比几种不同归一化算法在ImageNet数据集上batch size设置为32时的训练和验证误差：





# 归一化：比较

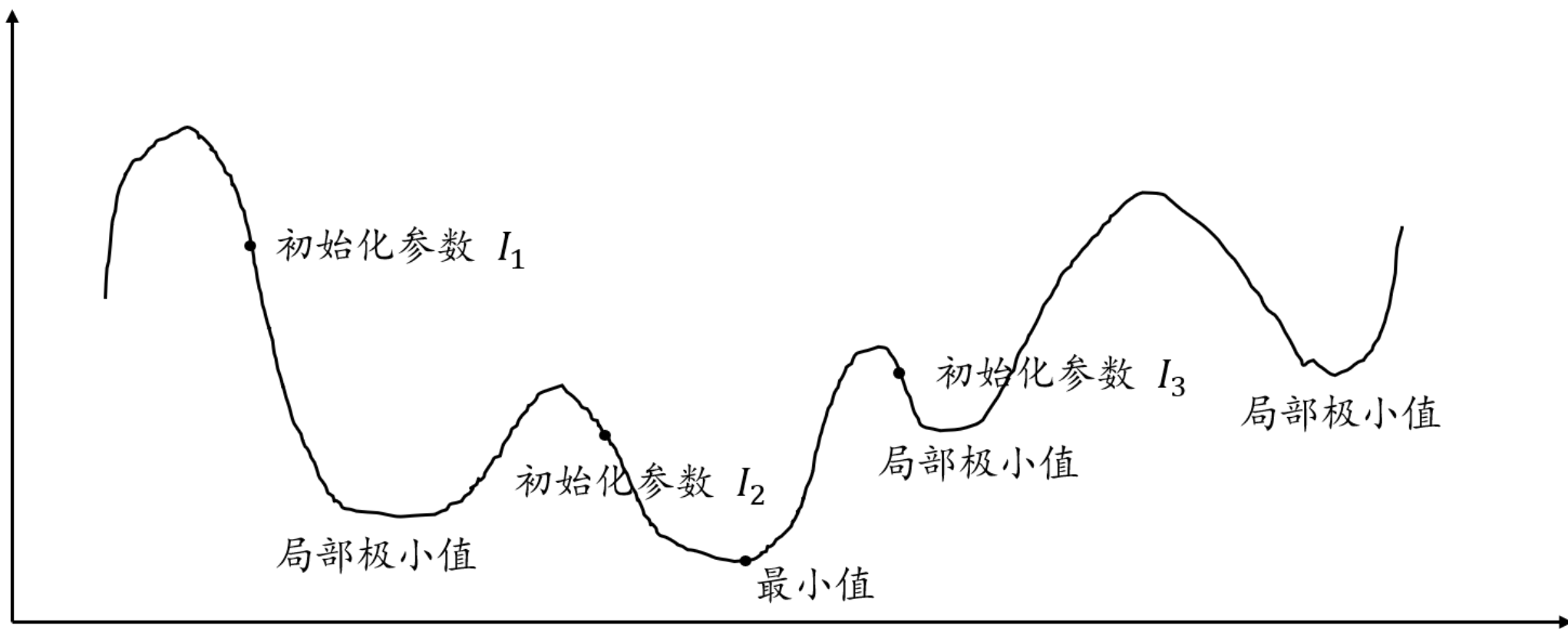
**例2：**对比批归一化和组归一化算法在ImageNet数据集上对于不同batch size的误差变换情况：



05

## 参数初始化

## 网络参数初始化为什么重要？

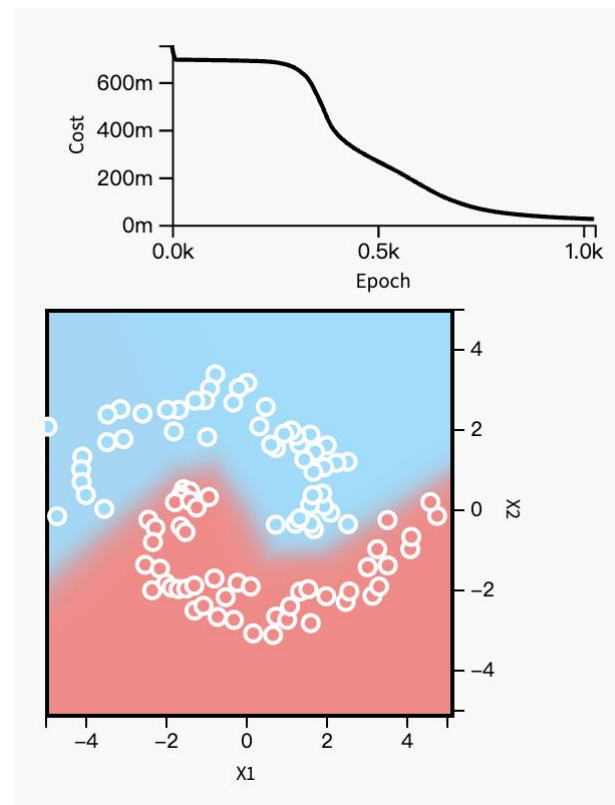
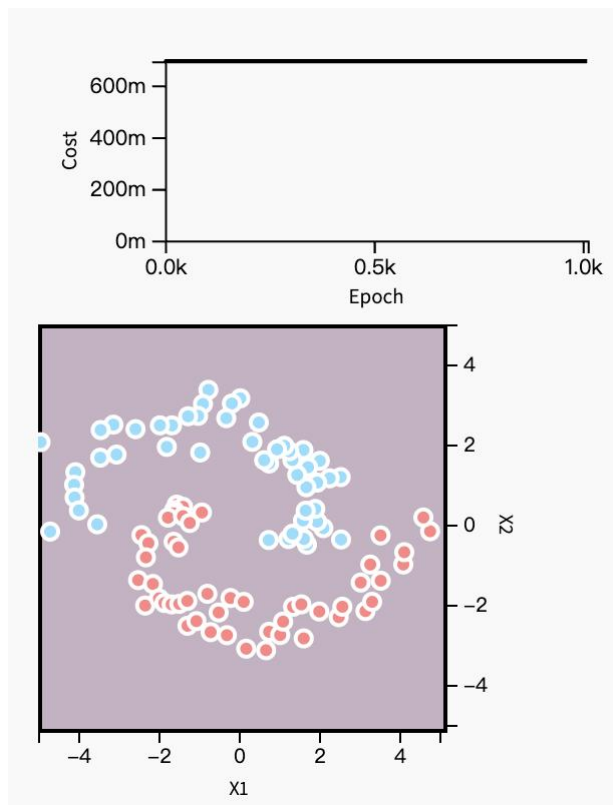


不同的初始化参数，计算得到的“全局最小”差距很大

- 多层神经网络的目标函数通常是非凸函数，求解困难。
- 常用的优化算法的基本思想：针对目标函数的梯度下降，局部极小值点的存在使得优化的结果很难是最优解。
- **好的初始化值能够帮助网络更快地计算得到最优值，更容易收敛到目标函数。**

