

# 机器学习与深度学习 ——反向传播算法与损失函数



Personal Website: <https://www.miaopeng.info/>



Email: [miaopeng@stu.scu.edu.cn](mailto:miaopeng@stu.scu.edu.cn)



Github: <https://github.com/MMeowhite>



Youtube: <https://www.youtube.com/@pengmiao-bmm>

# 目录章节

---

CONTENTS

01 神经网络：当前挑战

02 神经网络：损失函数

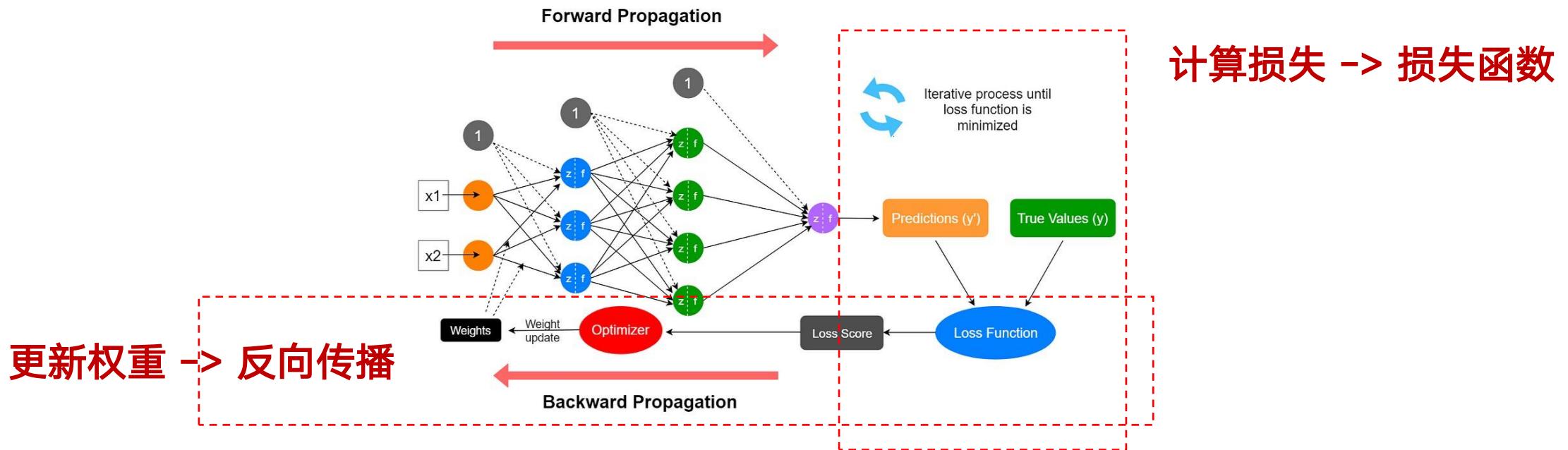
03 神经网络：反向传播算法

04 神经网络：算法实现

05 总结

# ▶ 怎么更新权重？

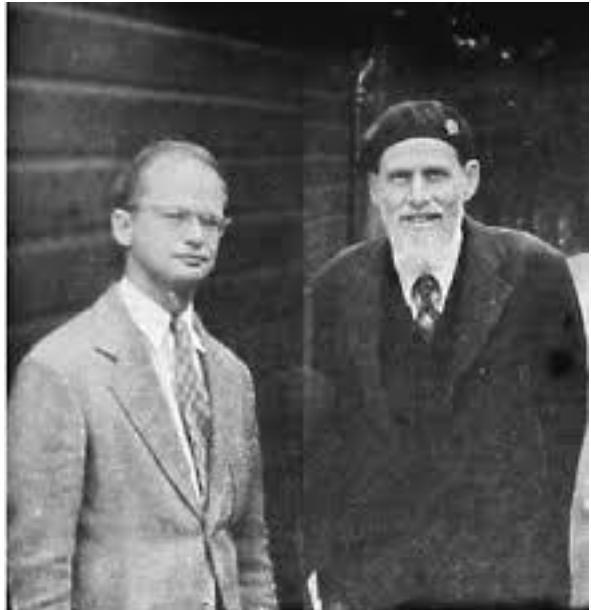
- 训练神经网络 = 调整参数让输出接近目标，但是如果神经网络是一个黑箱，怎么知道它做错了什么？
- 这时就需要提出怎么去衡量前向传播得出的结果（预测值）与标签值（真实值）之间的误差以及怎么根据误差反向调整整个神经网络的权重矩阵呢？



