



全连接神经网络 (2)

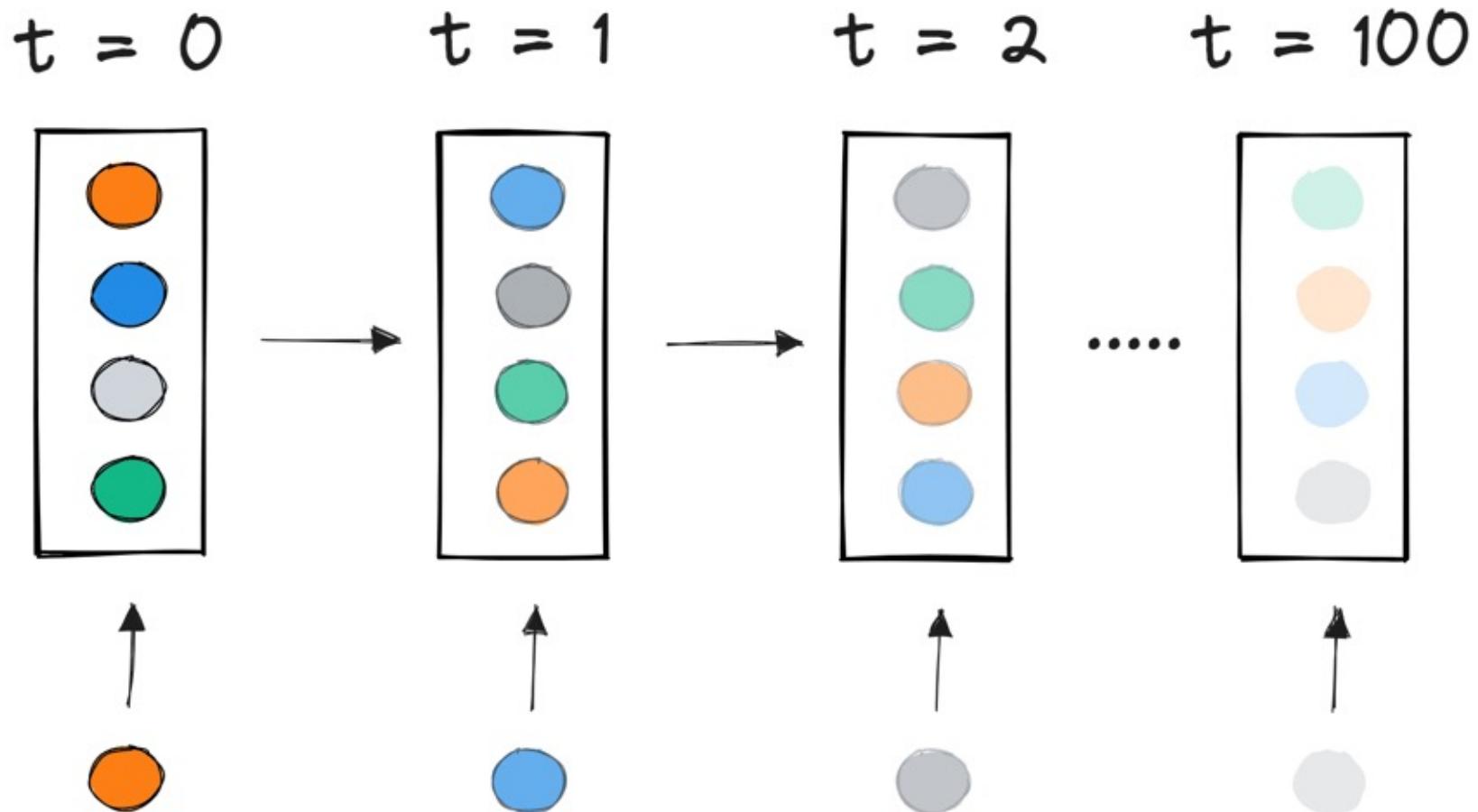
叶山 中国地质大学（北京）

yes@cugb.edu.cn

梯度下降的问题

梯度消失

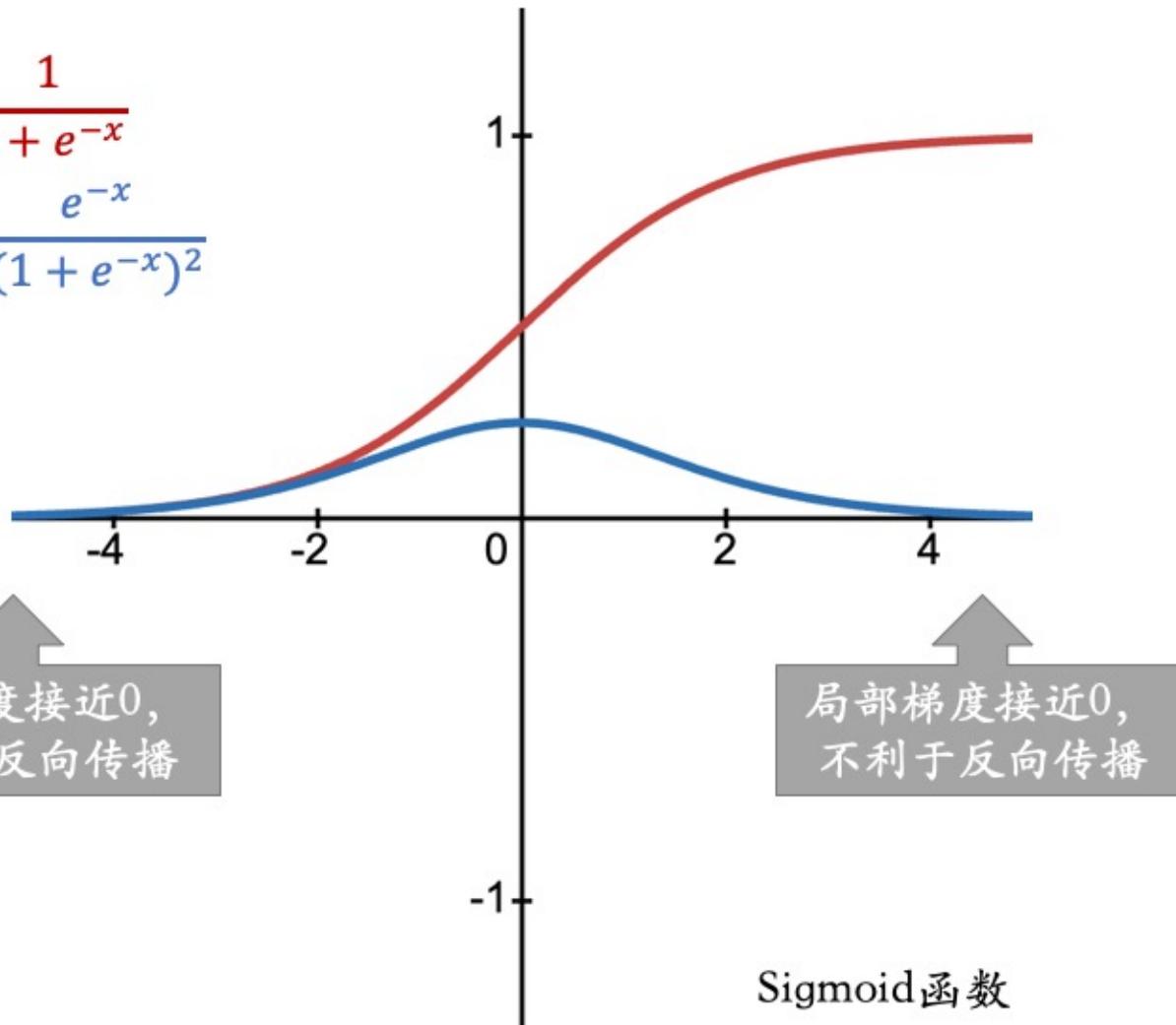
$$\frac{df}{dx} = \frac{df}{dh} \times \frac{dh}{dz} \times \frac{dz}{dx} \times \dots$$



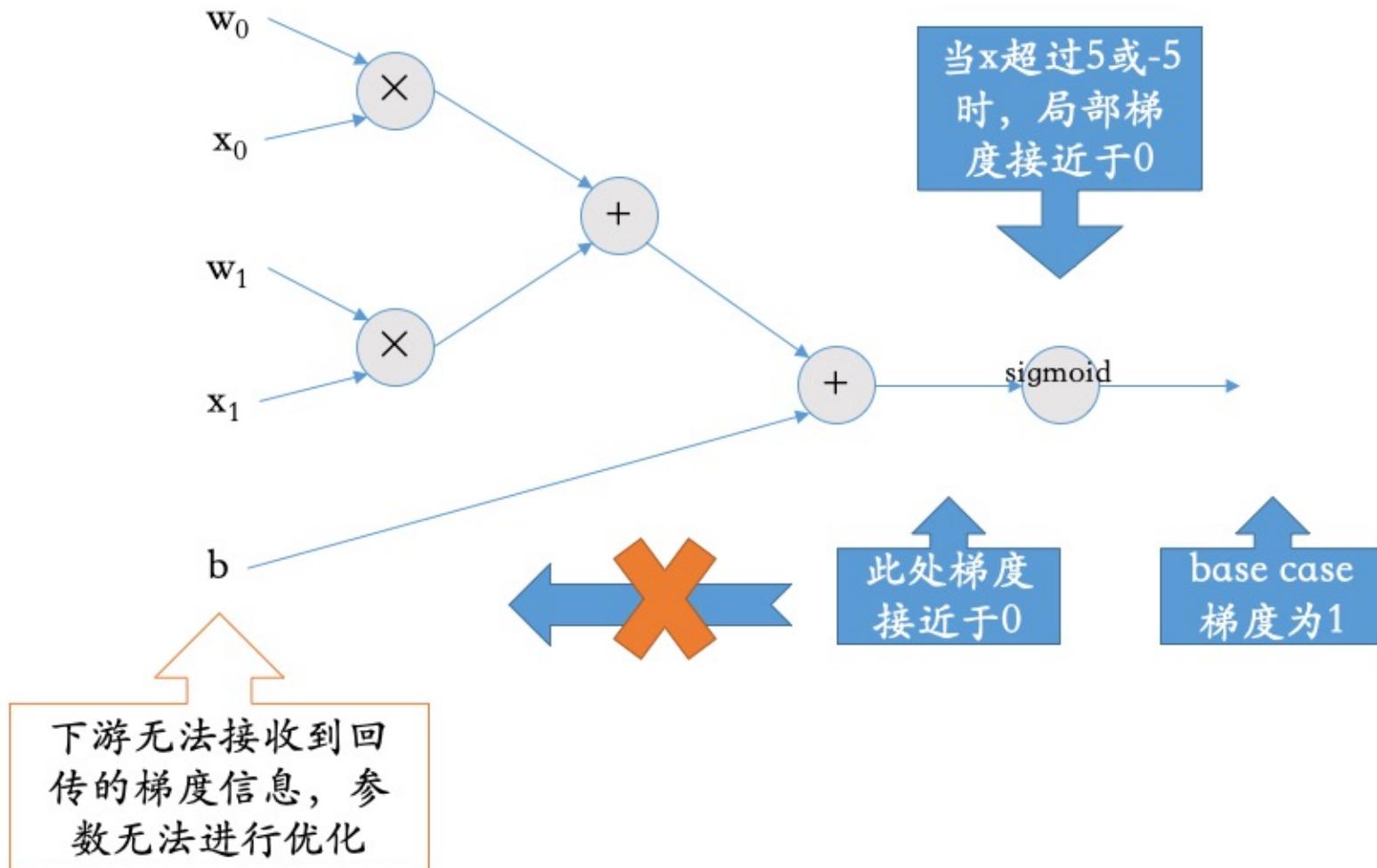
利用链式反应做反向梯度计算，如果其中某些局部梯度较小（甚至为0），叠加后会削弱信号的强度，导致梯度信息难以顺利回传到输入层。

激活函数的选择

$$y = \frac{1}{1 + e^{-x}}$$
$$y' = \frac{e^{-x}}{(1 + e^{-x})^2}$$



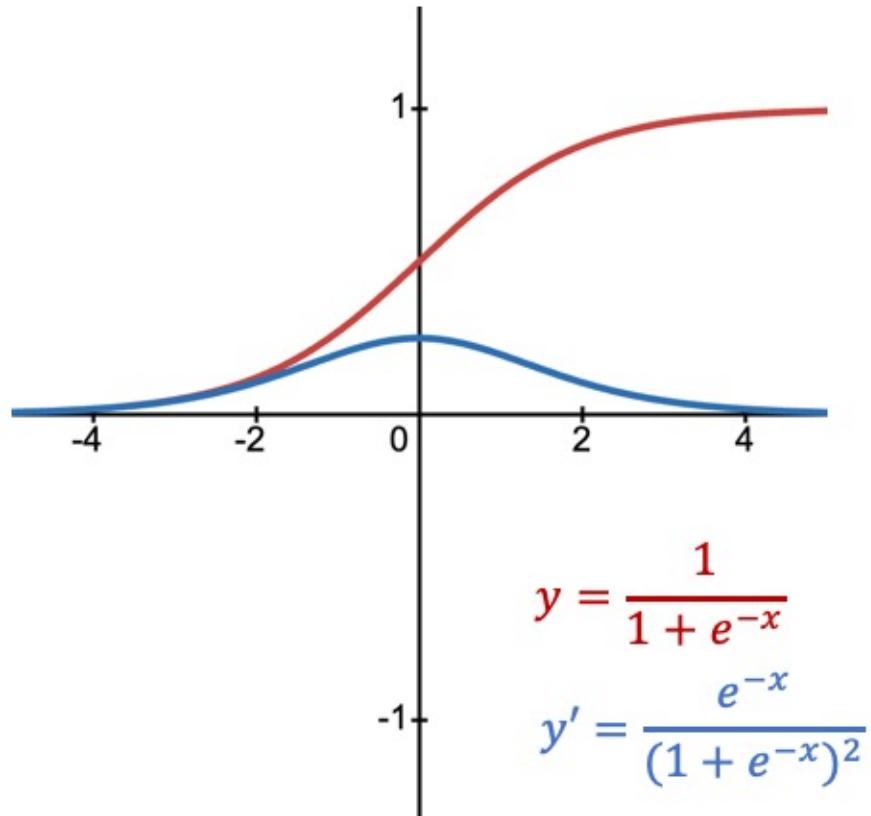
激活函数的选择



激活函数的选择

Sigmoid函数的缺点：

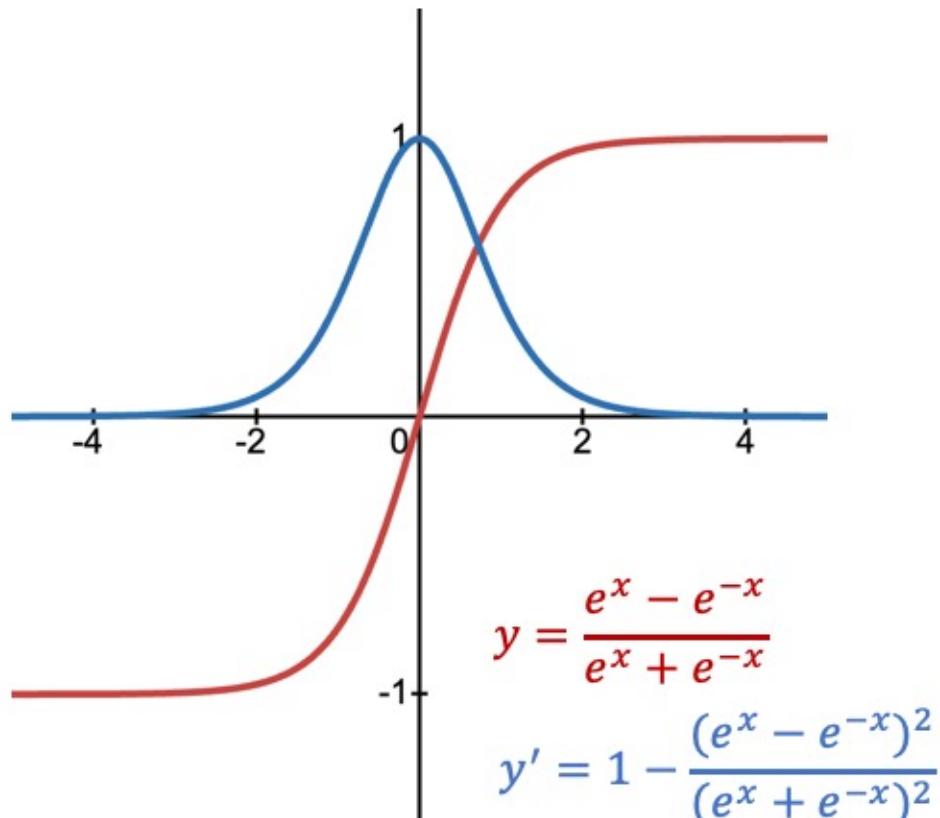
- 当x超出-5到5的范围后，会导致局部梯度趋近于0，出现梯度消失的现象，让模型参数得不到有效的训练。
- 当x=0时，局部梯度达到最大的0.25，但这个数也不够大。链式法则中如果出现多个0.25相乘，也会让梯度变得极小，难以训练模型。