# 全0初始化

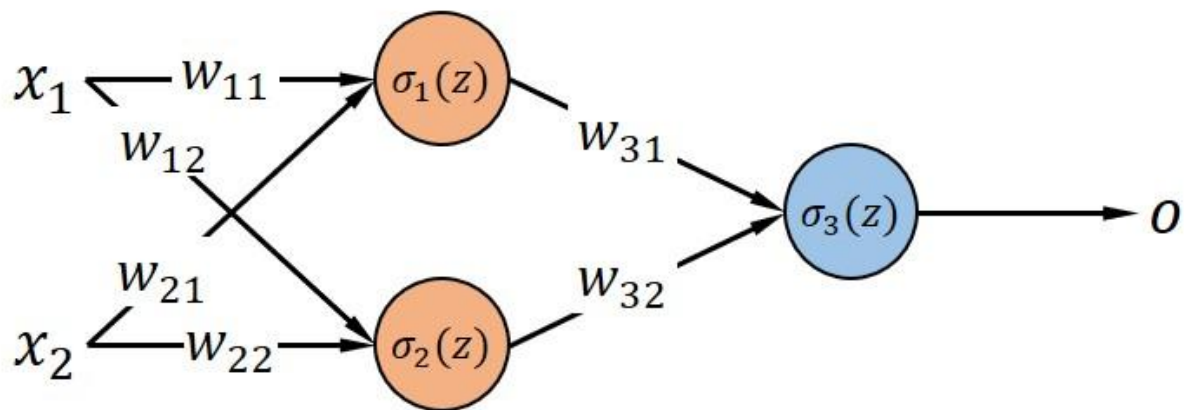
- 参数初始化最简单的方式：把神经网络参数全部初始化为0。
- 左图是全0初始化后的训练效果，右图是选用合适方法初始化后的训练效果。



# 网络参数初始化

- 目前没有发现一种初始化方式可以适用于任何网络结构
- 初始化需要避免“对称权重”现象（唯一确知的特性）

初始化  $w_{11} = w_{12} = w_{21} = w_{22} = w_{31} = w_{32} = 0$



回传梯度相等  $w_{11} = w_{21}, w_{12} = w_{22}, w_{31} = w_{32}$

# 初始化参数对训练的影响

不合理的初始化会导致**梯度消失或梯度爆炸**现象。

**梯度消失或梯度爆炸**是在训练过程中梯度的范数过大或过小的情况，引起这个现象的本质原因是梯度回传时进行的多次相乘，这个计算会很快放大或者缩小梯度的范数。

1. 大的梯度会使网络十分不稳定，会导致权重成为一个特别大的值，最终导致溢出而无法学习。
2. 小的梯度传过多层网络，达到靠近输入的隐藏层后会越来越小，导致隐藏层无法正常地进行学习。

# 权重矩阵初始化

## 启发式思考：

- 考虑激活函数的学习曲线，初始化权重应该分布在梯度较大的区域；
- 初始化权重不应过大或者过小
  - 优化角度：权重应该足够大来传播信息
  - 正则化角度：权重应该较小



# 权重矩阵初始化

## 常见的权重矩阵初始化方法：

1. 基于固定方差的参数初始化
2. 基于方差缩放的参数初始化
3. 正交初始化

# 基于固定方差的参数初始化

**主要思想：** 从一个固定均值(通常为 0)和方差的分布中采样来生成参数的初始值

- **方法1：**

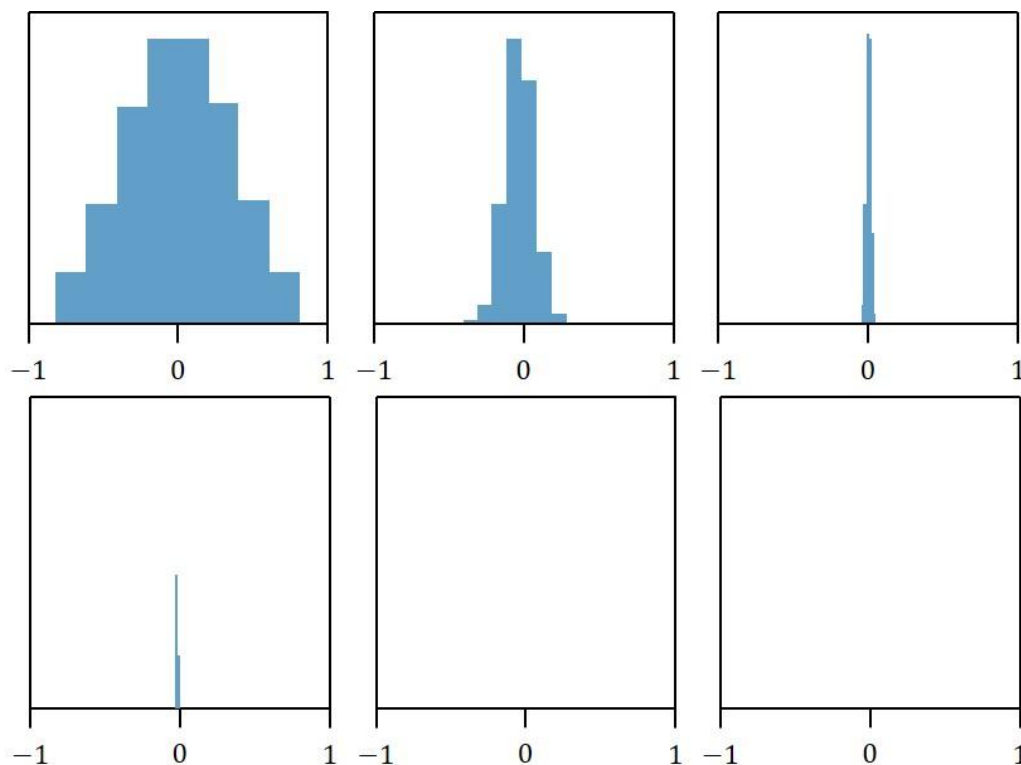
对于任意网络权值 $w_{ij}$ ，从区间  $[-r, r]$  的**均匀分布**中随机选取初始值。

- **方法2：**

对于任意网络权值 $w_{ij}$ ，从均值为 0、方差为常数  $k$  的**高斯分布**中随机选取初始值。

# 基于固定方差的参数初始化

- **缺点：**下图是采用**基于固定方差的参数初始化**后每层的激活函数输出值的分布，可以看出当神经网络的层数增多时，会发现越往后面层的激活函数的输出值几乎都接近于0，极易出现梯度消失。



# 基于方差缩放的参数初始化

- 要高效地训练神经网络，给参数选取一个合适的随机初始化区间是非常重要的。
- 一般而言，参数初始化的区间应该根据神经元的性质进行差异化的设置。如果一个神经元的输入连接很多，它的每个输入连接上的权重就应该小一些，以避免神经元的输出过大或过饱和。
- 初始化一个深度网络时，为了缓解梯度消失或爆炸问题，我们尽可能保持每个神经元的输入和输出的方差一致，根据神经元的连接数量来自适应地调整初始化分布的方差，这类方法称为**方差缩放**(Variance Scaling)。

# Xavier 初始化

- 假设在一个神经网络中，第  $l$  层的一个神经元  $a^{(l)}$ ，接收前一层的  $M_{l-1}$  个神经元的输出  $a^{(l-1)}$ ， $1 \leq i \leq M_{l-1}$ ：

$$a^{(l)} = f\left(\sum_{i=1}^{M_{l-1}} w_i^{(l)} a_i^{(l-1)}\right)$$

- 其中  $f(\cdot)$  为激活函数， $w_i^{(l)}$  为参数， $M_{l-1}$  是第  $l-1$  层神经元个数，简单起见，这里设置  $f(x) = x$

# Xavier 初始化

- 假设  $w_i^{(l)}$  和  $a_i^{(l-1)}$  的均值都为 0，并且互相独立，则  $a^{(l)}$  的均值和方差为：

$$\mathbb{E}[a^{(l)}] = \mathbb{E}\left[\sum_{i=1}^{M_{l-1}} w_i^{(l)} a_i^{(l-1)}\right] = \sum_{i=1}^{M_{l-1}} \mathbb{E}[w_i^{(l)}] \mathbb{E}[a_i^{(l-1)}] = 0$$

$$\begin{aligned}\text{var}(a^{(l)}) &= \text{var}\left(\sum_{i=1}^{M_{l-1}} w_i^{(l)} a_i^{(l-1)}\right) \\ &= \sum_{i=1}^{M_{l-1}} \text{var}(w_i^{(l)}) \text{var}(a_i^{(l-1)}) \\ &= M_{l-1} \text{var}(w_i^{(l)}) \text{var}(a_i^{(l-1)})\end{aligned}$$