如何衡量做错了? -> 损失函数；知道错了之后怎么修正错误 -> 反向传播

## ► 反向传播与损失函数发展历史：初始阶段

- 1940s – 1950s：该时期为早期神经网络思想萌芽



McCulloch & Pitts



Rosenblatt

- 1943年 McCulloch 和 Pitts 提出人工神经元模型（MCP模型）：逻辑门类比，无法学习。
- 1958年：Frank Rosenblatt 提出感知机（Perceptron）模型：使用阶跃函数作为激活函数；手动调整权重，有简单的“感知机学习规则”

局限：感知机只能处理线性可分问题，无法学习非线性函数（如 XOR）

# ▶ 反向传播与损失函数发展历史：萌芽阶段

- 1974: Werbos在博士学位论文中提出反向传播思想。



Paul John Werbos

BEYOND REGRESSION:  
NEW TOOLS FOR PREDICTION AND ANALYSIS  
IN THE BEHAVIORAL SCIENCES

A thesis presented  
by  
Paul John Werbos  
to  
The Committee on Applied Mathematics  
in partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy  
in the subject of  
statistics

Harvard University  
Cambridge, Massachusetts  
August, 1974

Copyright reserved by the author

right, over  $j$ , for all  $i, t$  and  $m$ , we may reduce the cost down to  $Cn^5T$ . But at this point, the simplifications stop; an "M" of these dimensions, with these properties, is clearly central to the algorithm presented in Box and Jenkins. We could go on to discuss further details of the Marquardt algorithm in the multivariate case, but the number of calculations required -  $Cn^5T$  - is already large enough to contrast strongly with the new algorithm we will present below.

Now: how do we arrive at a less expensive algorithm to accomplish the same objectives?

To begin with, we will build our new algorithm on a well established foundation, the classic method of steepest ascent; we will maximize  $L(\theta, P, \mathbf{z}_1, A)$  by writing:

$$B_k^{(n+1)} = B_k^{(n)} + w g_k \frac{\partial L}{\partial B_k}, \quad (3.41)$$

where  $w$  is an arbitrary scale factor to be adjusted during maximization, and where  $g_k$  is an arbitrary positive "metric factor" to be applied to  $B_k$ . We will include  $\theta$ ,  $P$  and  $\mathbf{z}_1$  as components of  $B$ ; however, we will not include  $A$ . Starting from a given  $B^{(0)}$ ,  $A^{(0)}$  and  $w$ , we will first compute  $\frac{\partial L}{\partial B_k}$ . Then we will compute  $\frac{\partial L}{\partial B}$ . From  $\frac{\partial L}{\partial B}$  alone, equation (3.5) allows us to compute all the  $\{a_t\}$ , from times  $t=1$  to  $t=T$ . It is a well-known fact, for a given set of data,  $\{a_t\}$ , that the maximum likelihood estimate "A" of the covariance matrix generating this data as a random process of zero mean, will simply be

**局限：**当时计算能力不足；存在梯度消失；缺乏大数据和正则化，泛化能力差；改工作偏向理论，控制理论与自动微分视角，并没有结合神经网络做大规模实验。Werbos提供了“数学工具”，但缺乏算力、数据和工程手段，导致当时反向传播在神经网络中难以发挥真正潜力。

# ▶ 反向传播与损失函数发展历史：萌芽阶段

- 1986: Rumelhart、Hinton 和 Williams 在他们著名的论文 "Learning representations by back-propagating errors" 中系统推广了反向传播 (BP) 算法，使多层前馈神经网络的训练成为可能，并结合常用损失函数（如 MSE、交叉熵）来进行有效的误差传播与优化。



Rumelhart、Hinton & Williams

no advantage, we begin by generalizing our analysis to the set of nonlinear activation functions which we call semilinear (see Chapter 2). A semilinear activation function is one in which the output of a unit is a differentiable function of the net total input,

$$net_{ji} = \sum_i w_{ji} o_{pi}, \quad (7)$$

where  $o_i = i_i$  if unit  $i$  is an input unit. Thus, a semilinear activation function is one in which

$$o_{pi} = f_j(net_{ji}) \quad (8)$$

and  $f_j$  is differentiable. The generalized delta rule works if the network consists of units having semilinear activation functions. Notice that linear threshold units do not satisfy the requirement because their derivative is infinite at the threshold and zero elsewhere.