激活函数的选择

Tanh函数的缺点：

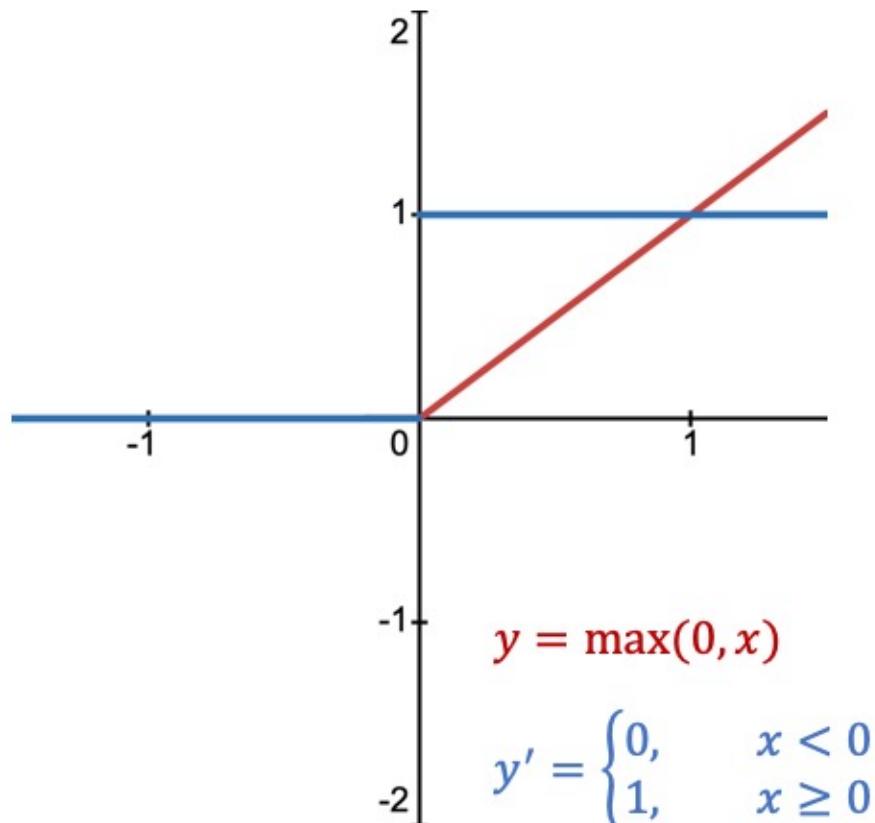
- 和Sigmoid相似，当x超出-3到3的范围后，会导致局部梯度趋近于0，出现梯度消失的现象，让模型参数得不到有效的训练。梯度有效的范围甚至低于Sigmoid。
- 最大的梯度取值为1，这一点优于Sigmoid。



激活函数的选择

ReLU函数的特点：

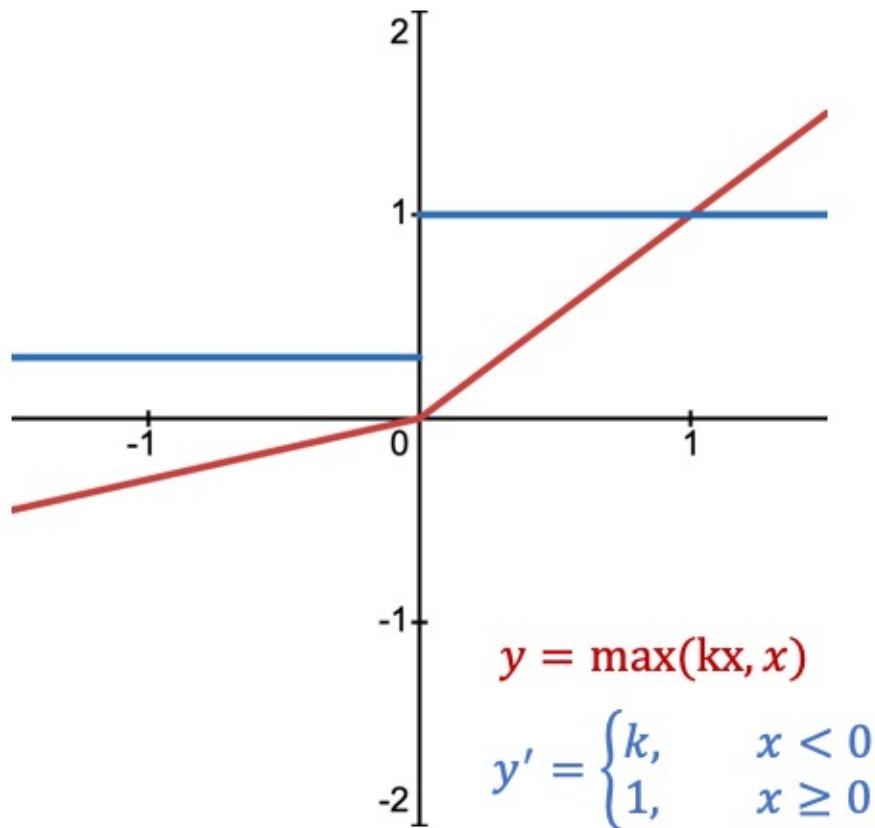
- 当 $x < 0$ 时，局部梯度永远为0，但当 $x \geq 0$ 时，局部梯度永远是1。
- 在 $x \geq 0$ 时，可以保证梯度永远不会消失。因此近年来ReLU成为了最常见的激活函数。



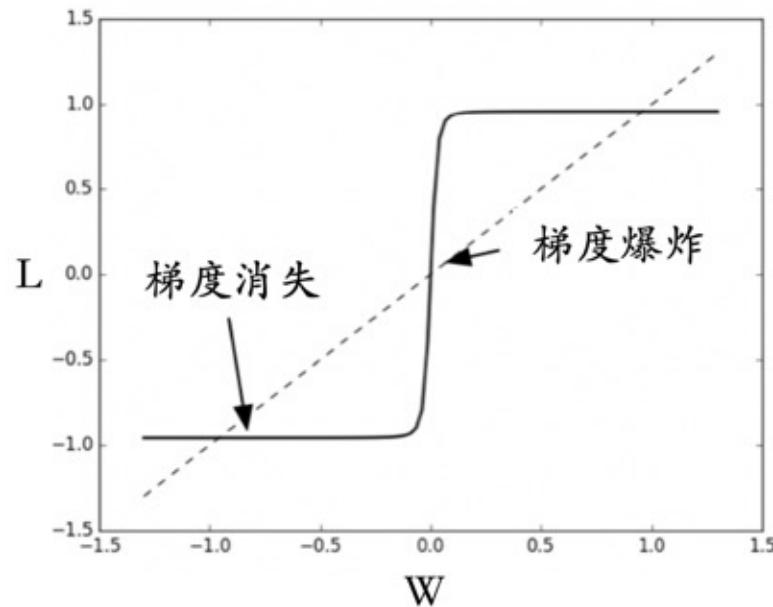
激活函数的选择

Leaky ReLU的特点：

- $x < 0$ 时梯度不是0，弥补了ReLU的弱点。除了 $x = 0$ 处不可导，其余任何取值不影响梯度回传。
- 在 $x \geq 0$ 时，梯度永远为1。当 $x < 0$ 时，梯度永远是k，而k通常是介于0和1之间的常数。



梯度爆炸

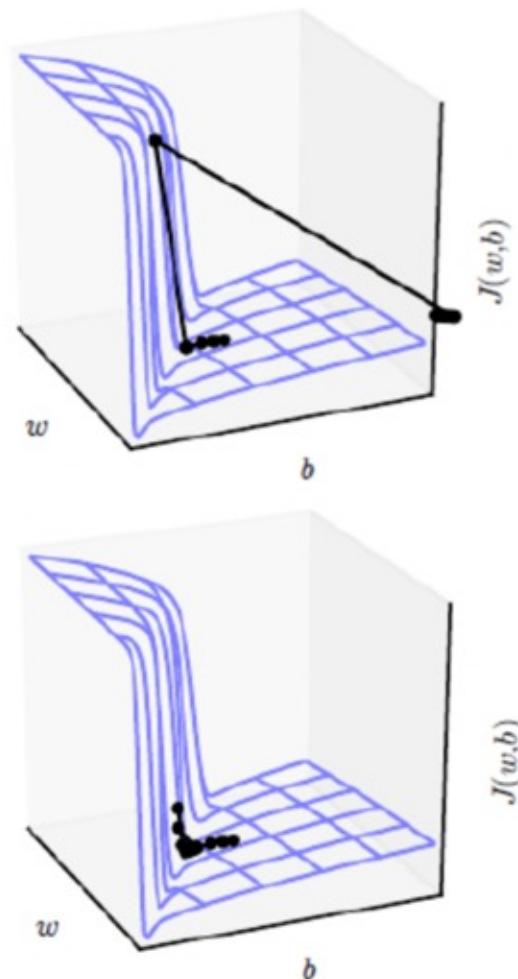


在神经网络的反向传播过程中，梯度值变得非常大，甚至呈指数级增长的现象。

- 成因：网络结构太复杂、激活函数不合适、权值初始化或优化算法不合适。
- 影响：可能导致权值溢出、计算错误，使得训练失败。

梯度裁剪

- 方案一：确定梯度的最大值，如果梯度超过了这个值，就直接裁剪至最大值。
- 方案二：将所有参数的梯度组成向量，计算其L2范数（向量各元素的平方和的平方根），如果L2范数大于设定的最大值，则根据L2范数与最大范数的比值，对梯度进行缩放。



梯度下降的问题

全样本梯度下降（标准/批量下降）

需要让所有样本参与梯度更新的运算。



随机下降

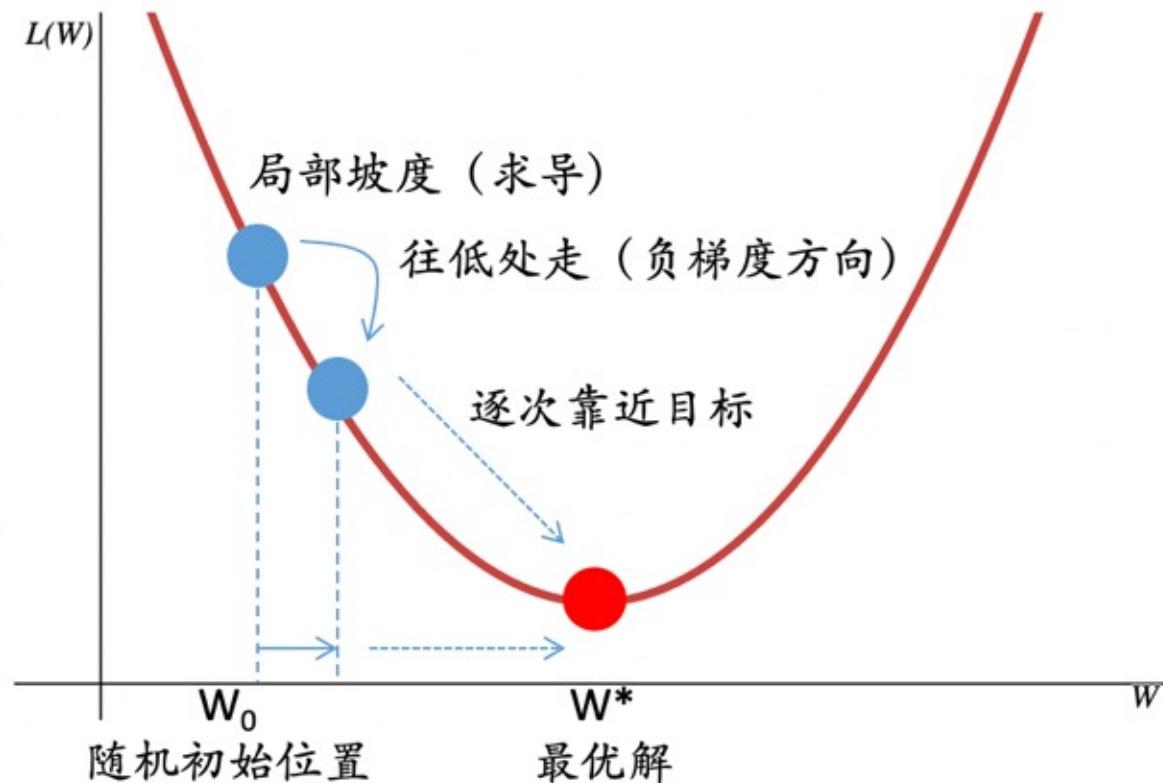
从数据集里随机采样一个样本，来代表全部数据集，进行梯度更新。



小批量下降

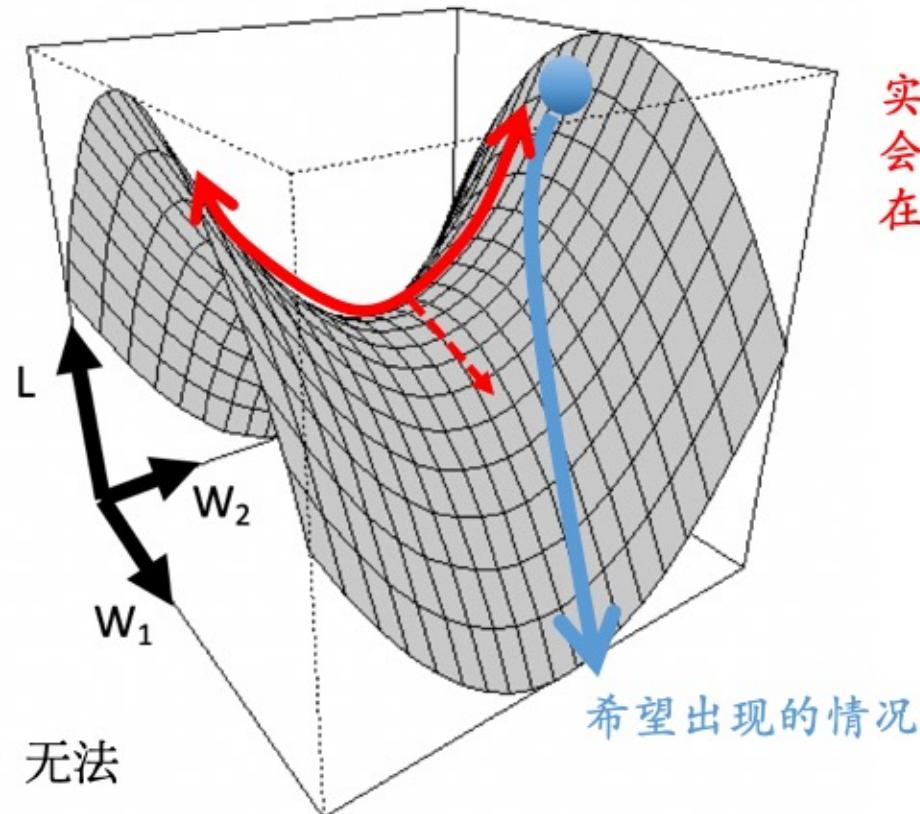
把样本分批运行，每次用其中一批代表全部数据。

考虑1个模型参数（比如权值）



考虑2个模型参数

Saddle 马鞍形

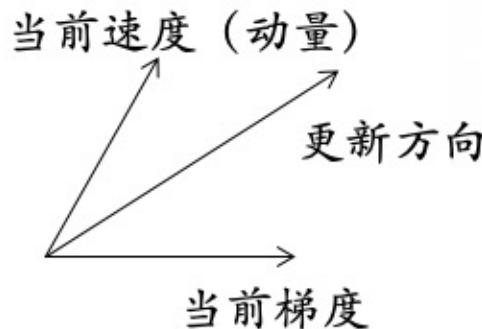
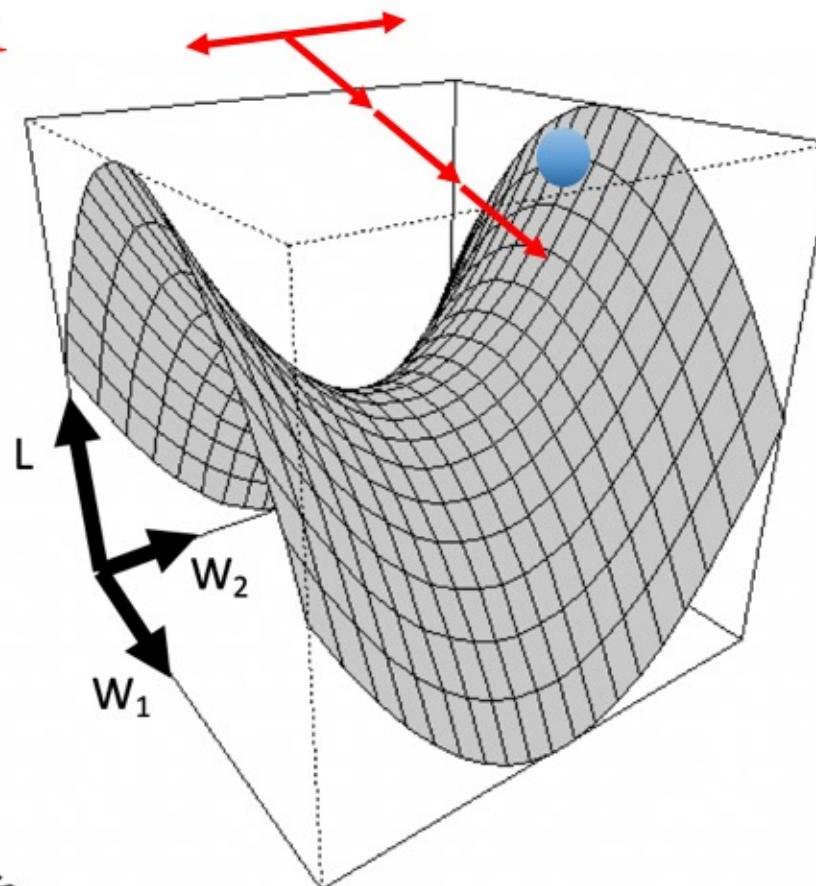


仅增加学习率（步长）无法快速收敛。

深度学习的优化策略

动量法 Momentum

W_2 方向来回震荡，历史动量会互相抵消，减少该方向的学习速度。



W_1 方向动量会逐渐积累，增加该方向的更新速度。

累加历史梯度信息来更新梯度，模拟物理学中的动量积累，迫使更新方向的改变。

动量法 Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

用当前梯度来更新权值。

SGD+Momentum

$$\begin{aligned} v_{t+1} &= \rho v_t + \nabla f(x_t) \\ x_{t+1} &= x_t - \alpha v_{t+1} \end{aligned}$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

用“速度”（累加了历史梯度值的“动量”）来更新权值，而非用当前梯度。

动量法 Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
    dw = compute_gradient(w)
    w -= learning_rate * dw
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