- 也就是说，输入信号的方差在经过该神经元后被放大或缩小了  $M_{l-1} \text{var}(w_i^{(l)})$  倍。

# Xavier 初始化

- 为了使得在经过多层网络后，信号不被过分放大或过分减弱，我们尽可能保持每个神经元的输入和输出的方差一致。这样  $M_{l-1} \text{var}(w_i^{(l)})$  设为 1 比较合理，即：

$$\text{var}(w_i^{(l)}) = \frac{1}{M_{l-1}}$$

- 同理，为了使得在反向传播中，误差信号也不被放大或缩小，需要将  $w_i^{(l)}$  的方差保持为：

$$\text{var}(w_i^{(l)}) = \frac{1}{M_l}$$

# Xavier 初始化

- 作为折中，同时考虑信号在前向和反向传播中都不被放大或缩小，可以设置为：

$$\text{var}(w_i^{(l)}) = \frac{2}{M_{l-1} + M_l}$$

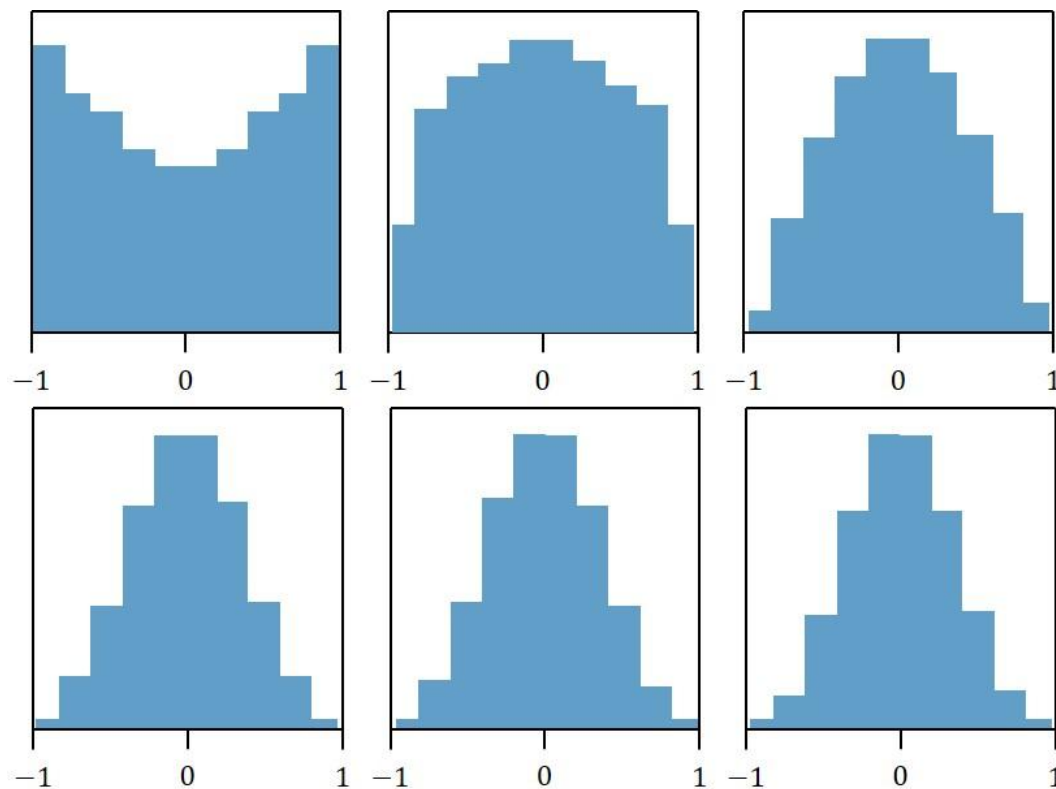
- 在计算出参数的理想方差后，可以通过以下高斯分布或均匀分布采样来随机初始化参数

初始化方法	激活函数	均匀分布 $[-r, r]$	高斯分布 $\mathcal{N}(0, \sigma^2)$
Xavier 初始化	Logistic	$r = 4\sqrt{\frac{6}{M_{l-1}+M_l}}$	$\sigma^2 = 16 \times \frac{2}{M_{l-1}+M_l}$
Xavier 初始化	Tanh	$r = \sqrt{\frac{6}{M_{l-1}+M_l}}$	$\sigma^2 = \frac{2}{M_{l-1}+M_l}$



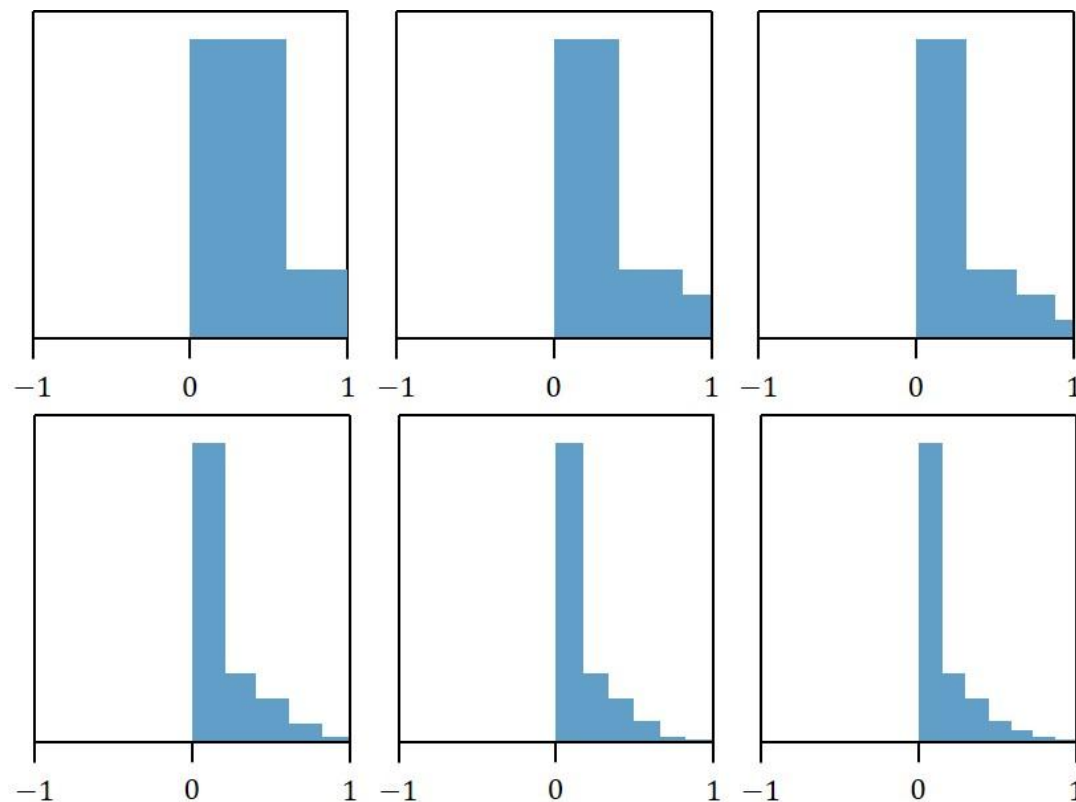
# Xavier 初始化

- **优点：**下图是采用**Xavier 初始化**后每层的激活函数( $\tanh$ )输出值的分布，可以看出深层的激活函数输出值还是非常服从标准高斯分布。