To get the correct generalization of the delta rule, we must set

$$\Delta_p w_{ji} \propto -\frac{\partial E_p}{\partial w_{ji}},$$

where  $E$  is the same non-squared error function defined earlier. As in the standard delta rule it is again useful to see this derivative as resulting from the product of two parts: one part reflecting the change in error as a function of the change in the net input to the unit and one part representing the effect of changing a particular weight on the net input. Thus we can write

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial net_{ji}} \frac{\partial net_{ji}}{\partial w_{ji}}. \quad (9)$$

By Equation 7 we see that the second factor is

$$\frac{\partial net_{ji}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} o_{pk} = o_{pj}. \quad (10)$$

Now let us define

$$\delta_{pj} = \frac{\partial E_p}{\partial net_{ji}}.$$

(By comparing this to Equation 4, note that this is consistent with the definition of  $\delta_{pj}$  used in the original delta rule for linear units since  $\delta_{pj} = net_{ji}$  when unit  $j$  is linear.) Equation 9 thus has the equivalent form

$$\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} o_{pj}.$$

This says that to implement gradient descent in  $E$  we should make our weight changes according to

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pj}, \quad (11)$$

just as in the standard delta rule. The trick is to figure out what  $\delta_{pj}$  should be for each unit  $w_{ji}$  in the network. The interesting result, which we now derive, is that there is a simple recursive computation of these  $\delta$ 's which can be implemented by propagating error signals backward through the network.

To compute  $\delta_{pj} = \frac{\partial E_p}{\partial net_{ji}}$ , we apply the chain rule to write this partial derivative as the product of two factors, one factor reflecting the change in error as a function of the unit and one reflecting the

LEARNING INTERNAL REPRESENTATIONS 7

change in the output as a function of changes in the input. That, we have

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{ji}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{ji}}. \quad (12)$$

Let us compute the second factor. By Equation 8 we see that

$$\frac{\partial o_{pj}}{\partial net_{ji}} = f'_j(net_{ji}),$$

which is the derivative of the semilinear function  $f_j$  for the  $j$ th unit, evaluated at the net input  $net_{ji}$  to that unit. To compute the first factor, we consider two cases. First, assume that unit  $u_j$  is an output unit of the network. In this case, it follows from the definition of  $E_p$  that

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}),$$

which is the same result as we obtained with the standard delta rule. Substituting for the two factors in Equation 12, we get

$$\delta_{pj} = (t_{pj} - o_{pj})f'_j(net_{ji}). \quad (13)$$

for any output unit  $u_j$ . If  $u_j$  is not an output unit we use the chain rule to write

$$\sum_k \frac{\partial E_p}{\partial w_{jk}} \frac{\partial net_{ji}}{\partial w_{jk}} = \sum_k \frac{\partial E_p}{\partial o_{pk}} \frac{\partial o_{pk}}{\partial net_{ji}} = \sum_k w_{jk} \delta_{pk} = \sum_k \frac{\partial E_p}{\partial w_{jk}} w_{jk}.$$

In this case, substituting for the two factors in Equation 12 yields

$$\delta_{pj} = f'_j(net_{ji}) \sum_k w_{jk} \delta_{pk}. \quad (14)$$

whether  $u_j$  is an output unit. Equations 13 and 14 give a recursive procedure for computing the  $\delta$ 's for all units in the network, which are then used to compute the weight changes in the network according to Equation 11. This procedure constitutes the generalized delta rule for a feedforward network of semilinear units.