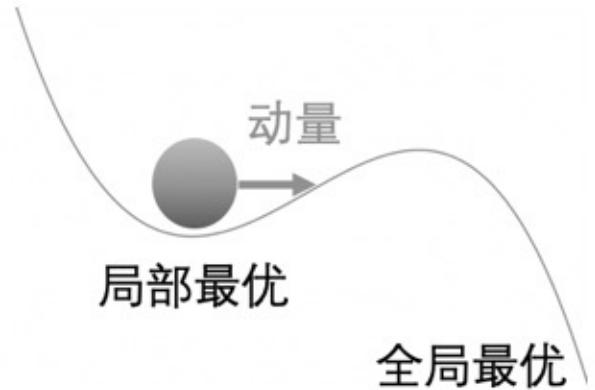
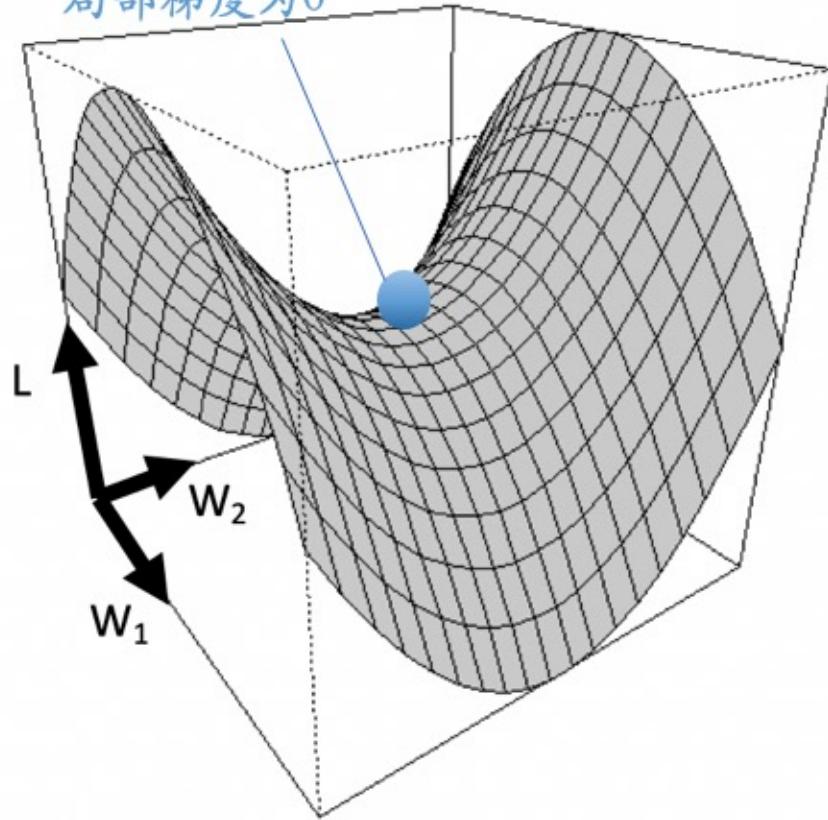
```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

历史梯度信息的衰减，模拟“摩擦力”。
如果它为0，则没有使用动量法；
如果它为1，则模型在达到最优后无法停下。
常见设置为0.9。

动量法 Momentum

鞍点 (saddle point)

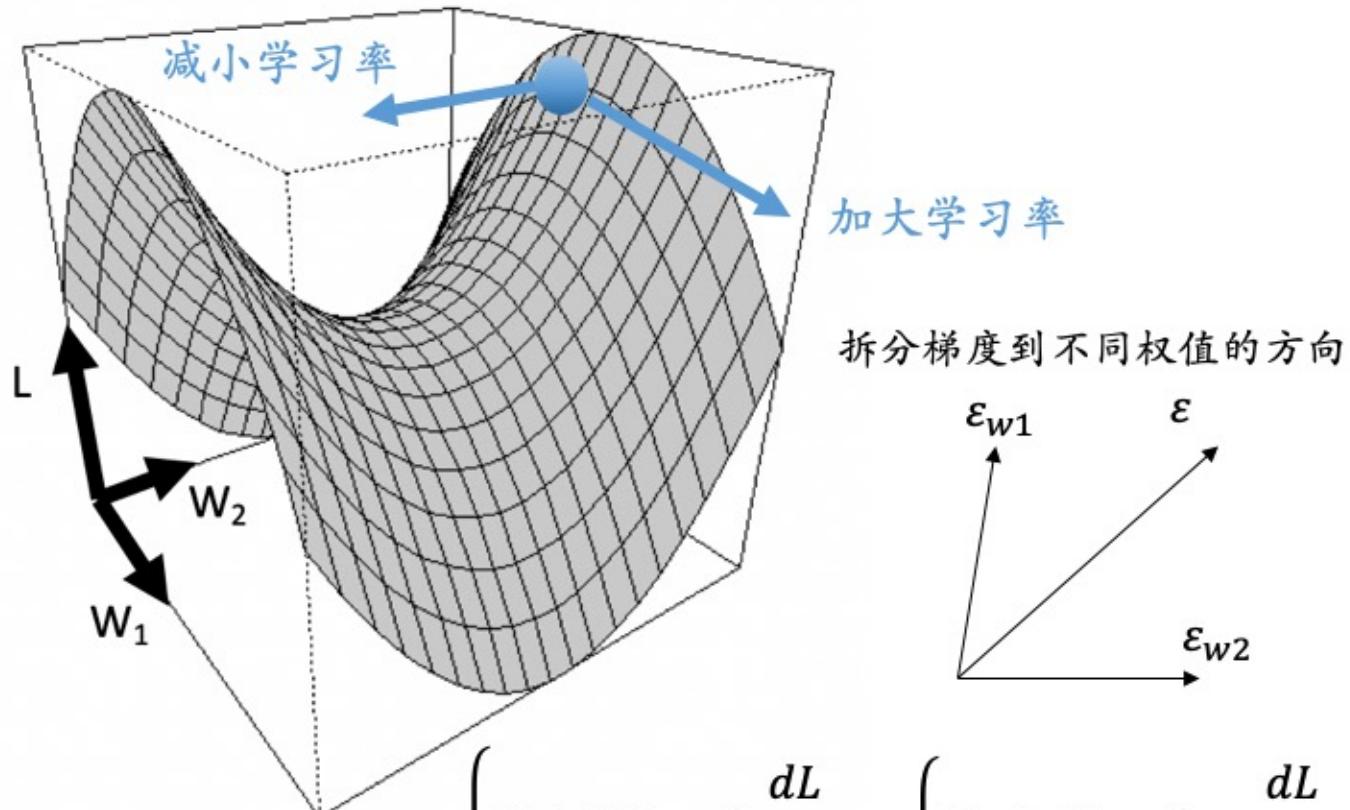
局部梯度为0



动量法的其他用处：

帮助模型“突破”高维空间里的局部最优点或鞍点

自适应梯度 AdaGrad



如何识别振荡方向?

历史梯度的平方值（或绝对值）
较大的方向为振荡方向，反之
为平缓方向。

$$\begin{cases} w_1 = w_1 - \varepsilon \frac{dL}{dw_1} \\ w_2 = w_2 - \varepsilon \frac{dL}{dw_2} \end{cases} \rightarrow \begin{cases} w_1 = w_1 - \varepsilon_{w1} \frac{dL}{dw_1} \\ w_2 = w_2 - \varepsilon_{w2} \frac{dL}{dw_2} \end{cases}$$

自适应梯度 AdaGrad

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared += dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

累加历史梯度的平方值

用历史梯度的平方值（开方后）
影响当前方向的学习率。
历史梯度的平方值越大，
该方向的学习率越小。

防止分母为0

潜在问题：

如果某方向的grad_squared累积到特别大的值，在训练后期会失去调节作用，导致该方向学习过于缓慢。

解决方案：

使用RMSProp法，对AdaGrad进行改进。

均方根传递 RMSProp

为累加的历史梯度平方施加衰减率decay_rate进行抑制。
距离当前轮次越远的历史梯度，对grad_squared的影响越小。

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



衰减率控制历史梯度平方值的累加轮次，取值范围在0和1之间。
如果衰减率为1，则不考虑梯度的历史累加，等效于SGD。
如果衰减率为0，则考虑所有的梯度历史累加，等效于AdaGrad。

是否能把动量法和AdaGrad/RMSProp算法的优点结合起来？

答案：能，用Adam算法！

Adam

Adam算法

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1): # Start at t = 1
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

动量法

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```

体现动量法思路的部分

moment1 ~ v 累加动量（速度）
beta1 ~ rho 摩擦力（动量衰减）

Adam

体现AdaGrad/RMSProp思路的部分

Adam算法

moment2 ~ grad_squared 累加历史平方梯度
beta2 ~ decay_rate 衰减率

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

RMSProp

```
grad_squared = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
    w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Adam

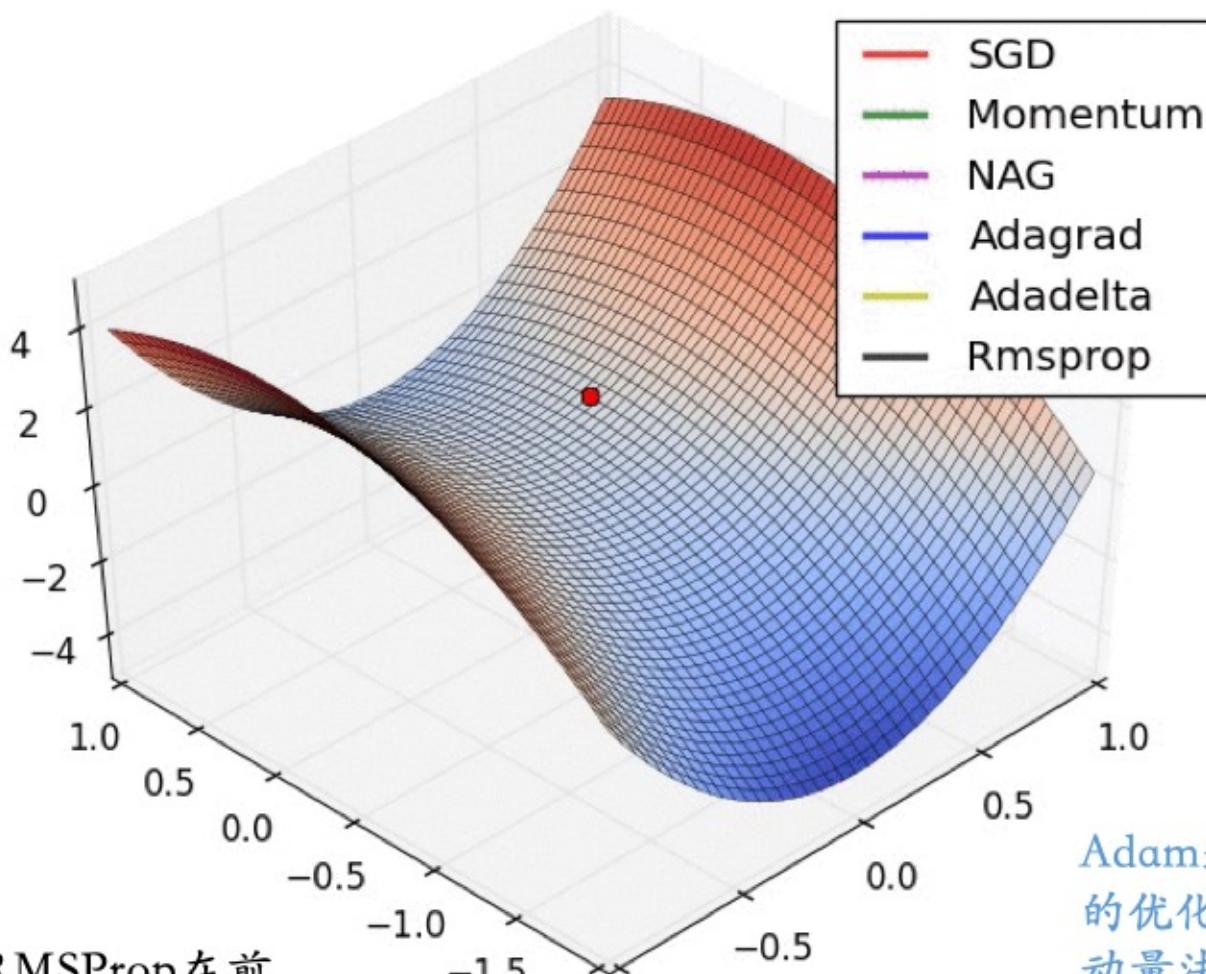
Adam算法

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
    dw = compute_gradient(w)
    moment1 = beta1 * moment1 + (1 - beta1) * dw
    moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
    moment1_unbias = moment1 / (1 - beta1 ** t)          修正偏差
    moment2_unbias = moment2 / (1 - beta2 ** t)
    w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

冷启动问题：如果 $\beta_1 = 0.9$ 、 $\beta_2 = 0.99$ （常见设置），当 $t=1$ 时， dw 的更改幅度极小，模型的学习率极低。