# Xavier 初始化

- **缺点：**采用Xavier 初始化对ReLU激活函数的效果不佳，可以看出当达到5、6层后又开始逐渐趋向于0。



# He 初始化

- 当第 $l$ 层神经元使用 ReLU 激活函数时，通常有一半的神经元输出为 0，因此其分布的方差也近似为使用恒等函数时的一半。这样，只考虑前向传播时，参数  $w_i^{(l)}$  的理想方差为

$$\text{var}(w_i^{(l)}) = \frac{2}{M_{l-1}}$$

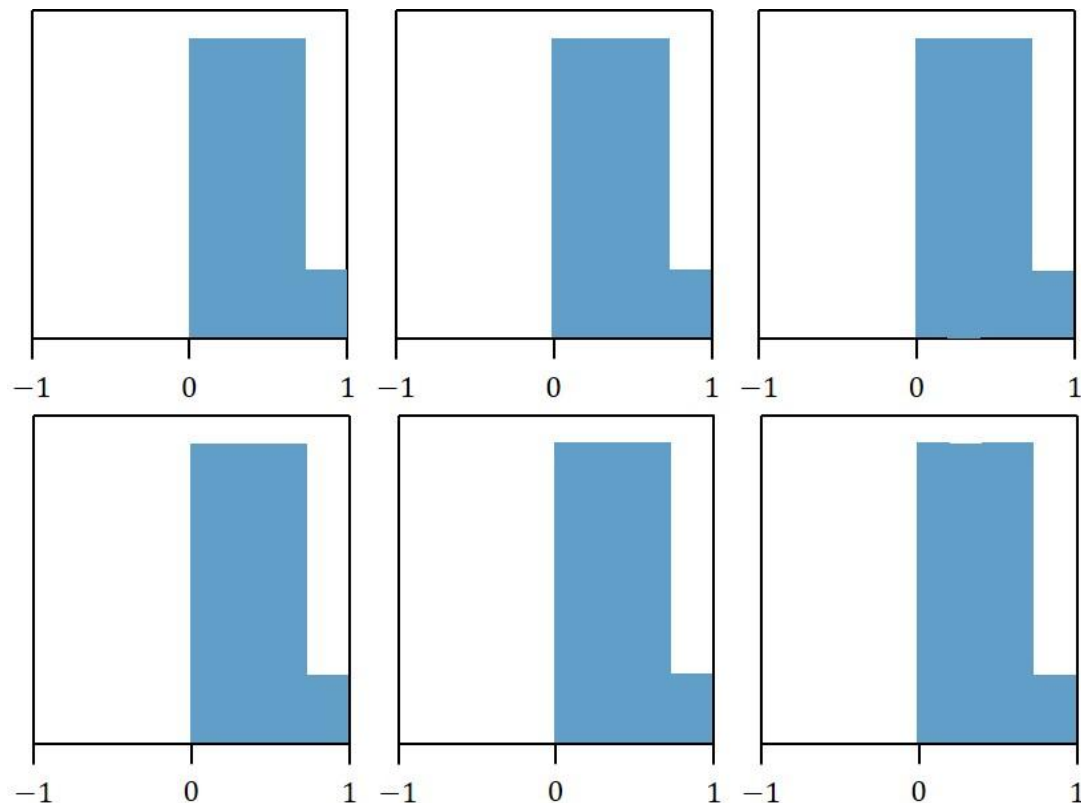
其中  $M_{l-1}$  是第  $l-1$  层神经元个数

- 从而，当使用 ReLU 激活函数时可以通过以下高斯分布或均匀分布采样来随机初始化参数

初始化方法	激活函数	均匀分布 $[-r, r]$	高斯分布 $\mathcal{N}(0, \sigma^2)$
He 初始化	ReLU	$r = \sqrt{\frac{6}{M_{l-1}}}$	$\sigma^2 = \frac{2}{M_{l-1}}$

# He 初始化

- 下图是采用 **He 初始化** 权重后，隐藏层使用ReLU时，激活函数的输出值的分布情况，可以看出针对ReLU激活函数，He 初始化效果比 Xavier 初始化好很多。



# 正交初始化

**主要思想：** 正交初始化可以避免训练开始时就出现梯度消失或梯度爆炸现象

• **启发式规则：** 初始化权重矩阵为正交矩阵，满足  $W^T W = I$

步骤：

1. 用均值为0方差为1的高斯分布初始化一个矩阵
2. 将这个矩阵用奇异值分解得到两个正交矩阵，使用其中之一作为权重矩阵

实际使用中通常需要将正交矩阵乘以一个缩放系数

# 偏置矩阵初始化

偏置矩阵通常不需要考虑破坏对称性的问题，通常我们可以把偏置矩阵初始化为全0矩阵。

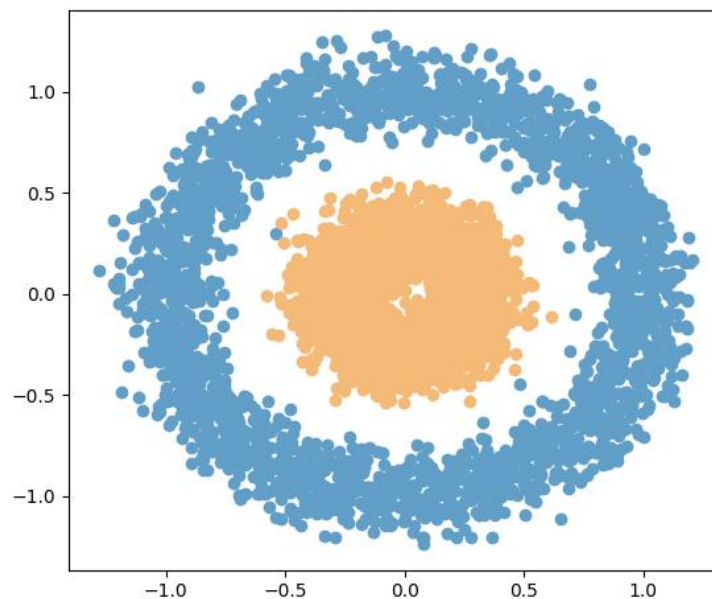
通常情况下，偏置矩阵还是会初始化为全0矩阵，除了一些例外情况：

- 偏置作为输出单元，初始化偏置以获得正确的输出边缘的统计是有利的；
- 需要选择偏置以避免初始化引起的太大饱和

# 初始化参数对训练的优化程度

**例：**人工生成一个“同心圆环”数据集，数据有2类，5000个样本。

- 数据分布如下图所示。



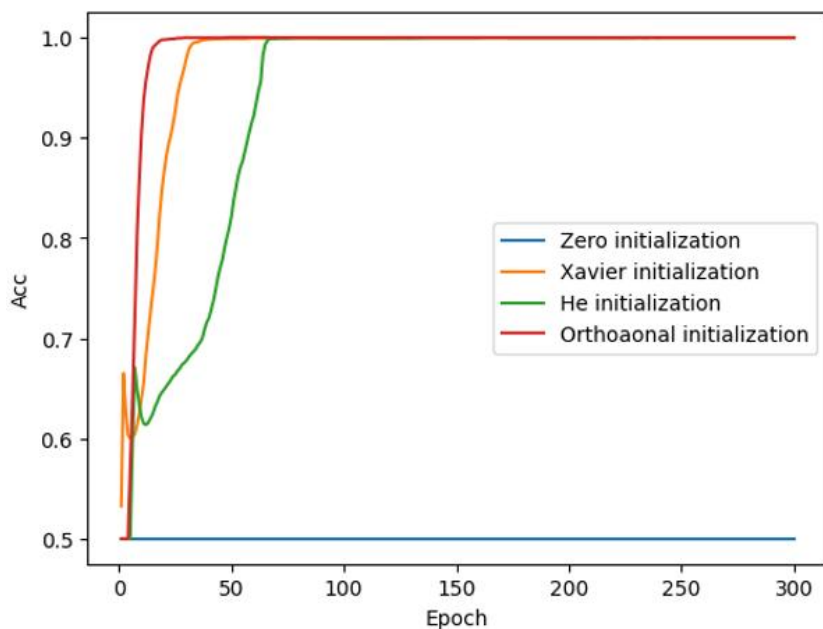
- 在这里，我们采用一个简单的多层神经网络，其输入输出层都是2个神经元，具体结构为每层[2,5,3,2]个神经元。输出层使用 $softmax$ 进行分类，其余层使用 $ReLU$ 函数作为激活函数。