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pj},$$

The other two equations specify the error signal. Essentially, the determination of the error signal is a recursive process which starts with the output units. If a unit is an output unit, its error signal is very similar to the standard delta rule. It is given by

$$\delta_{pj} = (t_{pj} - o_{pj})f'_j(net_{ji})$$

where  $f'_j(net_{ji})$  is the derivative of the semilinear activation function which maps the total input to the unit to an output value. Finally, the error signal for hidden units for which there is no specified target is determined recursively in terms of the error signals of the units to

**局限：深度受限（梯度消失问题）；容易陷入局部极小值；对初始化与超参数敏感；训练成本高，训练缓慢；泛化能力不足以及过拟合。**

## ► 反向传播与损失函数发展历史：复兴以及现代

- 2006: Hinton & Salakhutdinov "Reducing the dimensionality of data with neural networks" → 提出逐层无监督预训练（RBM + DBN），缓解梯度消失问题，推动深度网络可行。
- 2010: Nair & Hinton 提出 ReLU，缓解梯度消失。
- 2012: AlexNet (Krizhevsky 等) 结合 ReLU、Dropout、GPU，深度 CNN 取得巨大突破。
- 2015: BatchNorm (Ioffe & Szegedy) 稳定深层网络训练，成为标准组件；Kaiming He引入残差结构 (skip connection) ResNet，极大缓解了深层网络梯度消失问题，使得超深网络可训练。Diederik P. Kingma等引入Adam优化器，结合动量和自适应学习率，成为训练深度网络的主流方法。

**局限：**可解释性不足；数据与计算依赖巨大；泛化与鲁棒性不足；训练以及优化难题（超参数高度敏感、梯度消失/爆炸）。是近年来出现因果推理、神经符号结合（Neuro-symbolic）、小样本学习、能效优化等研究方向的动因。

# ▶ 反向传播与损失函数发展历史

- 反向传播和损失函数的发展历程，是深度学习从不可训练的神经元模型走向高效可训练网络系统的关键转折，凝聚了数十年在数学、计算与实践中的突破与演进。

时间	损失函数进展	反向传播进展
1940s-50s	初步误差概念	无
1974	-	Werbos提出反向传播思想
1986	MSE + Sigmoid + Softmax	Rumelhart等推广BP算法
1990s	交叉熵流行	BP应用困难（梯度消失）
2006	-	Hinton重启深度学习，恢复BP
2012	多样损失广泛使用	AlexNet推动BP在深度网络广泛应用
2020s	感知损失、对比损失、自监督损失	无监督方法、自动微分、并行优化探索等