修正偏差：在启动时，提升模型的学习率。随着 t 的增加，修正幅度越来越小，因此它是针对冷启动进行修正。

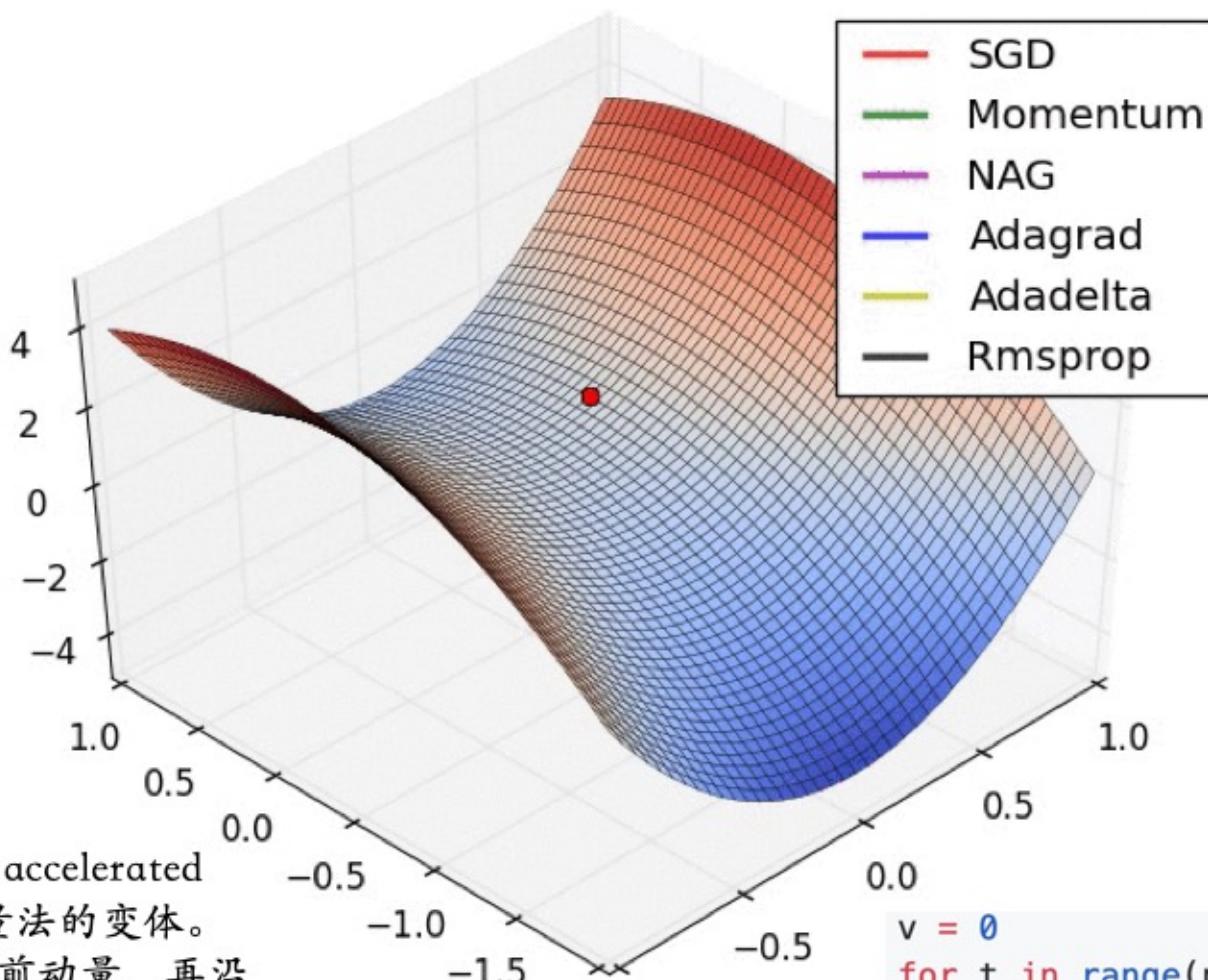
动画演示效果对比



AdaGrad和RMSProp在前期优于动量法，后期被动量法反超。它们都优于SGD（训练失败）。

Adam通常被用作默认的优化算法。
动量法在某些情况下可能会达到比Adam更高的效率，但条件十分苛刻，需要反复调参。

动画演示效果对比

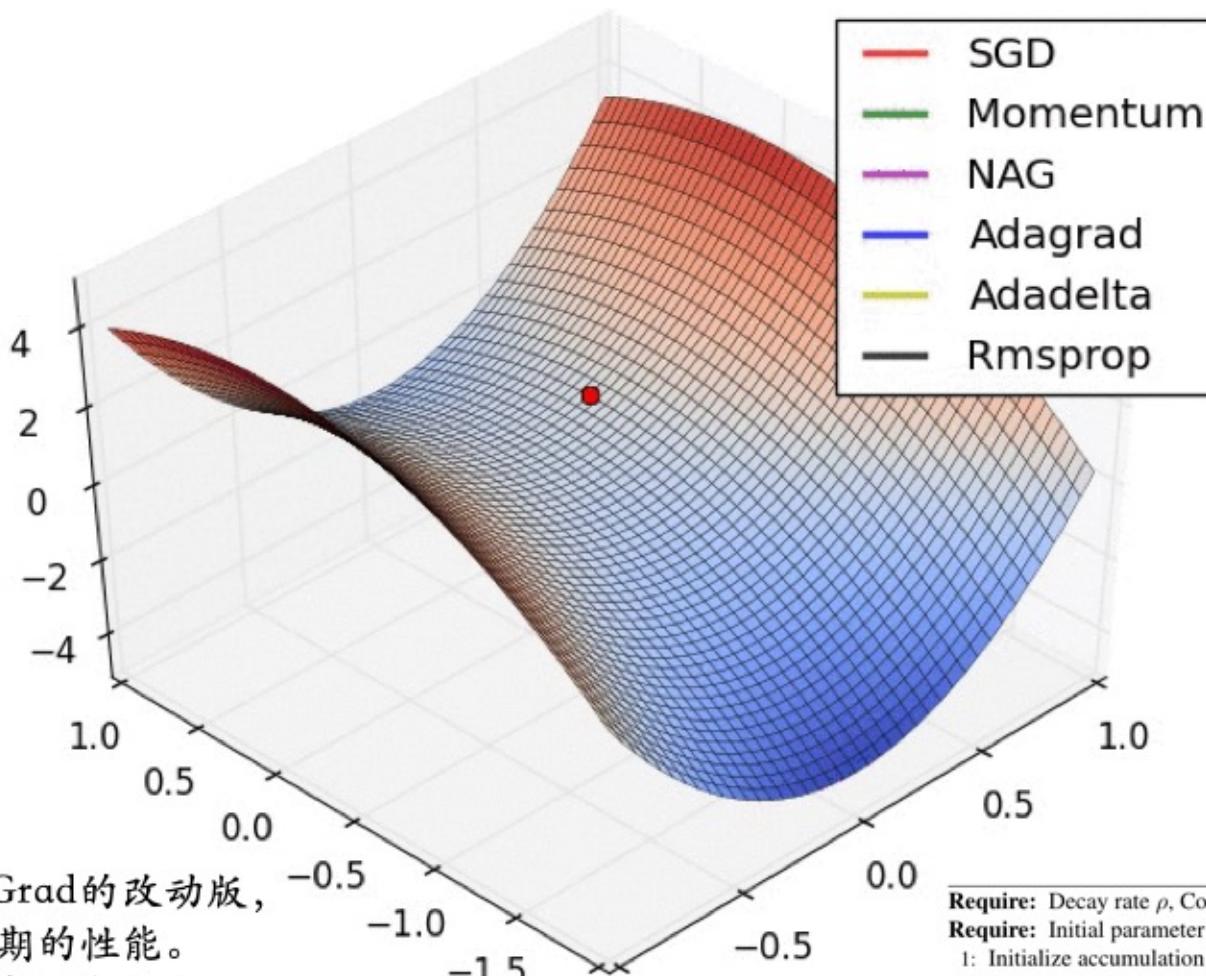


NAG (Nesterov accelerated gradient) : 动量法的变体。

动量法先计算当前动量，再沿动量方向下降； NAG先沿之前的动量方向下降，再判断方向是否正确，并做出修正。

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    old_v = v
    v = rho * v - learning_rate * dw
    w -= rho * old_v - (1 + rho) * v
```

动画演示效果对比



AdaDelta: AdaGrad的改动版，
旨在提升训练后期的性能。

AdaGrad累积所有历史梯度的
平方，AdaDelta添加了固定大
小的滑动窗口，只累加窗口内
的历史梯度。

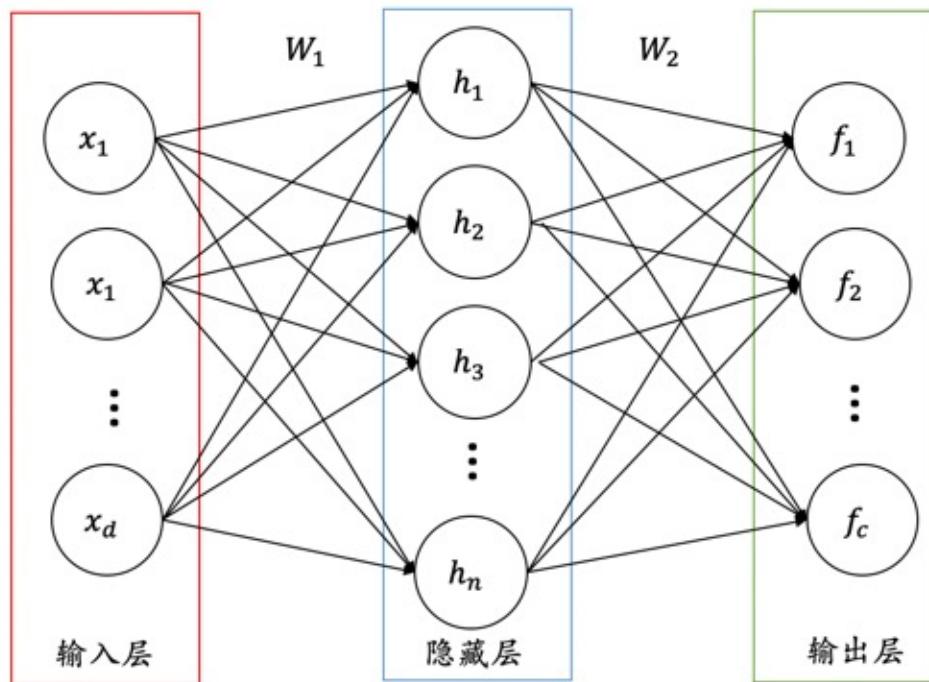
Require: Decay rate ρ , Constant ϵ

Require: Initial parameter x_1

- 1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
- 2: **for** $t = 1 : T$ **do** %% Loop over # of updates
- 3: Compute Gradient: g_t
- 4: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho)g_t^2$
- 5: Compute Update: $\Delta x_t = -\frac{\sqrt{E[\Delta x^2]_{t-1}}}{\sqrt{E[g^2]_t}} g_t$
- 6: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1 - \rho)\Delta x_t^2$
- 7: Apply Update: $x_{t+1} = x_t + \Delta x_t$
- 8: **end for**

训练全连接神经网络

权值初始化

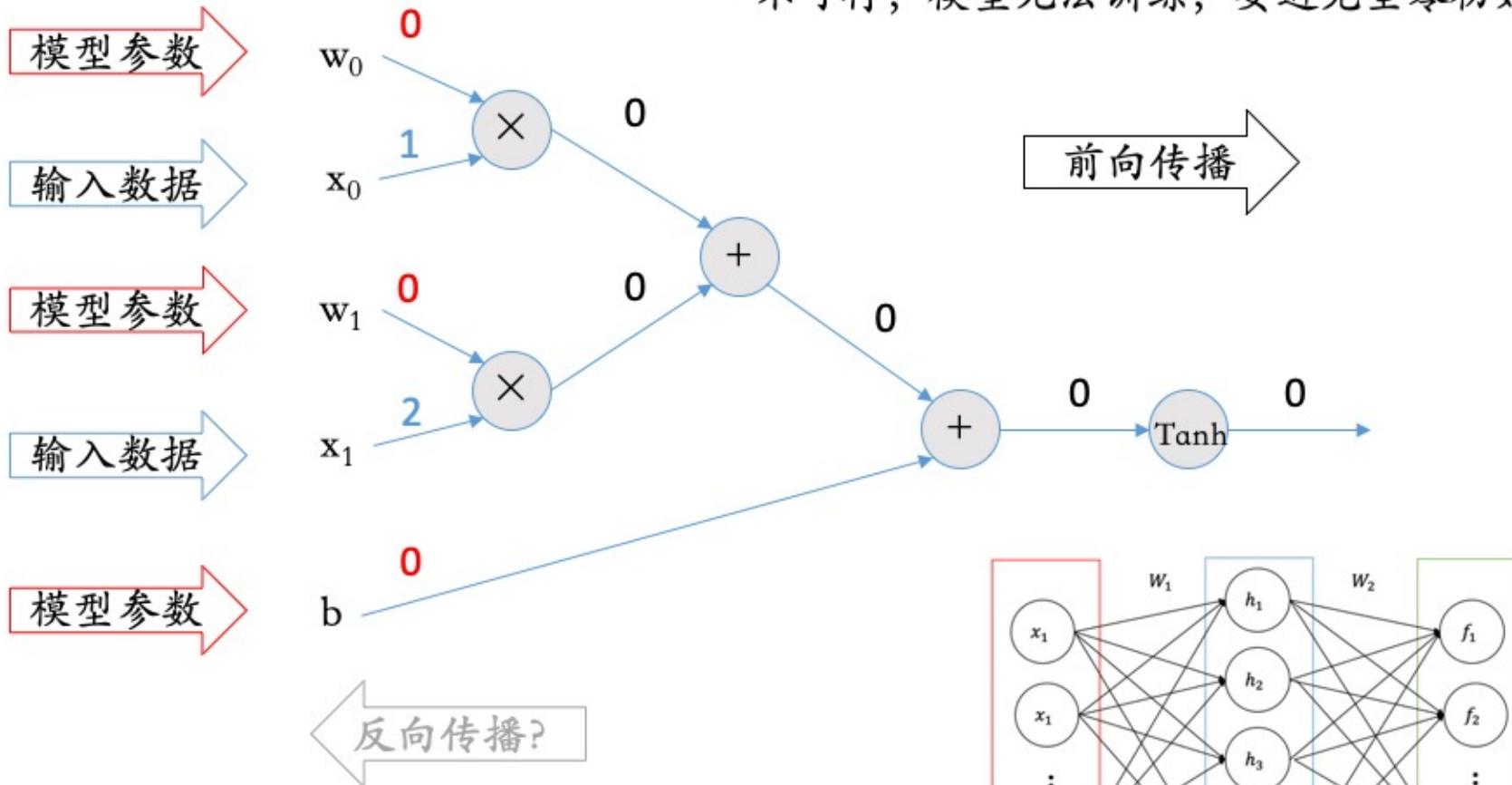


权值初始化：定义 W_1 和 W_2 的初始值
注意：和设置超参数不是一回事

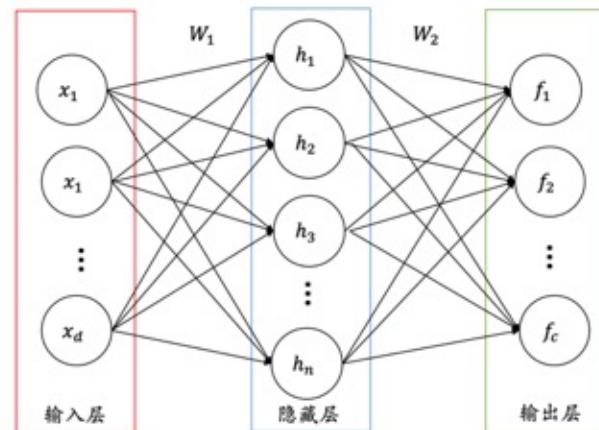
全零初始化

初始化时，把所有模型参数设置为0，是否可行？

- 不可行，模型无法训练，要避免全零初始化。



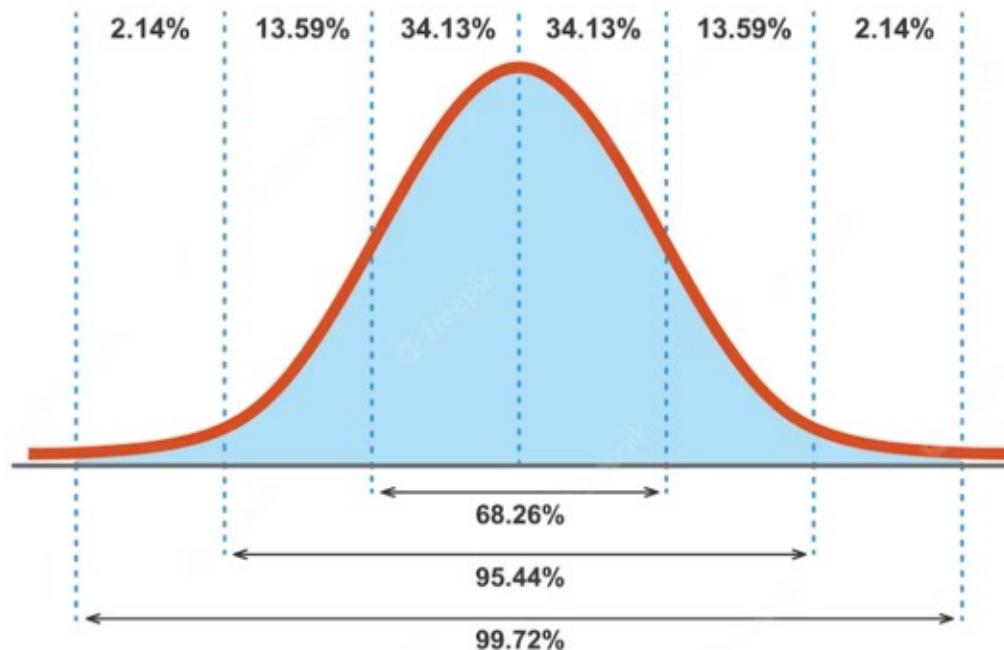
无论输入的数据是什么，
回传的梯度全都一样。
每个神经元学到的东西都一样。



随机初始权值

- 定义一个高斯分布（设定均值、标准差）
- 根据这个分布，对初始权值进行采样

```
W = 0.01 * np.random.randn(Din, Dout)
```

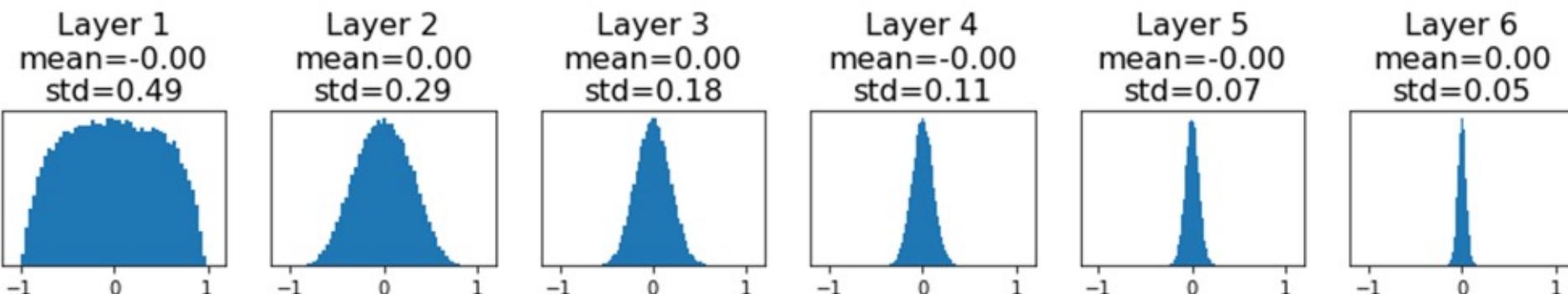


随机初始权值

正态分布均值为0、标准差为0.01

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

如果有更多层次：权值向0
收缩，信息无法传递



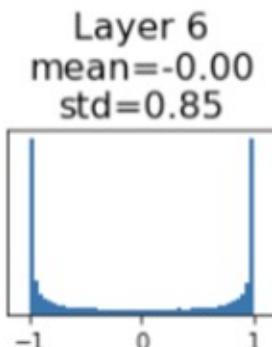
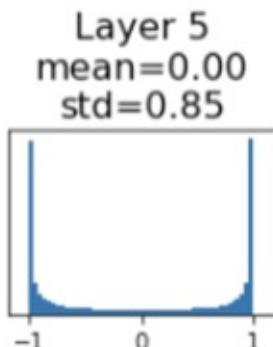
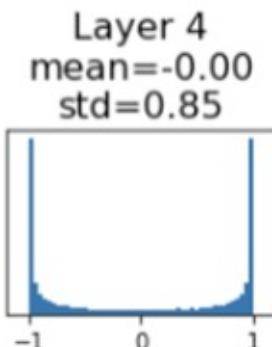
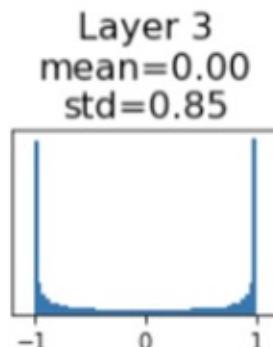
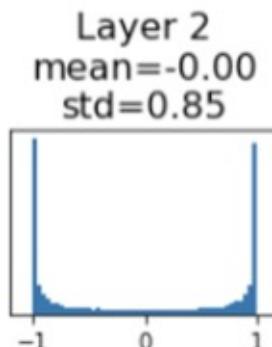
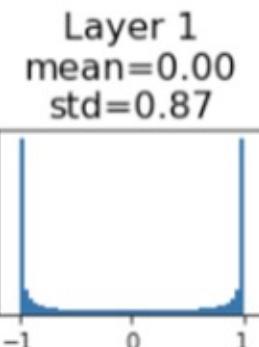
深层网络无法学习！

随机初始权值

正态分布均值为0、标准差为0.05

```
dims = [4096] * 7      Increase std of initial weights  
hs = []                  from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

模型处于全饱和状态，局部梯度为0，仍然无法训练



泽维尔初始化 Xavier



Xavier Glorot

```
dims = [4096] * 7           "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

各层的激活值和局部梯度的
方差在传播时尽量保持一致

Layer 1
mean=-0.00
std=0.63

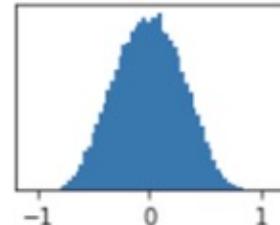
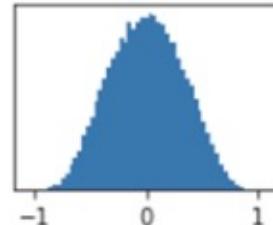
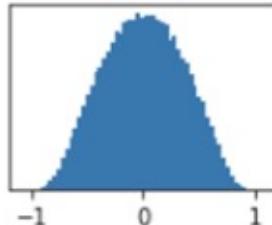
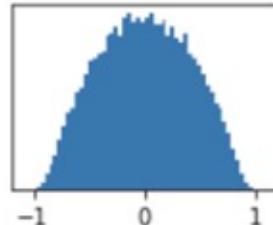
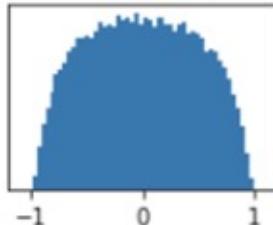
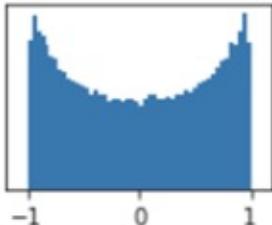
Layer 2
mean=-0.00
std=0.49

Layer 3
mean=0.00
std=0.41

Layer 4
mean=0.00
std=0.36

Layer 5
mean=0.00
std=0.32

Layer 6
mean=-0.00
std=0.30



$$y = \varphi(w_1x_1 + \dots + w_nx_n)$$

泽维尔初始化 Xavier

什么极端情况下，才能让 $Var(x_i) = Var(y_i)$?

$$Var(y) = \sum_{i=1}^n Var(w_i x_i)$$

$$Var(y_i) = nVar(w_i x_i)$$

$$= n(Ew_i^2Ex_i^2 - (Ew_i)^2(Ex_i)^2)$$

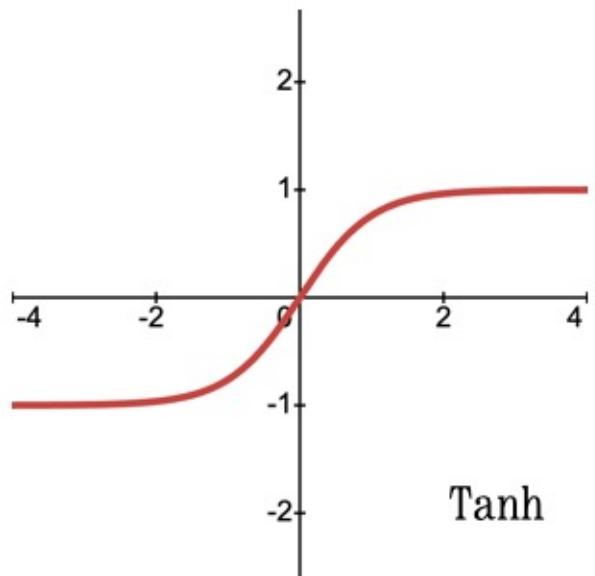
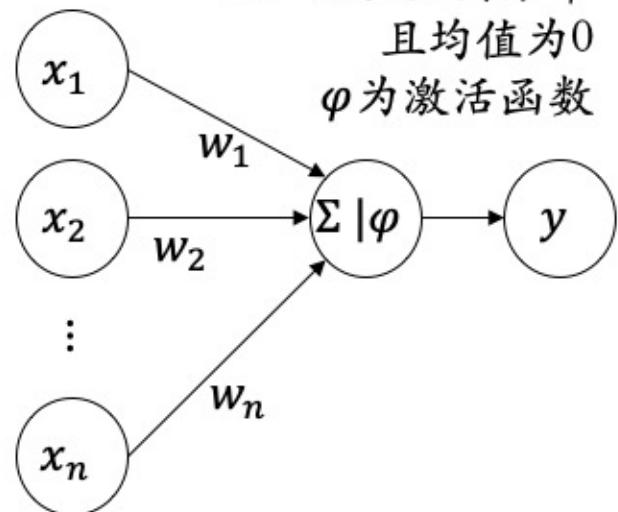
$$= nVar(w_i)Var(x_i)$$

如果 $Var(w_i) = \frac{1}{n}$ 则 $Var(y_i) = Var(x_i)$