# 初始化参数对训练的优化程度

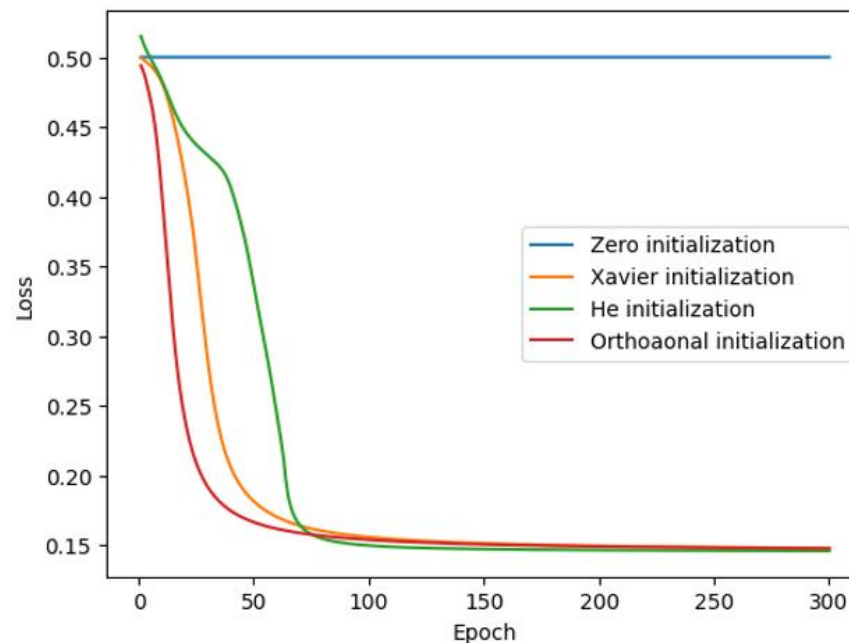
对比全0初始化和前面介绍的三种初始化方法：Xavier初始化、He初始化和正交初始化在这个数据集上的表现。

全0初始化将使得网络无法正确学习分类结果，其余几种初始化方式在这个问题上能够达到差不多的水平。

分类正确率



loss下降程度





# 初始化参数对训练的优化程度

- 即使采用了合理的初始化，梯度爆炸或消失的问题也会出现。
- 除了使用固定策略或者是随机值对模型参数进行初始化，还有可能使用网络训练的方法学习到模型的初始化参数。
- 下面讨论两种方式，分别是使用**无监督网络**训练的网络预训练方式和使用**监督网络**的迁移学习的方式。
- 无论是使用哪种策略，使用得当的情况下都能够使我们的网络获得更快的收敛率和更好的泛化误差。

06

## 网络预训练

# 网络预训练

**网络预训练**是采用相同结构的，并且已经训练好的网络权值作为初始值，在当前任务上再次进行训练。

为什么使用网络预训练？

- 为了能够在**更短时间内**训练得到**更好的网络性能**；
- 相似的任务之间，训练好的神经网络可以复用，通常作为特征提取器。

# 网络预训练

## 常用的预训练方法

- 无监督预训练

- 玻尔兹曼机
- 自编码器

- 有监督预训练

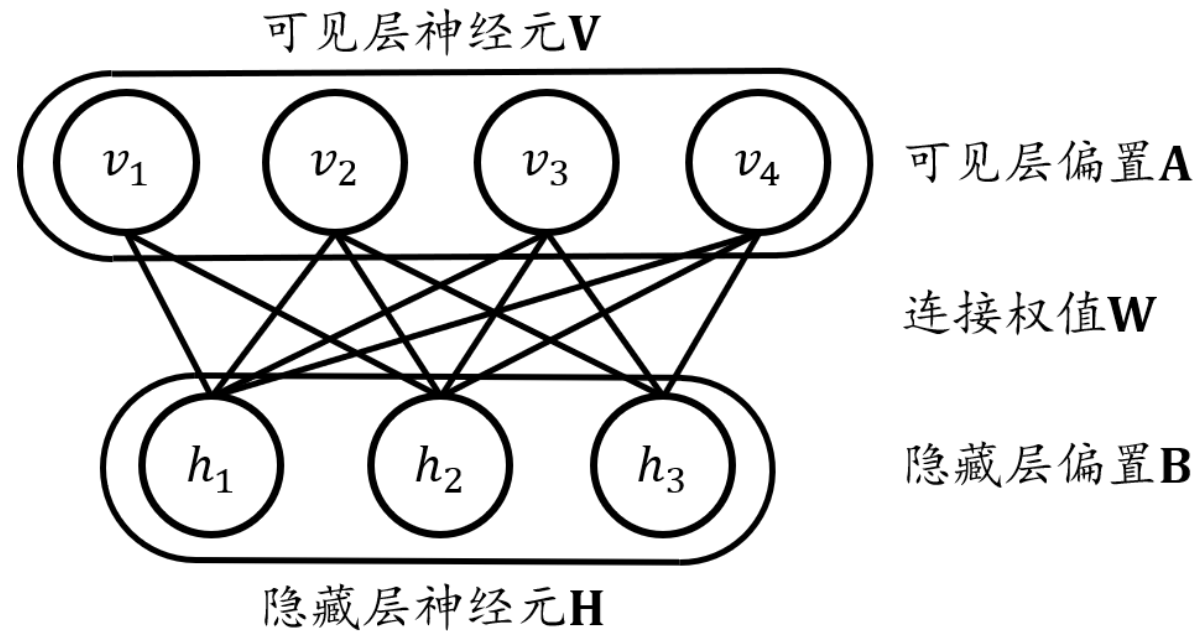
- 迁移学习

## 6.1 无监督预训练

# 玻尔兹曼机(BM)

- **玻尔兹曼机** (Boltzmann Machines, BM) : 一种对称连接的, 神经元根据能量函数计算概率进行激活的网络结构
- 常用的是改进版本: 限制玻尔兹曼机 (RBM)
- 单层RBM结构如图:

可见层神经元 $V$  接受输入,  
隐藏层神经元 $H$  得到特征。



# 玻尔兹曼机(BM)

RBM是生成式神经网络

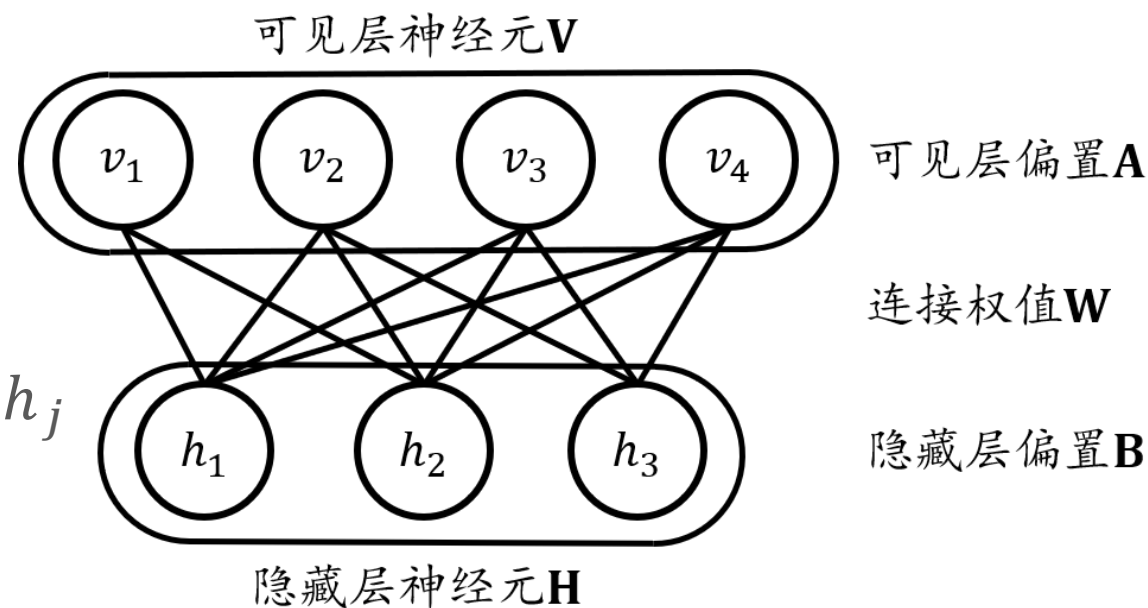
- 可见状态向量（输入向量） $v$ ，隐藏状态变量 $h$
- RBM描述的是 $(v, h)$ 的联合分布