# 目录章节

---

CONTENTS

01

神经网络：当前挑战

02

神经网络：损失函数

03

神经网络：反向传播算法

04

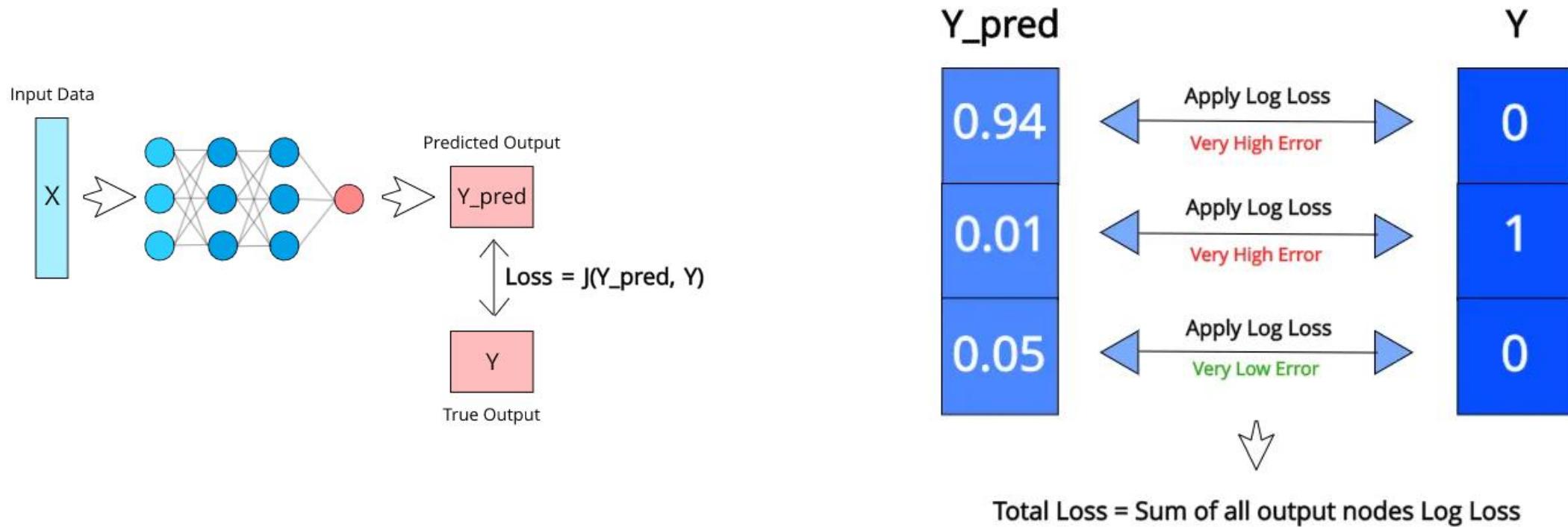
神经网络：算法实现

05

总结

# ▶ 什么是损失函数？

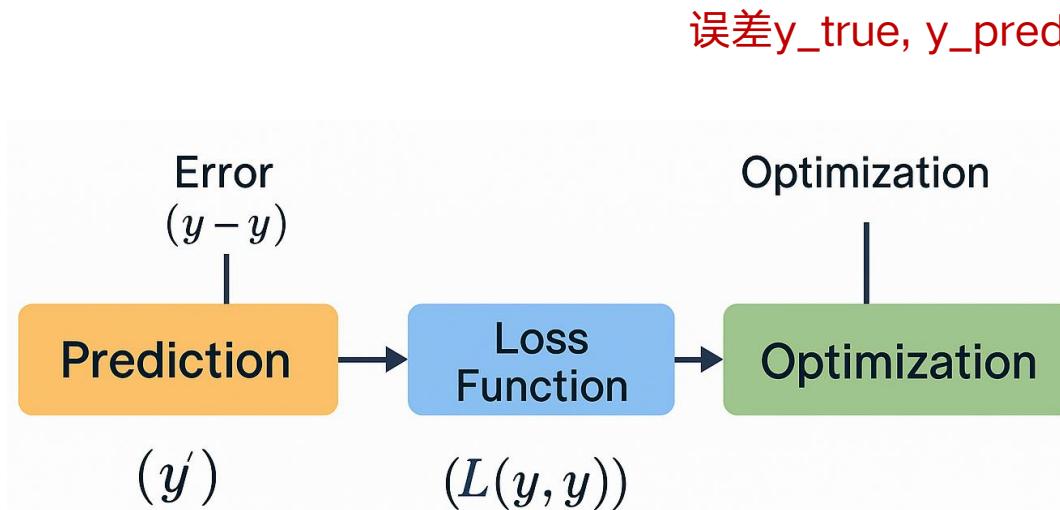
- 定义：衡量模型输出与真实值之间误差的函数。
- 常见损失函数示例：1) MSE（均方误差） - 回归；2) Cross Entropy（交叉熵） - 分类；Hinge loss - 支持向量机。



**损失函数是衡量模型输出与真实目标之间差距的标尺，指引模型学习方向的核心。**

## ▶ 损失函数与误差（全局）

- 误差（Error）【全局】：最终网络输出的预测值和真实值之间的差异，比如  $e = y_{\text{true}} - y_{\text{pred}}$ 。衡量整个网络表现得好不好，是训练的总目标。可能是向量（多维），有正有负，不能直接衡量整体好坏。
- 损失函数（Loss Function）：把误差转换为可优化的标量，用来衡量模型整体性能。作用是把误差变成一个“方向明确的目标”，方便通过梯度下降最小化。



误差  $y_{\text{true}}, y_{\text{pred}}$  + 函数  $f = \text{损失函数 } L$

$$L = \frac{1}{2} (y - \hat{y})^2$$
$$L = - \sum y \log \hat{y}$$

The diagram shows two common loss functions. The top part shows the Mean Squared Error (MSE) loss, where the difference between the predicted value  $\hat{y}$  and the true value  $y$  is squared and then averaged by 2. The bottom part shows the Cross-Entropy loss, which is the negative sum of the true values  $y$  multiplied by the logarithm of the predicted probabilities  $\hat{y}$ .