```

dims = [4096] * 7           "Xavier" initialization:
hs = []                      std = 1/sqrt(Din)
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)

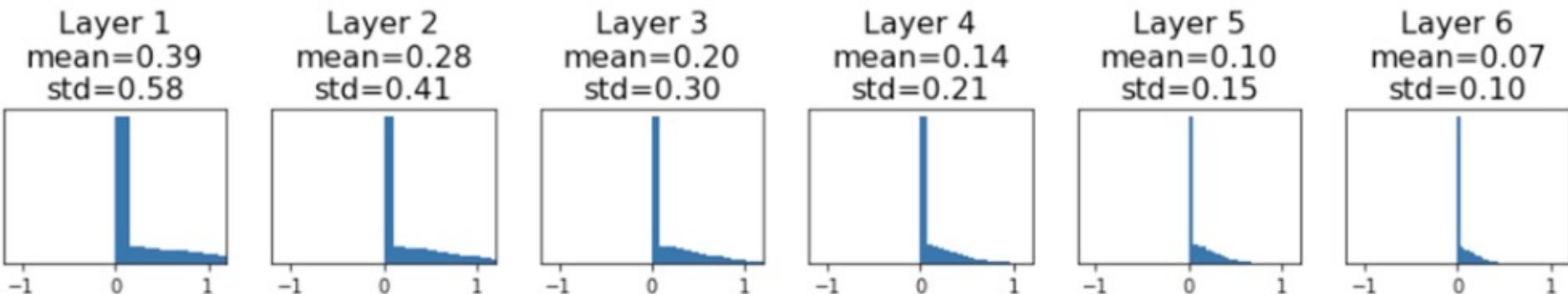
```



如果激活函数是ReLU

```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

权值集中在0附近



MSRA初始化 (Kaiming初始化)

```
dims = [4096] * 7    ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```



何恺明

ReLU: 对于均值为0的高斯分布，有一半的值会被抛弃。

解决方案：将输出y的方差扩大一倍。

Layer 1
mean=0.57
std=0.83

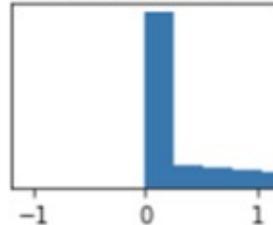
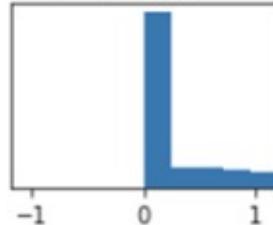
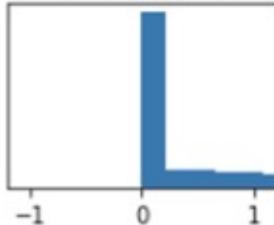
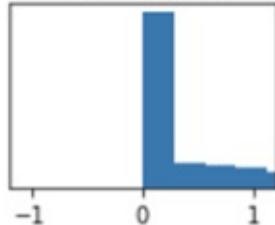
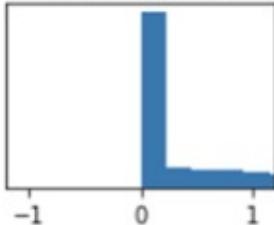
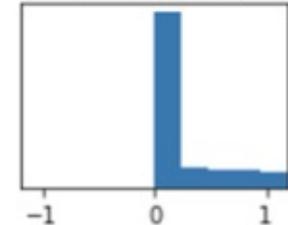
Layer 2
mean=0.57
std=0.83

Layer 3
mean=0.56
std=0.83

Layer 4
mean=0.55
std=0.81

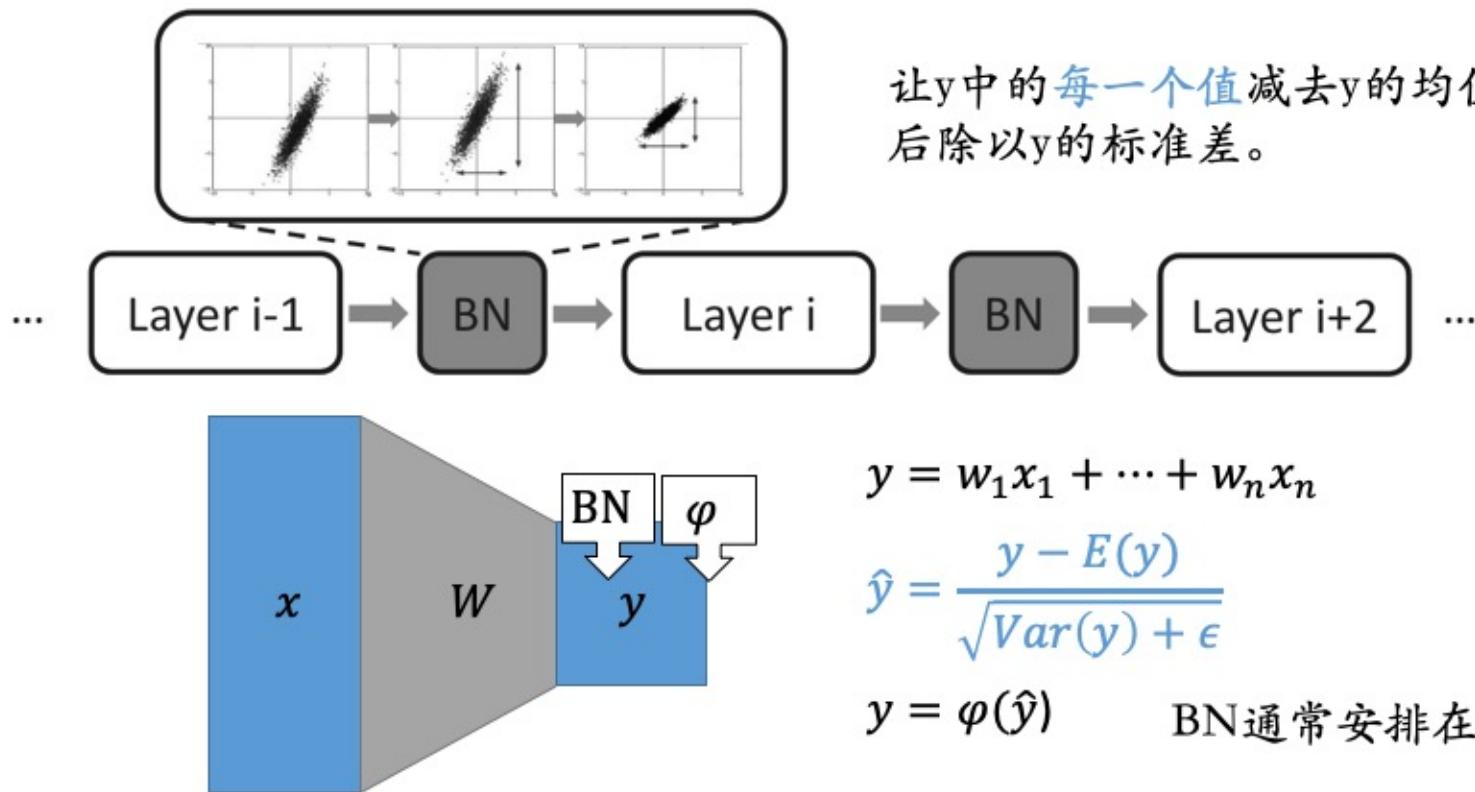
Layer 5
mean=0.55
std=0.81

Layer 6
mean=0.55
std=0.81



批归一化

Batch normalization (BN) : 直接对输出y的分布做调整，让其满足均值为0、方差（标准差）为1的正态分布。

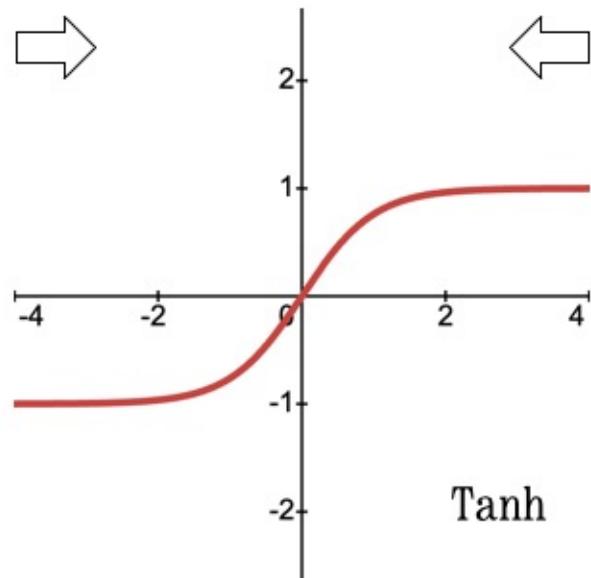


批归一化

BN安排在激活函数前，让激活函数的输入值满足均值为0、方差为1的正态分布：

- 缓解前向运算时的信号过小问题。
- 缓解Sigmoid和Tanh引起的梯度回传不畅问题。
- 避免ReLU出现所有输入值都为负数的极端情况。

让激活函数的输入值向有梯度的区间集中



批归一化

平移缩放：在BN的框架下，对均值和方差（标准差）的取值做出个性化调整，使其更加满足具体的任务需求。

引入模型参数（并非超参数）：

- γ 决定方差（标准差）的大小（对正态分布进行“缩放”）
- β 决定均值的大小（对正态分布进行“平移”）

$$y = w_1x_1 + \dots + w_nx_n$$

$$\hat{y} = \frac{y - E(y)}{\sqrt{Var(y) + \epsilon}}$$

$$\hat{y}' = \gamma\hat{y} + \beta$$

$$y = \varphi(\hat{y}')$$

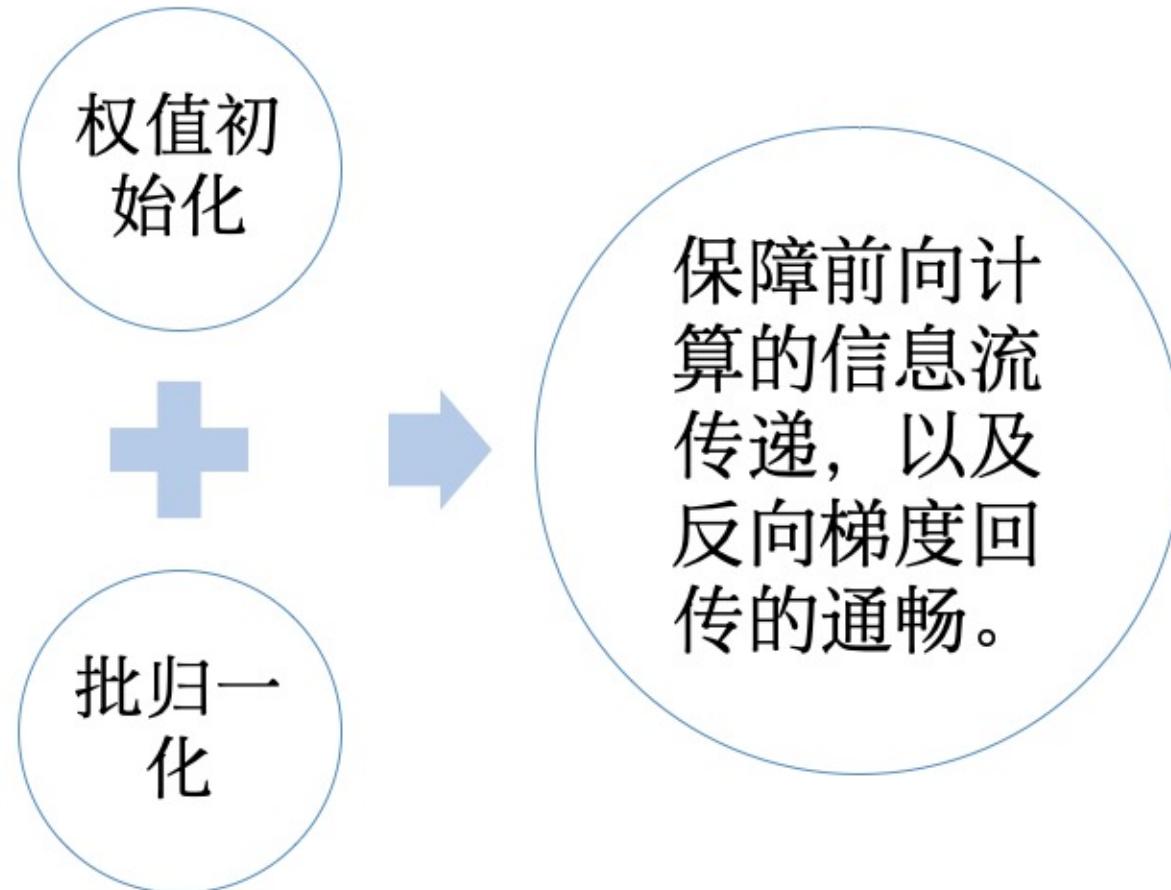
$$\mu = 2 \\ \sigma = 1$$

$$\mu = 2 \\ \sigma = 2$$

$$\mu = -2 \\ \sigma = 1$$

$$\mu = -2 \\ \sigma = 2$$

小结

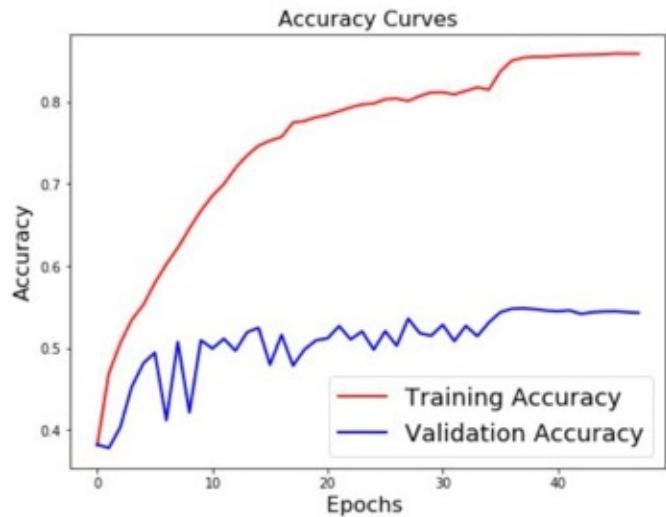
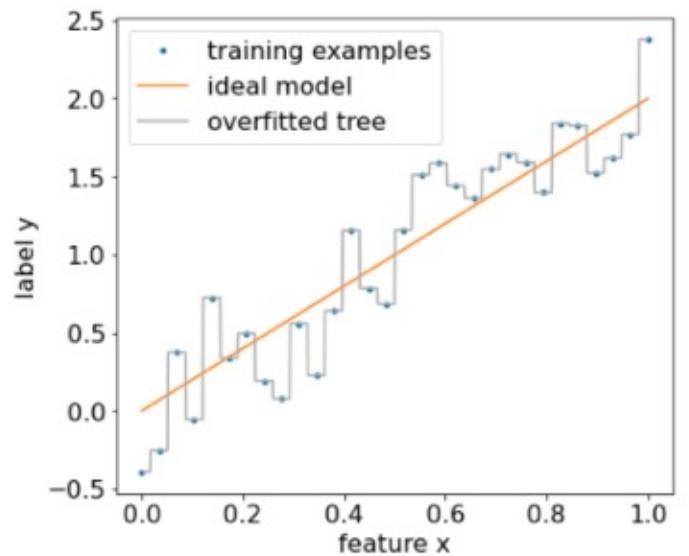


欠拟合和过拟合

过拟合 Overfitting

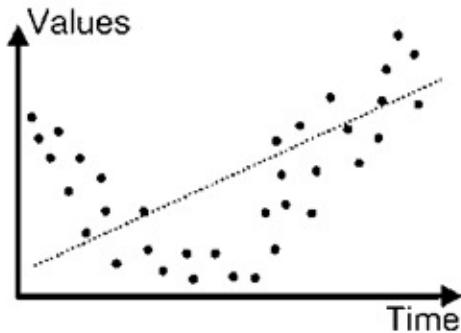
模型为了得到更好的效果，而使假设变得过度复杂和严格。

- 具体来说，一个过度复杂的模型会对训练集的数据进行“记忆”而非“学习”，它会在训练集上表现得非常好，但在未知数据或者测试集上表现得较差。
- 模型过于复杂或者参数过多，导致模型对于训练数据过于敏感，而对于新数据的泛化能力下降。

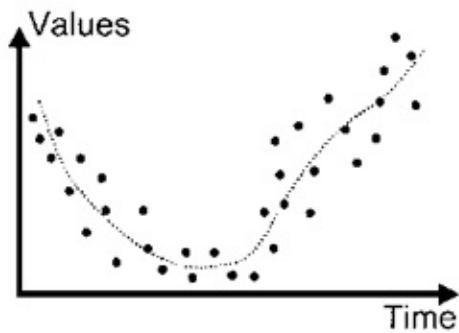