- 能量函数：

$$E(v, h) = - \sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i w_{ij} h_j$$

- 训练目标：

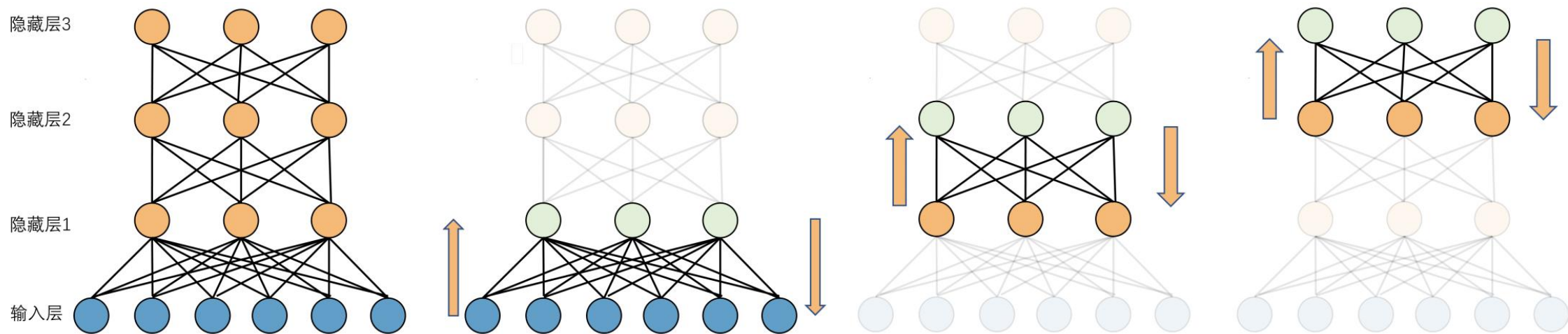
最小化RBM模拟的数据分布与数据实际分布之间的对比散度  
(contrastive divergence, CD)



# 玻尔兹曼机(BM)

**逐层预训练方法：**将多个预训练好的RBM堆叠起来就可以作为网络的初始化结果。由多层受限制玻尔兹曼机堆叠起来的网络叫作**深度信念网络** (Deep Belief Network, DBN)

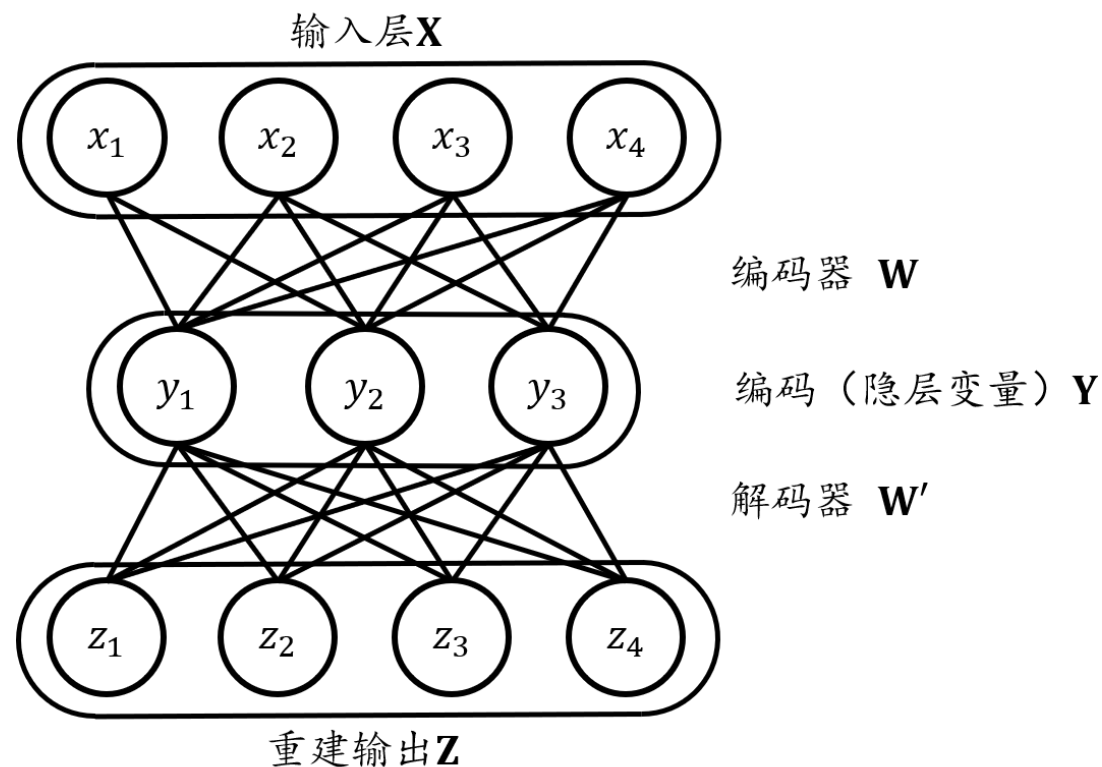
**堆叠方式：**将训练好的上一层输出作为下一层的输入





# 自编码器(AE)

**自编码器** (Auto-encoder, AE) : 是无监督的全连接网络结构, 其训练目标为重构误差最小化



单层AE的结构如图所示:

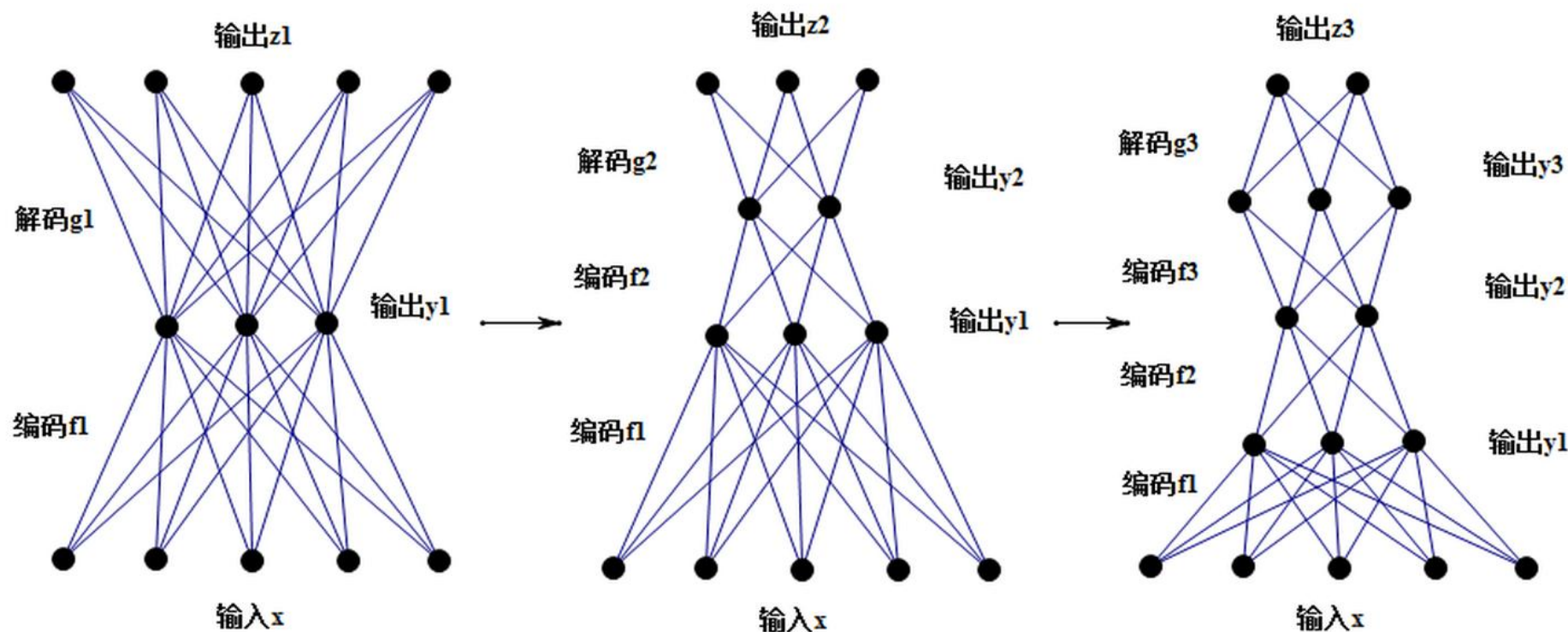
- 编码器: 高位数据转换为低维特征编码
- 解码器: 还原原始数据

# 自编码器(AE)

## 逐层预训练方法：

- (1) 给定初始输入 $x$ ，采用无监督方式训练第一层自动编码器，减小重构误差达到设定值。
- (2) 把第一个自动编码器隐含层的输出作为第二个自动编码器的输入，采用以上同样的方法训练自动编码器。
- (3) 重复第二步。

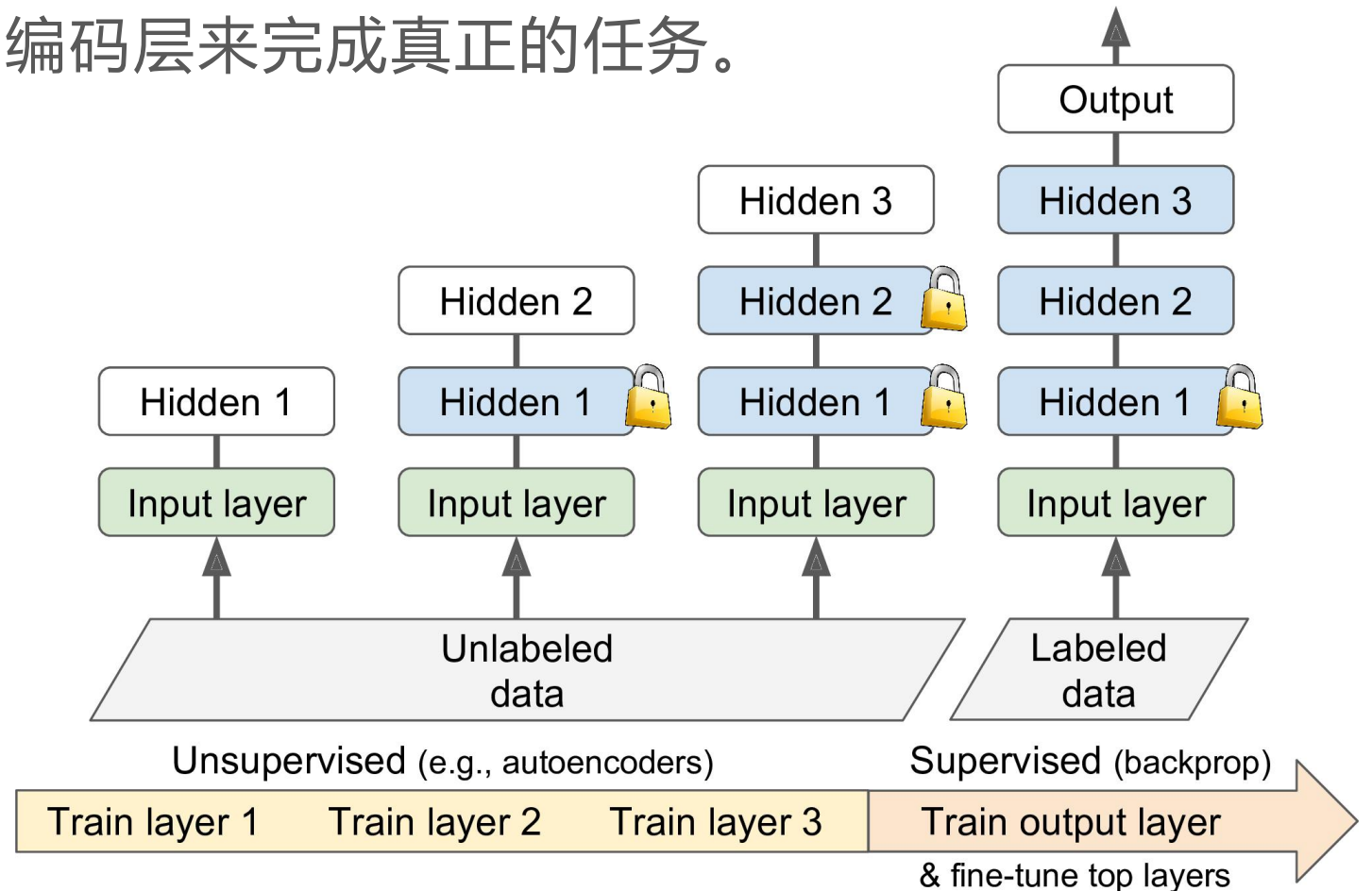
将多个自编码器堆叠起来的网络叫作  
**堆叠自编码器**  
(Stacked Auto-Encoder, SAE)