误差是模型预测与真实值的偏差，损失函数是对误差进行加工后得到的可优化目标，用于指导训练。

# ▶ 损失函数的种类

- 损失函数大致可分为**回归类、分类类、对比学习类、生成模型类与任务特定类**（如分割或排序），每种对应不同的任务需求和数据特性。选择合适的损失函数能够直接影响模型的学习效果与泛化能力，是模型训练的关键一环。
- 选择损失函数的依据：1) 任务性质（回归/分类）；2) 模型性质（是否输出概率）；3) 鲁棒性（对异常值的敏感程度）。

Classification Loss Functions (Binary + Multi-class)		
Binary Cross Entropy (BCE)	Loss function for binary classification tasks.	$\mathcal{L}_{BCE} = \frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(x_i)) + (1 - y_i) \cdot \log(1 - p(x_i))$
Hinge Loss	Penalizes wrong and right (but less confident) predictions. Commonly used in SVMs.	$\mathcal{L}_{\text{Hinge}} = \max(0, 1 - (f(x) \cdot y))$
Cross Entropy Loss	Extension of BCE loss to multi-class classification.	$\mathcal{L}_{ce} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \cdot \log(f(x_{ij}))$ <small>N : samples; M : classes</small>
KL Divergence	Minimizes the divergence between predicted and true probability distribution	$\mathcal{L}_{KL} = \sum_{i=1}^N y_i \cdot \log(\frac{y_i}{f(x_i)})$

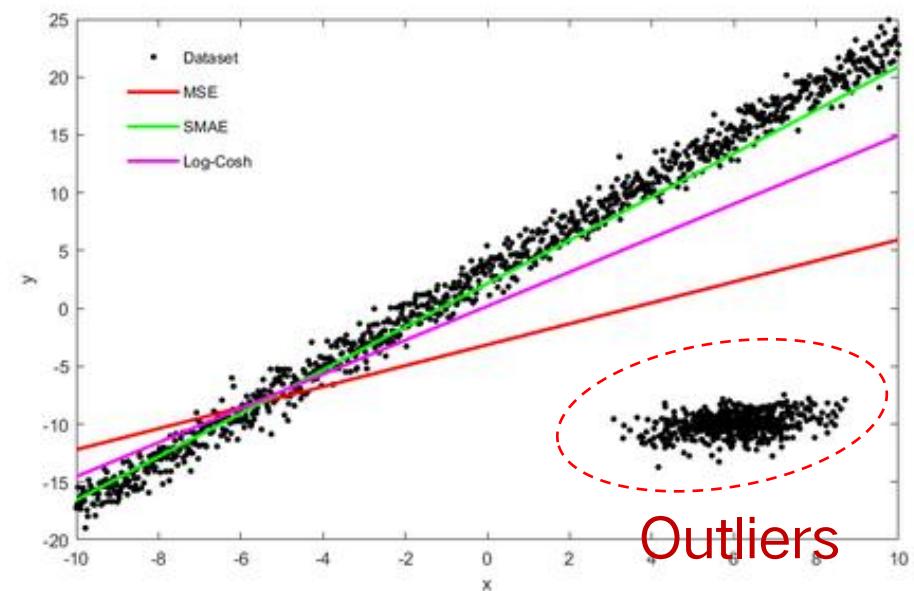
Regression Loss Functions		
Mean Bias Error	Captures average bias in prediction. But is rarely used for training.	$\mathcal{L}_{MBE} = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))$
Mean Absolute Error	Measures absolute average bias in prediction. Also called L1 Loss.	$\mathcal{L}_{MAE} = \frac{1}{N} \sum_{i=1}^N  y_i - f(x_i) $
Mean Squared Error	Average squared distance between actual and predicted. Also called L2 Loss.	$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2$
Root Mean Squared Error	Square root of MSE. Loss and dependent variable have same units.	$\mathcal{L}_{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2}$
Huber Loss	A combination of MSE and MAE. It is parametric loss function.	$\mathcal{L}_{\text{Huber}} = \begin{cases} \frac{1}{2}(y_i - f(x_i))^2 & :  y_i - f(x_i)  \leq \delta \\ \delta( y_i - f(x_i)  - \frac{1}{2}\delta) & : \text{otherwise} \end{cases}$
Log Cosh Loss	Similar to Huber Loss + non-parametric. But computationally expensive.	$\mathcal{L}_{\text{LogCosh}} = \frac{1}{N} \sum_{i=1}^N \log(\cosh(f(x_i) - y_i))$