欠拟合 Underfitting

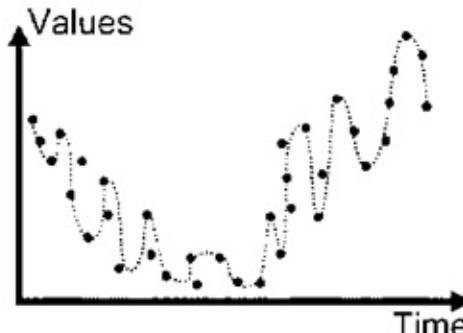
模型未能有效地捕捉到训练数据的特征，导致预测结果准确率低。这通常意味着模型过于简单，无法充分学习数据的复杂性和内在规律。



Underfitted



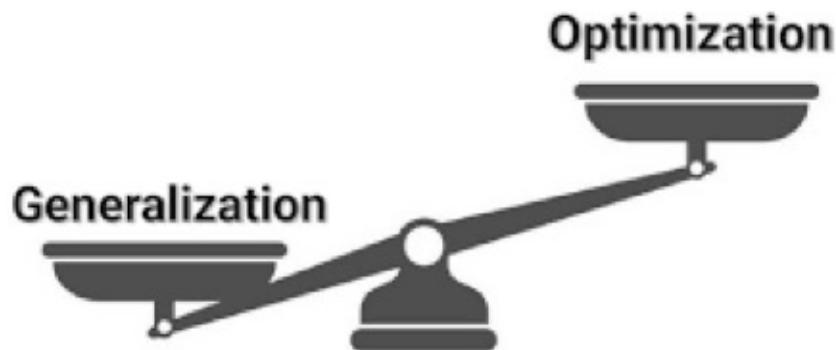
Good Fit/Robust



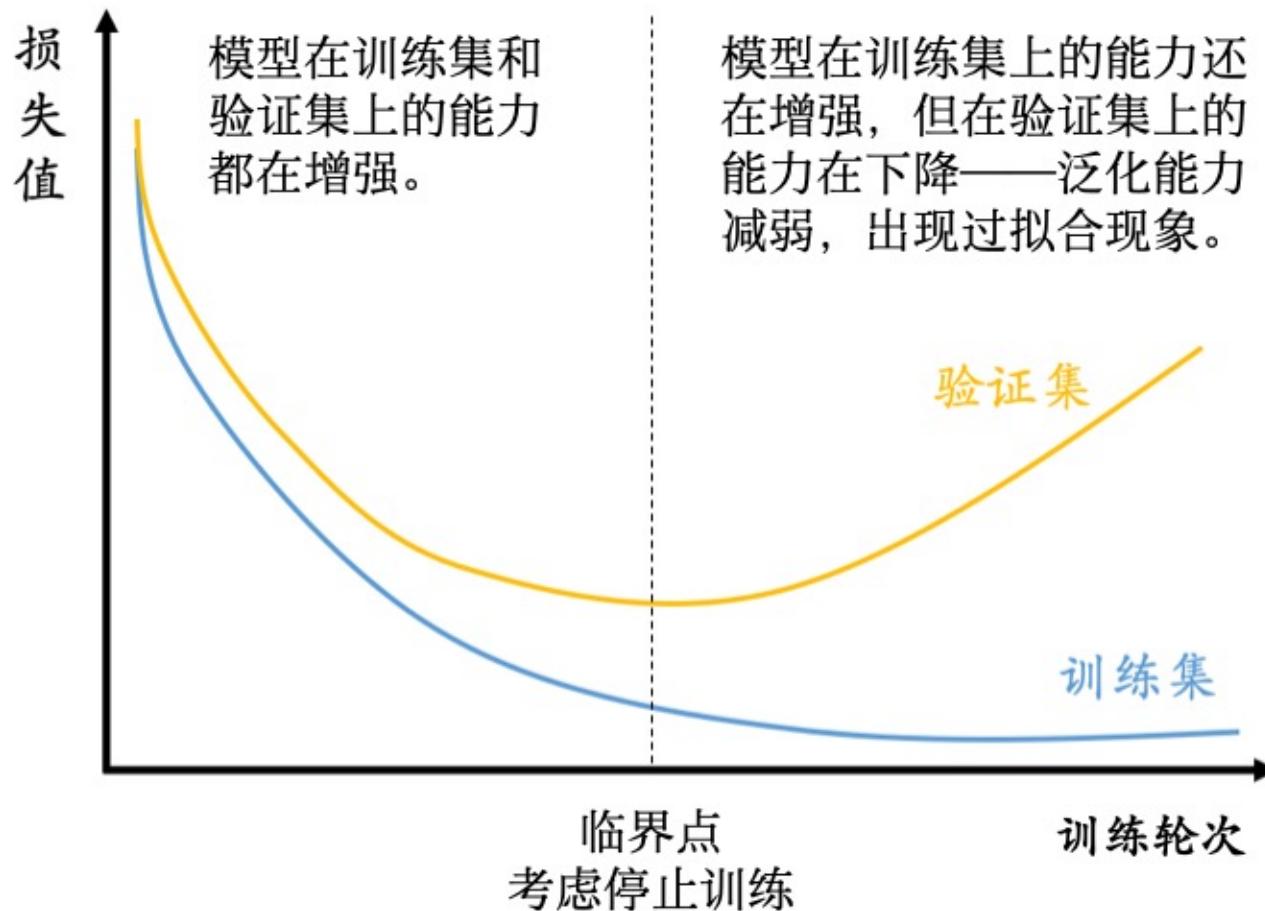
Overfitted

机器学习的根本问题

- 优化（optimization）：让模型能够更好地拟合训练集的样本，让损失值尽量低。
- 泛化（generalization）：让模型在训练集之外的数据集上同样能表现良好。



过拟合和泛化能力

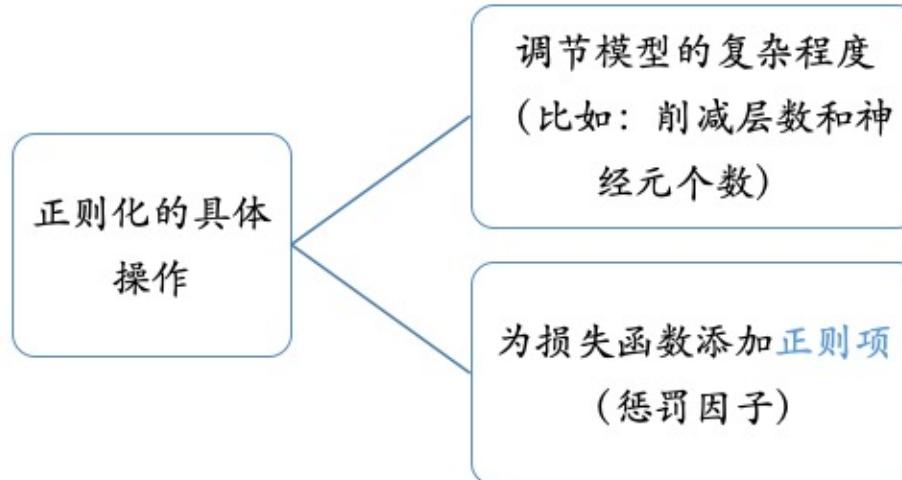


减少过拟合

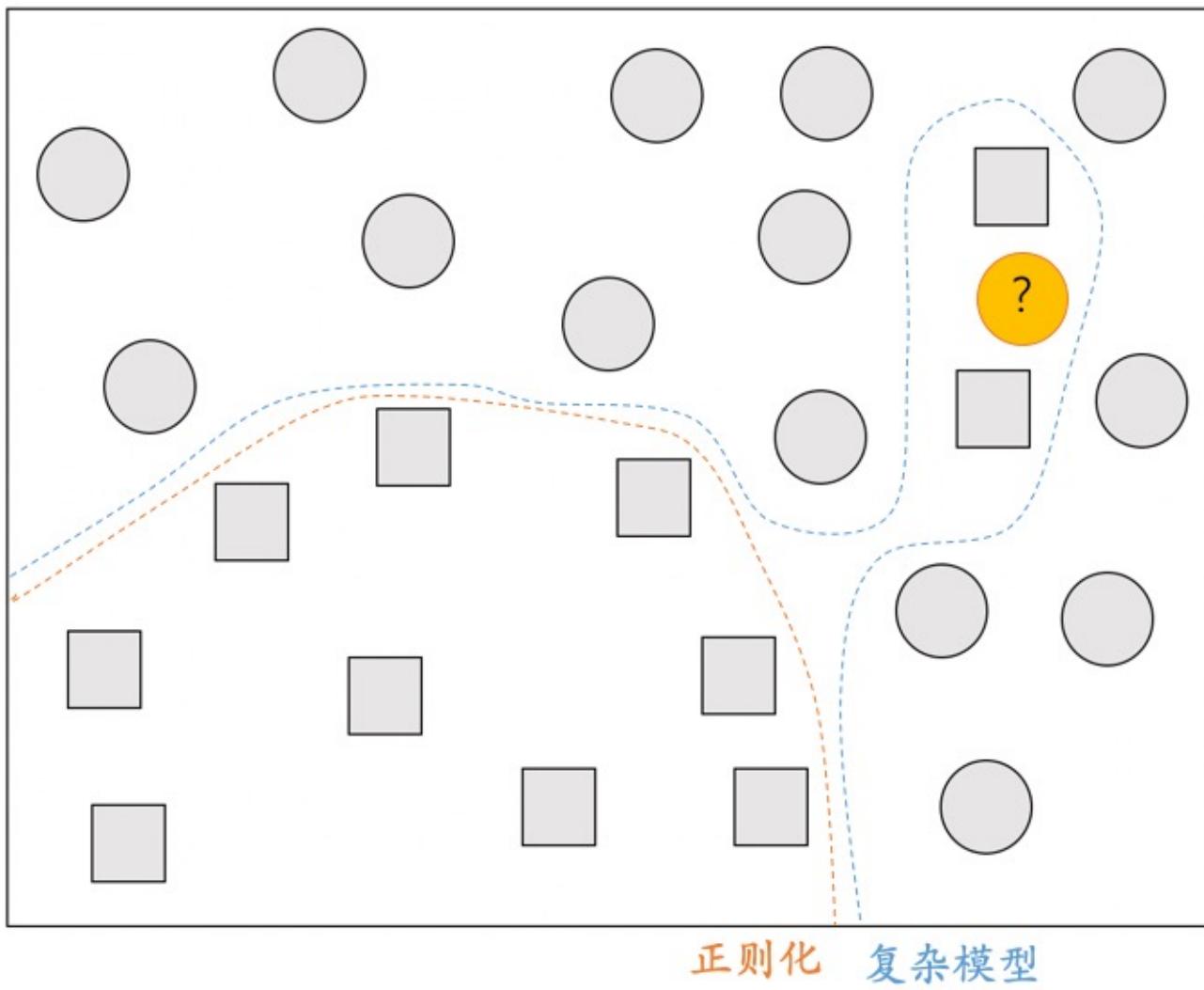
- 方案一：投入更多的训练数据
 - 通常难以实现——成本、技术等限制

减少过拟合

- 方案一：投入更多的训练数据
 - 通常难以实现——成本、技术等限制
- 方案二：对允许模型存储的信息量加以限制
 - 限制模型的“记忆力”，不让它“死记硬背”——技术上可行，即正则化



正则项（非线性任务）



正则项（非线性任务）

让权值分布更平均，削减个别权值的“话语权”，让模型“兼听则明”。



让决策边界更平滑，抓住大的趋势和规律，不要过分关注局部特例，从而抑制过拟合现象。

减少过拟合

- 方案一：投入更多的训练数据
 - 通常难以实现——成本、技术等限制
- 方案二：对允许模型存储的信息量加以限制
 - 限制模型的“记忆力”，不让它“死记硬背”——技术上可行，即正则化
- 方案三：让隐藏层的部分神经元不被激活
 - 缩减模型的有效部分，从而化简模型——技术上可行，即随机失活（dropout）

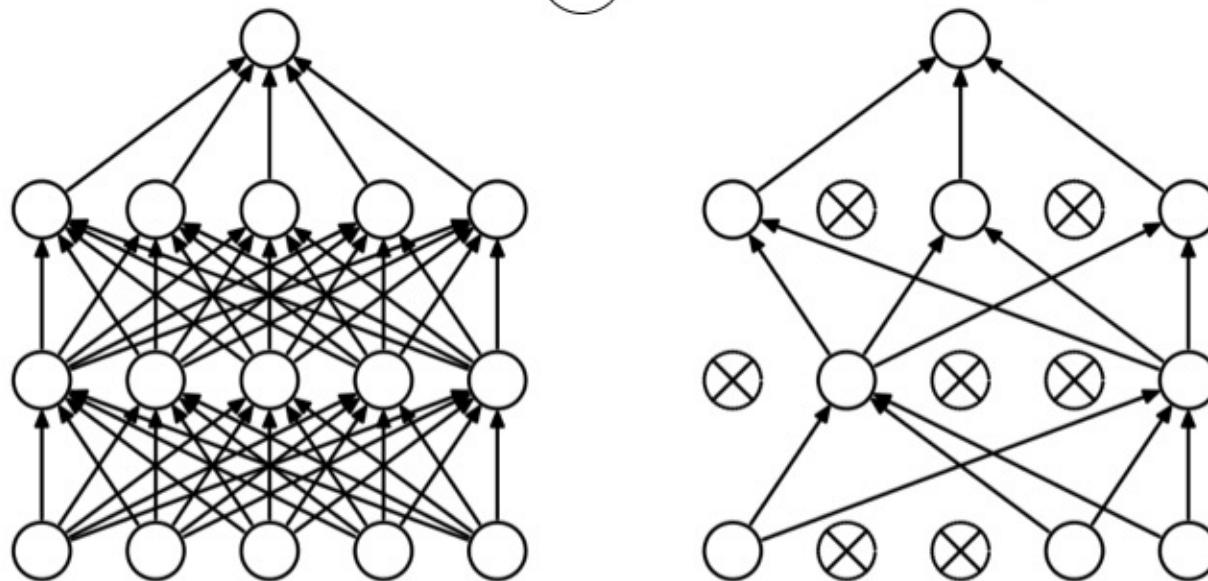
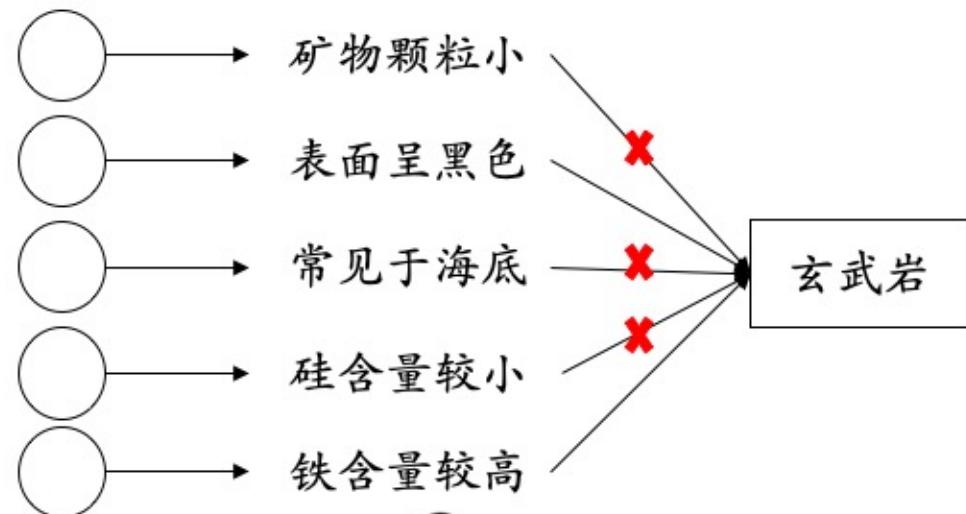
Dropout

让隐藏层的神经元有一定的概率不被激活。

- 操作方法：训练模型时，在隐藏层里随机抽取某些神经元，将它们的输出值强行设置为0，相当于从网络里将该神经元删除。
- Dropout rate：超参数随机失活率，决定有多大比例的神经元会被失活处理，通常设置在0.2到0.5之间。

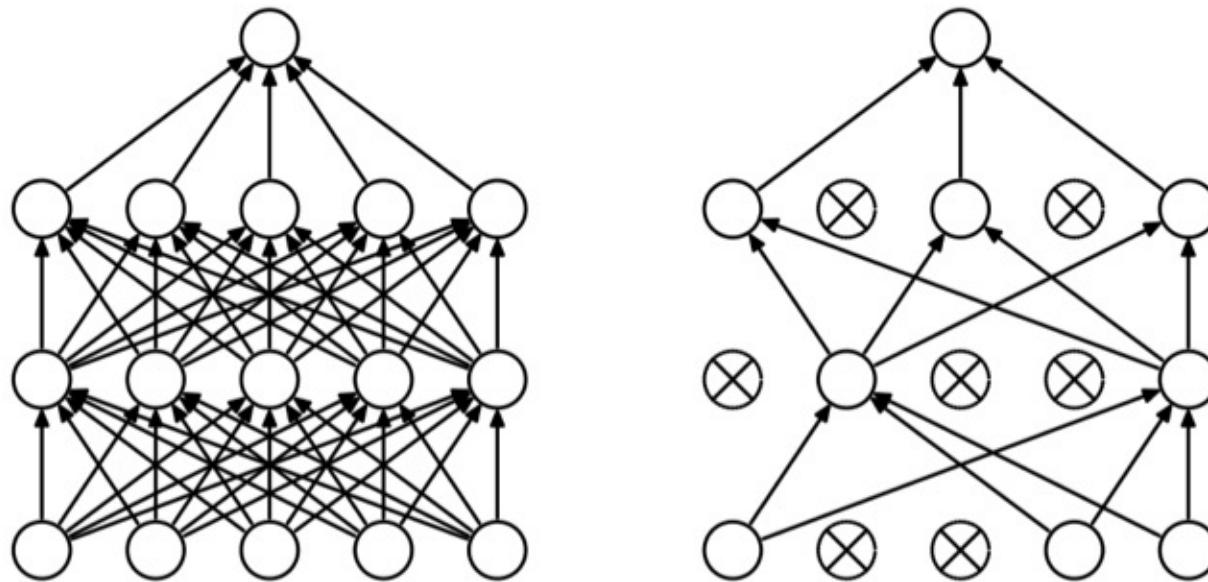


Dropout



促使权重分散，让每个神经元保留更多冗余信息，从而抑制过拟合。

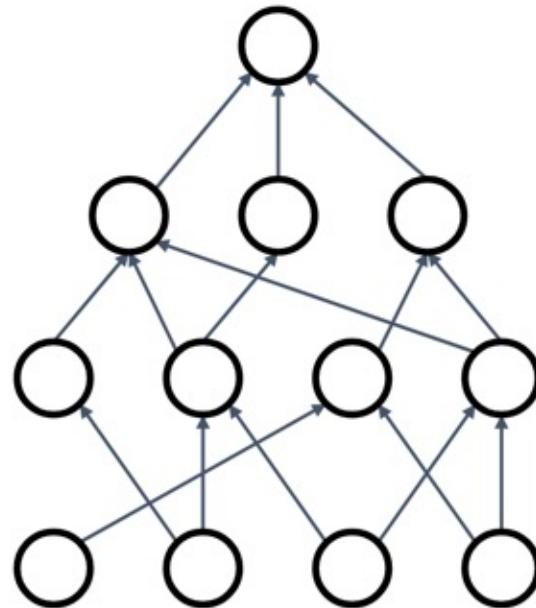
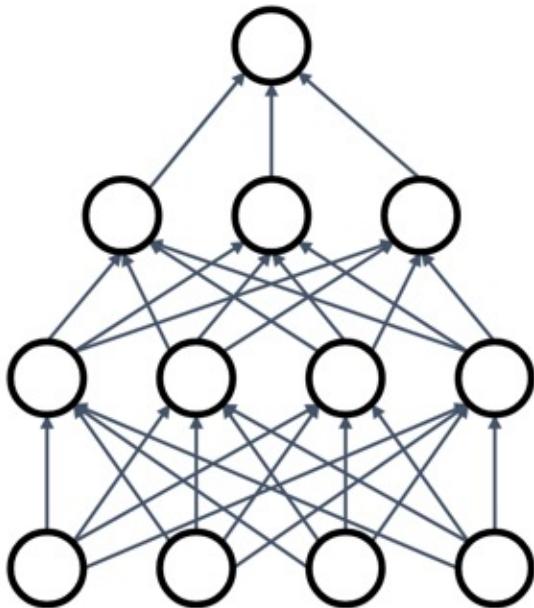
Dropout



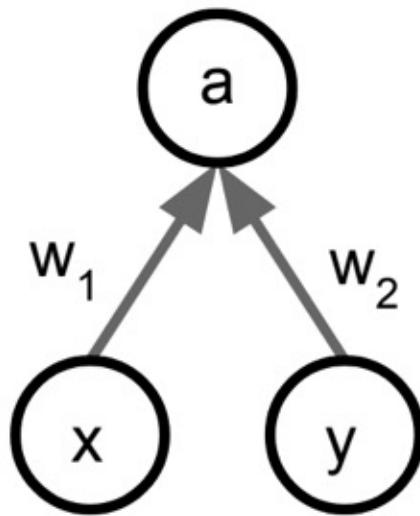
每一个轮次失活的节点是随机的，相当于每一次训练了一个不同的小网络。最终结果等效于将不同的小网络模型做集成，用集成后的模型做决策。

“独立决策”变成了“群体决策”。

DropConnect



Dropout的实现



Dropout Rate = 0.5

输出值期望 (训练)

$$\begin{aligned}E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\&\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\&= \frac{1}{2}(w_1x + w_2y)\end{aligned}$$

输出值期望 (测试/实战)

$$E[a] = w_1x + w_2y$$

需要乘以
dropout rate

输出值的期望在训练和测试时不一样!

Dropout的实现

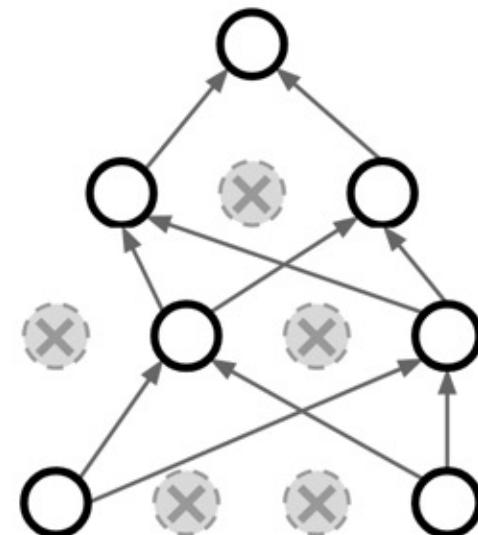
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

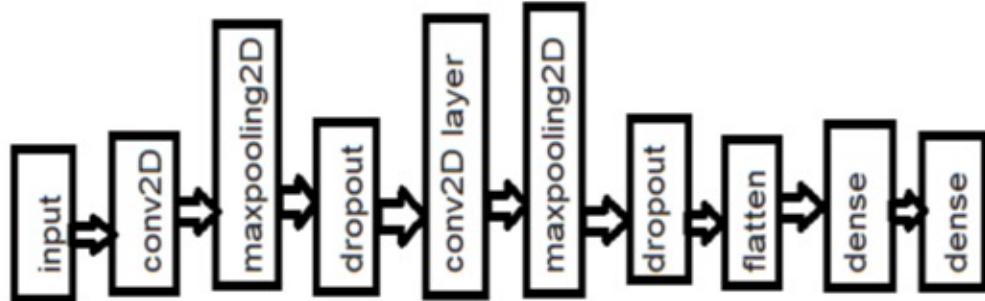
    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```



Dropout层

$p = 0.5$