# 无监督预训练

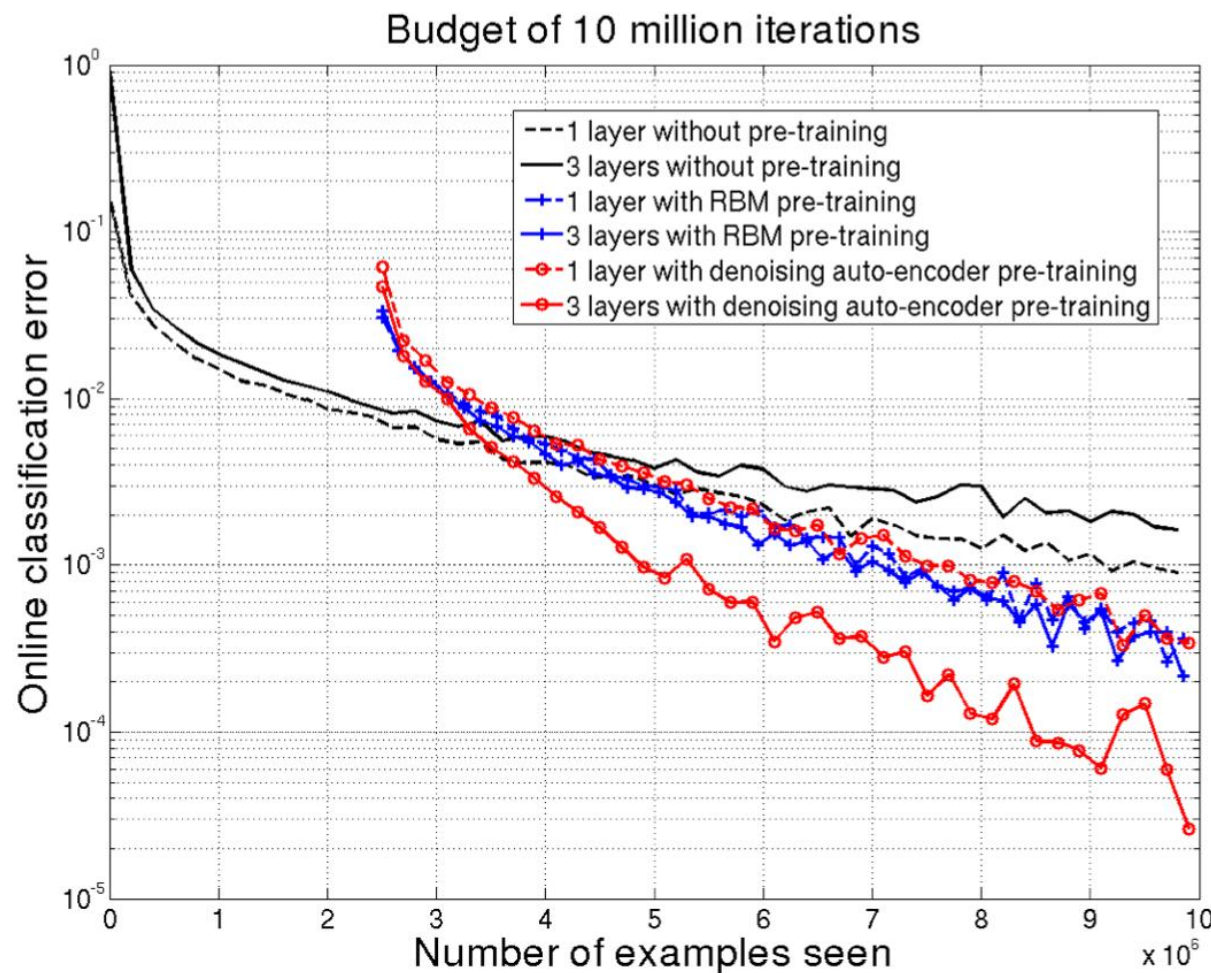
如果有一个很大的数据集但绝大部分是未标注数据，可以使用所有的数据先进行无监督预训练，然后复用编码层来完成真正的任务。

**DBN**和**SAE**是早期无监督预训练的  
代表性方法。



# 无监督预训练的效果

## 无预训练 VS RBM预训练 VS AE预训练



# 无监督预训练为何会起作用

- 神经网络的使用梯度下降法之所以训练困难，主要原因在于它的目标函数对于参数来说是非凸函数，因此在参数空间中存在多个局部极小值。
- 对于无监督的预训练所起到的作用，有这样的两个猜想：
  - 预训练相当于在优化过程中加入了限制条件，将参数放置在更适合监督训练的优化空间中；
  - 初始化将参数放置到了一个能够优化到更小极值的初始点上。
- Erhan等人实验结果显示，无监督预训练是一种不寻常的正则化方式。尽管是一种正则化方式，但是当训练数据很多的时候，它对最终的训练目标仍起到积极的作用。



## 6.2 有监督预训练

# 迁移学习

- 迁移学习：已经训练好的网络作为自己网络的权重初始值。
- 通过大量带标签的数据集，大幅减少网络收敛的训练时间。
- 即使这个网络与当前所需要解决的问题并不一样，已经训练好的网络权重作为特征提取器也是有可能可以复用的。

站在巨人的肩膀上，复用已经得到的研究成果。

# 迁移学习

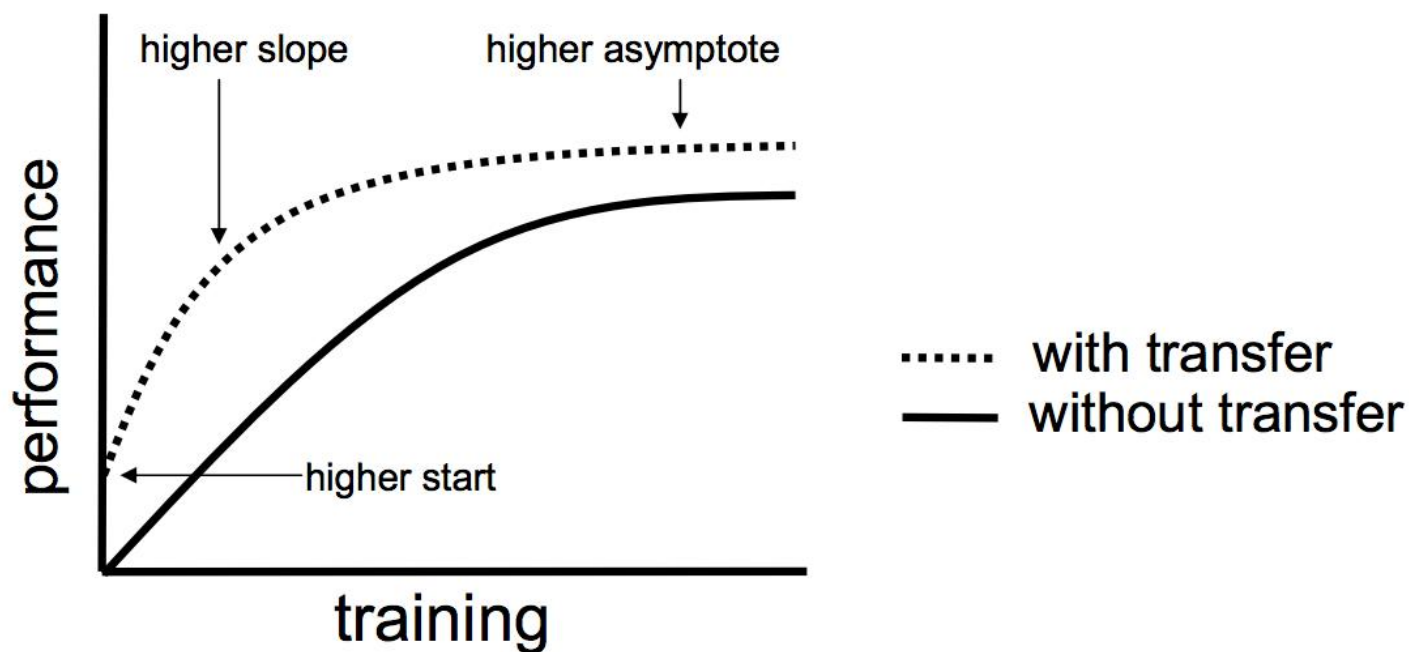
- 使用大型数据集预训练网络，可以使网络在其它任务上发挥更好的效果。
- 例如：我们已经得到使用大型人脸图片数据集训练好的网络，而现在的目标是对与这个数据集不重叠的另一些人脸进行识别。那么，即使网络顶层的分类器最终需要重新训练，中间网络层作为特征提取器，可以直接利用。



# 迁移学习

相比于从零开始直接训练一个模型，使用预训练模型帮助新模型的学习更好：

1. 更高的起点：初始模型的性能一般比随机初始化的模型要好；
2. 更高的斜率：训练时模型的学习速度比从零开始学习要快；
3. 更高的渐进：模型的最终性能更好。



# 迁移学习

使用预训练模型的方式：

- 直接作为特征提取网络
- 作为初始化模型进行微调

具体实践中，可以将预训练模型作为整体网络的全部或者一部分，也可以从预训练模型中截取需要的层数，具体如何使用取决于目标领域问题所需要的模型结构。

# Q&A

Questions and Answers