```
def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3
```

在训练阶段除以p

```
# backward pass: compute gradients... (not shown)
# perform parameter update... (not shown)
```

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

测试时不用再乘以p

输出值期望 (训练)

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) / p \end{aligned}$$

输出值期望 (测试/实战)

$$E[a] = w_1x + w_2y$$

Dropout层

```
model=Sequential()
###first layer
model.add(Dense(100,input_shape=(40,)))
model.add(Activation('relu'))
    ➔ model.add(Dropout(0.5))
###second layer
model.add(Dense(200))
model.add(Activation('relu'))
    ➔ model.add(Dropout(0.5))
###third layer
model.add(Dense(100))
model.add(Activation('relu'))
    ➔ model.add(Dropout(0.5))

###final layer
model.add(Dense(num_labels))
model.add(Activation('softmax'))
```

超参数的选择

超参数的选择

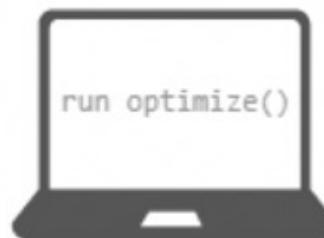


超参数

层数=3
神经元个数=512
学习率=0.1

层数=3
神经元个数=1024
学习率=0.01

层数=5
神经元个数=256
学习率=0.1



模型参数

权值优化

权值优化

权值优化



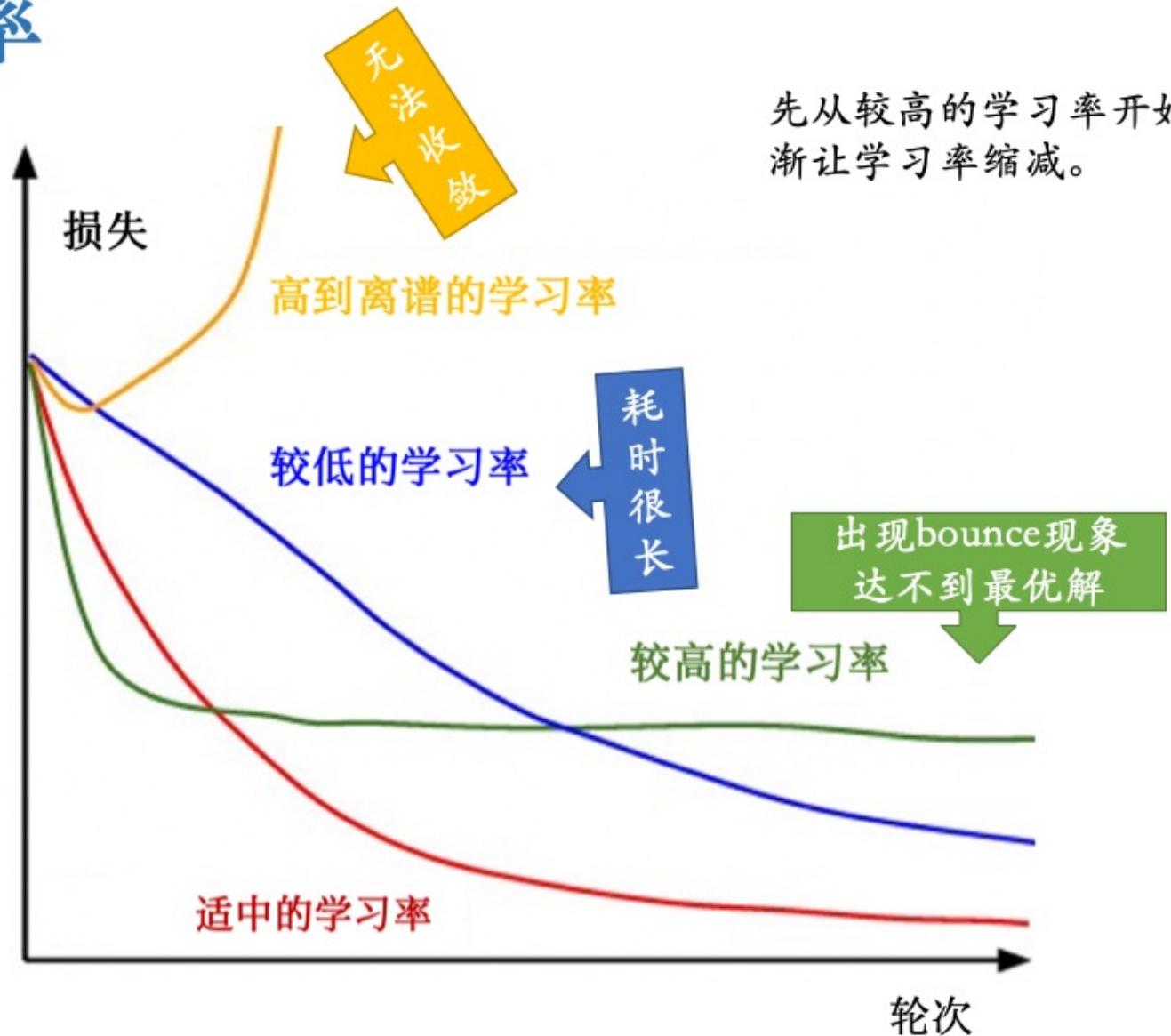
准确率

85%

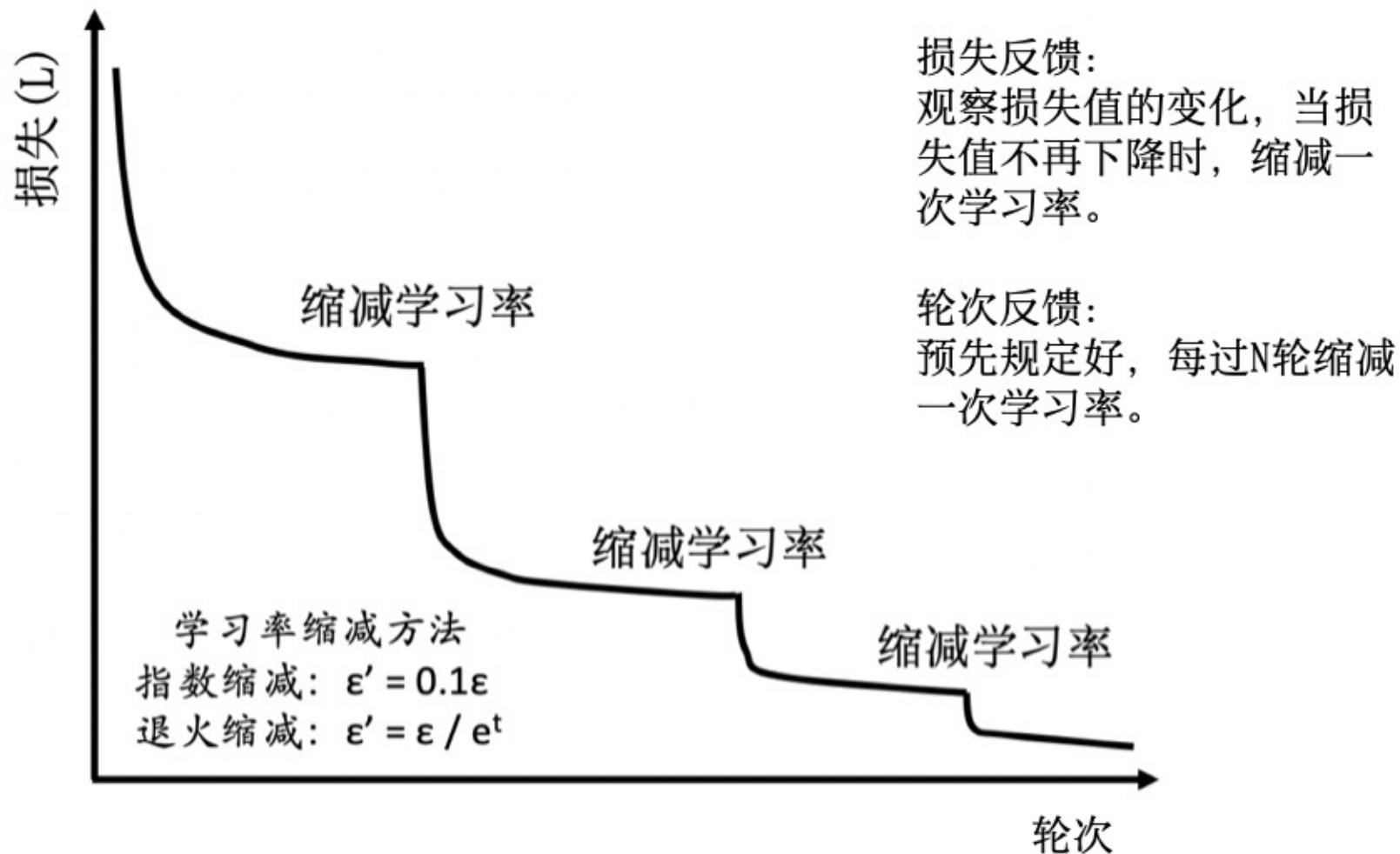
80%

92%

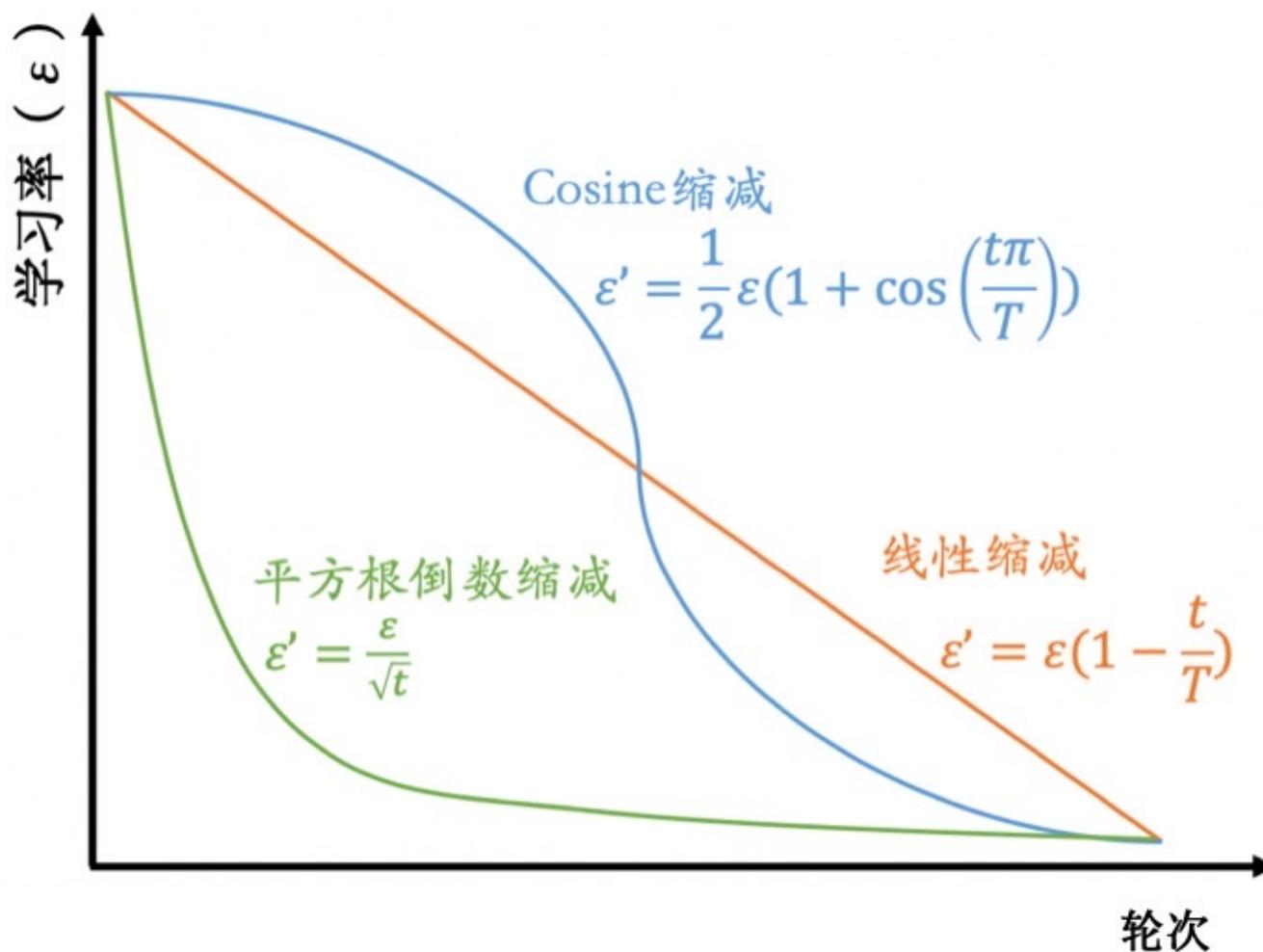
学习率



学习率分级缩减



学习率连续缩减



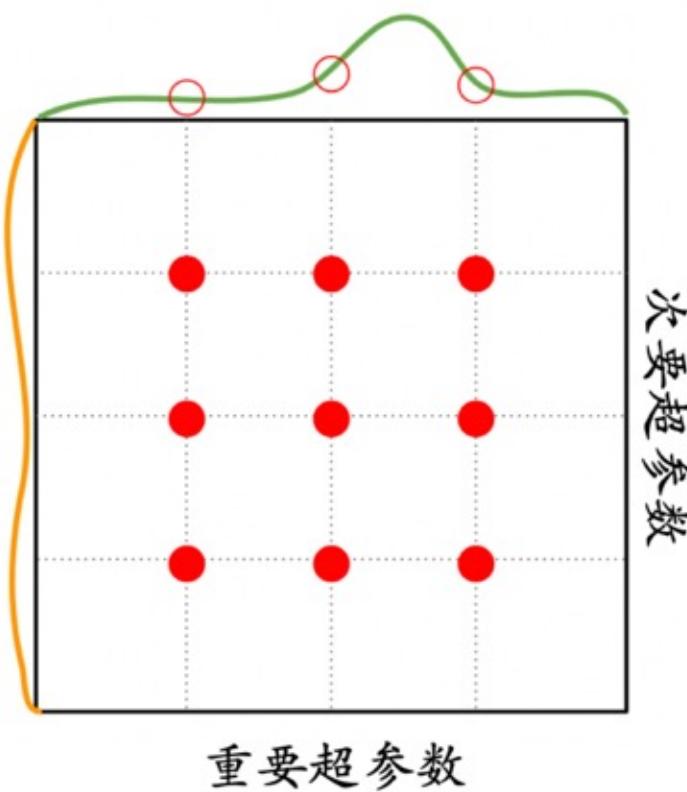
超参数可以在训练过程中发生改变（比如衰减），但改变方式并非模型通过学习来决定，而是事先人为规定好的。模型无权修改这些规定。

下一个问题：

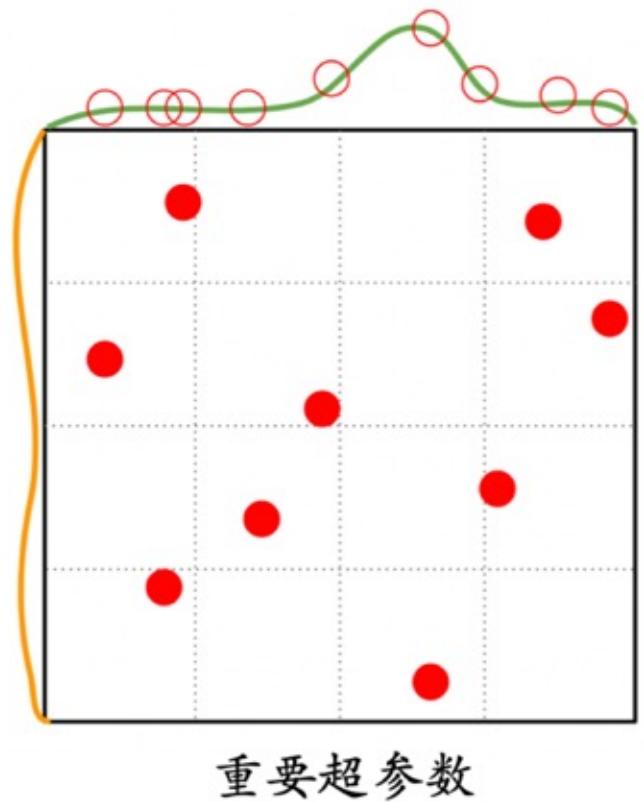
如何一开始就把多个超参数设置到位？

搜索合适的超参数组合

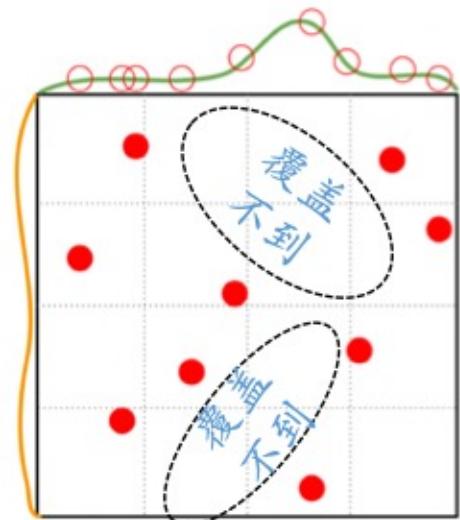
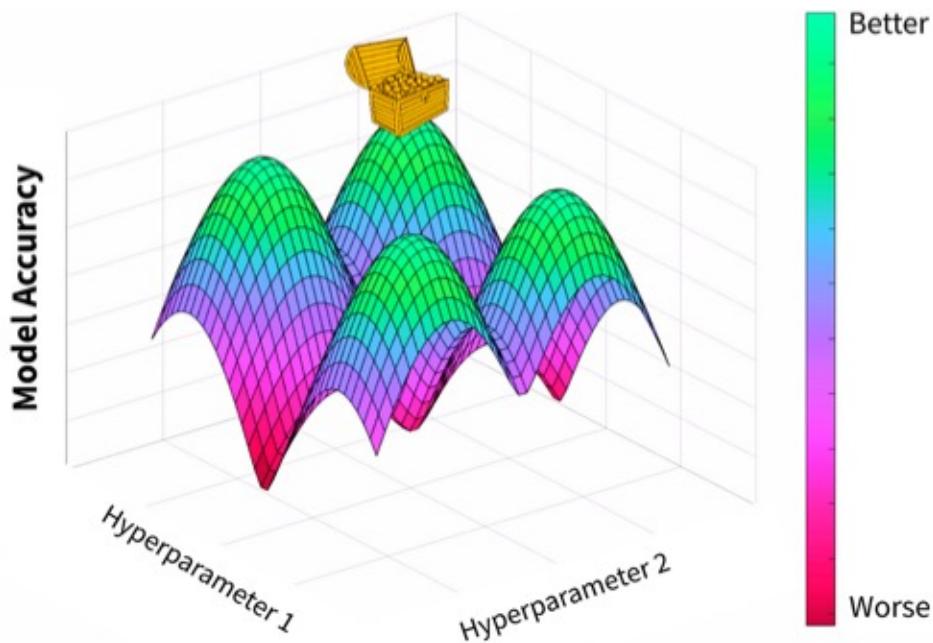
网格搜索



随机搜索



搜索合适的超参数组合

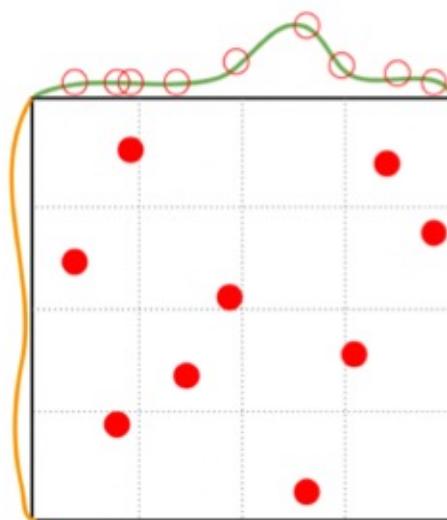


算力不支持全覆盖，
应该怎么办？

自适应搜索

- 第一步：初始状态的损失

取消超参数衰减，直接查看初始状态的损失值。确认模型是正确的。



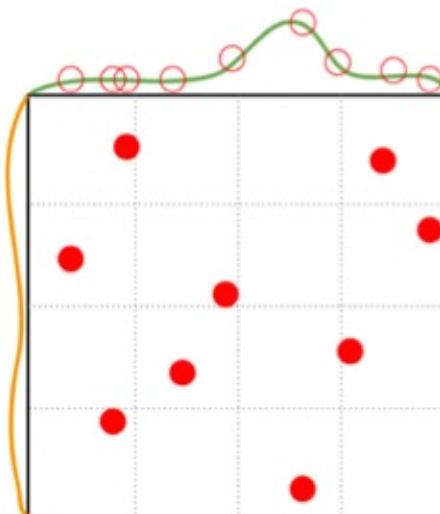
比如，当初始化权值为很小的随机值，损失应为类别数减一。

自适应搜索

- 第一步：初始状态的损失
- 第二步：小样本过拟合

在训练集中抽取很小的一部分，把模型训练到过拟合状态（取消正则化等操作、调整学习率、模型架构、初始化权值等）。保证模型正确。

模型应该快速过拟合，损失值应该快速降为0。如果损失值不下降，说明学习率太小或初始化不好；如果损失值崩溃，说明学习率太大。

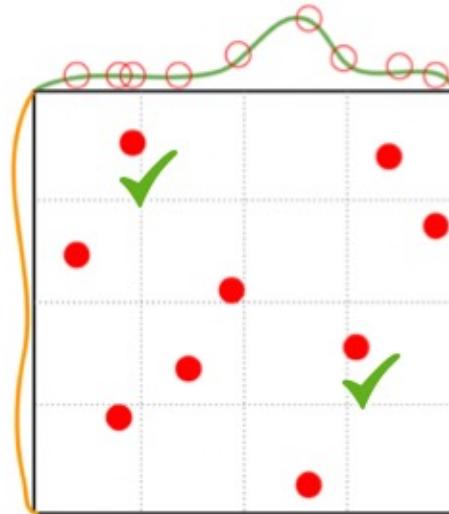


自适应搜索

- 第一步：初始状态的损失
- 第二步：小样本过拟合
- 第三步：寻找让损失下降的学习率

用所有数据来训练前一步得到的模型，尝试一些不同的学习率，找到一个能在100个迭代次数以内让损失值明显下降的学习率。

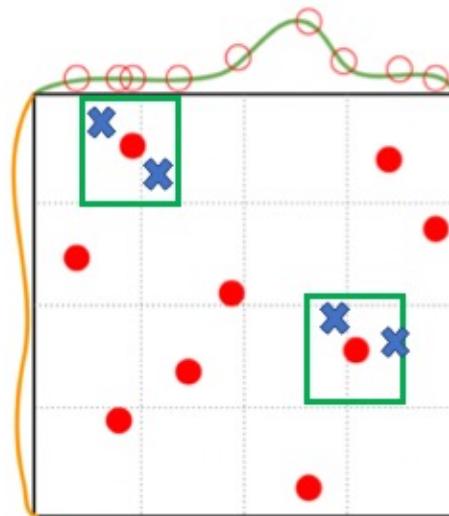
通常从0.1、0.01、0.001、0.0001的学习率开始尝试，寻找“种子选手”



自适应搜索

- 第一步：初始状态的损失
- 第二步：小样本过拟合
- 第三步：寻找让损失下降的学习率
- 第四步：在候选学习率的周围进行粗搜索

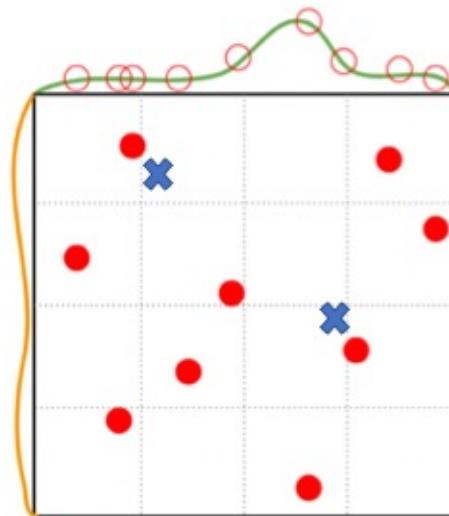
在上一步得到的“种子选手”周围，定义新的搜索范围，抽取一小批超参数组合，继续训练若干轮（epoch）并查看效果。



自适应搜索

- 第一步：初始状态的损失
- 第二步：小样本过拟合
- 第三步：寻找让损失下降的学习率
- 第四步：在候选学习率的周围进行粗搜索
- 第五步：缩小范围，增加训练量

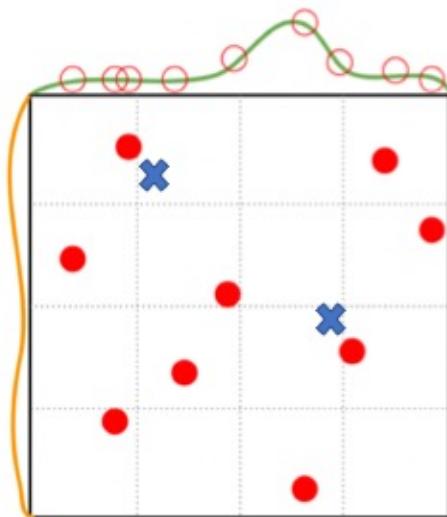
针对上一步缩小范围后剩下的超参数组合，增加训练时长（更多的epoch）查看效果，挑选性能最好的超参数组合。



自适应搜索

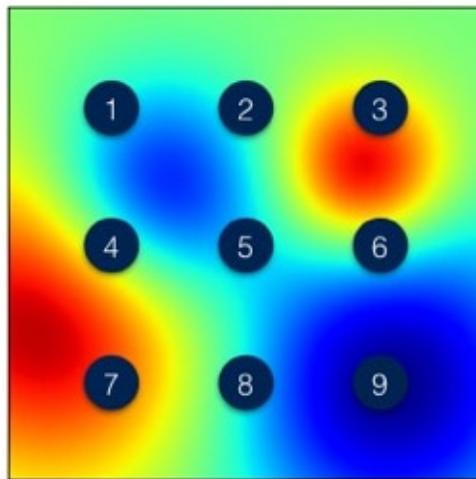
- 第一步：初始状态的损失
- 第二步：小样本过拟合
- 第三步：寻找让损失下降的学习率
- 第四步：在候选学习率的周围进行粗搜索
- 第五步：缩小范围，增加训练量
- 第六步：重复第三步到第五步

用递归的方式“优中取优”，直到找出满足需求的超参数组合。



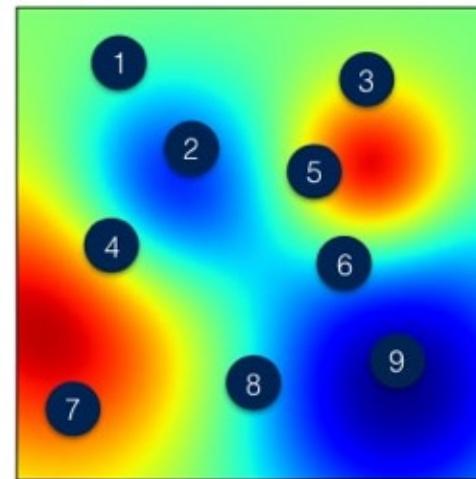
自适应搜索

网格搜索



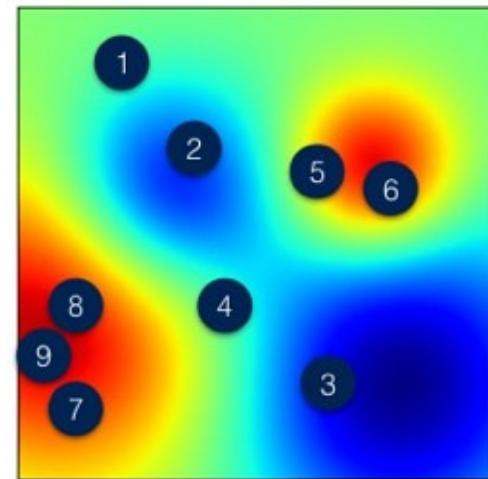
Grid Search

随机搜索



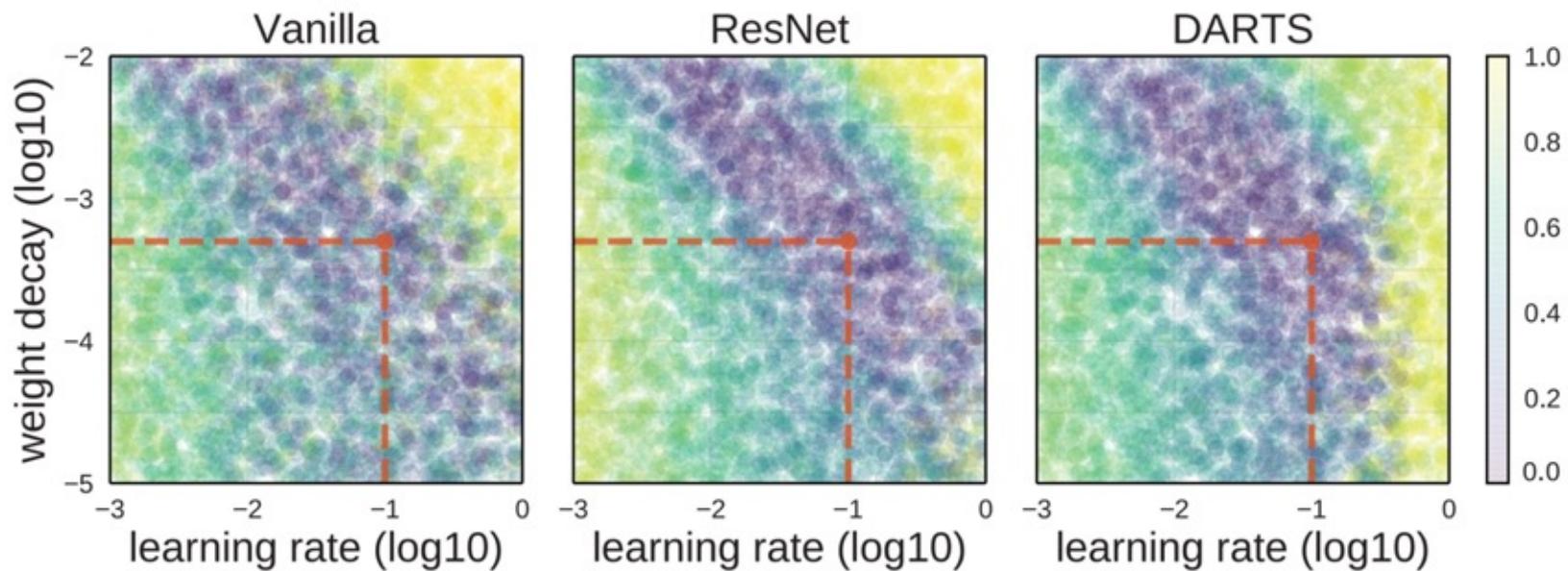
Random Search

自适应搜索



Adaptive Selection

自适应搜索



超参数组合搜索时，坐标系通常用Log空间