损失函数定义了“学习的目标”，不同任务对应不同设计，从回归到生成，从分类到对比，每一种都是模型学习方向的指南针。

# ▶ 损失函数：回归任务

- 回归任务中常用的损失函数包括均方误差（MSE）、平均绝对误差（MAE）和 Huber 损失等，分别在精度、鲁棒性与平衡性之间取舍。

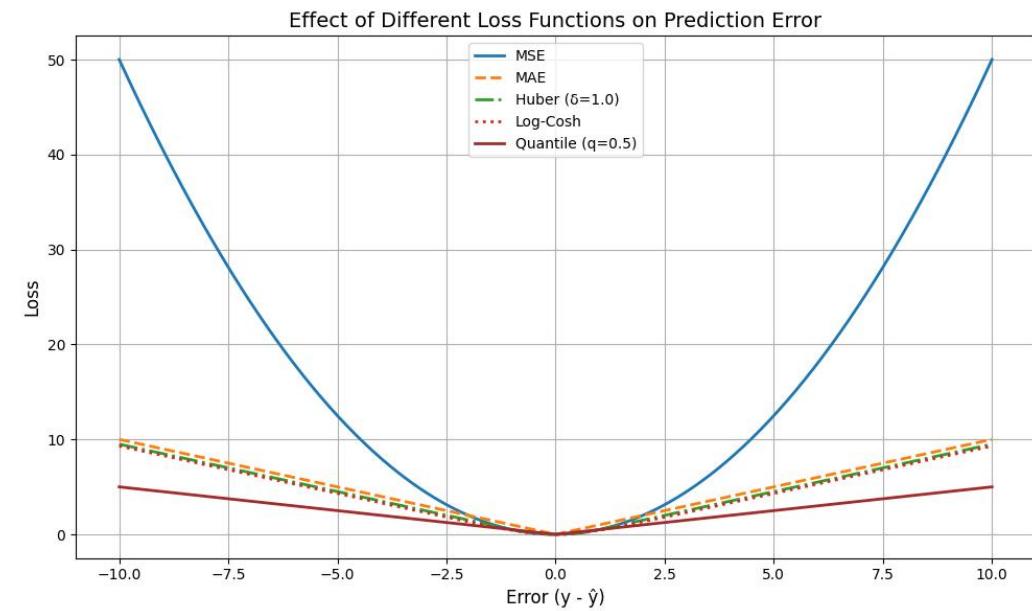
损失函数名称	公式	特点与应用
均方误差 (MSE)	$\frac{1}{n} \sum (y - \hat{y})^2$	对大误差敏感，常用于回归
平均绝对误差 (MAE)	$\frac{1}{n} \sum  y - \hat{y} $	对异常值不敏感，结果更稳健，可解释性强
Huber Loss	$L_\delta(a) = \begin{cases} \frac{1}{2}(a)^2 & \text{if }  a  \leq \delta \\ \delta( a  - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$	平衡 MSE 与 MAE 的优点
Log-Cosh Loss	$\sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i))$	平滑、对离群值不敏感
Quantile Loss	$\sum_{i=1}^n [q \cdot \max(y_i - \hat{y}_i, 0) + (1 - q) \cdot \max(\hat{y}_i - y_i, 0)]$	用于分位数回归（如 90% 分位）



不同损失函数对异常值的敏感程度不同，根据需求选择合适的函数以提高模型鲁棒性和精度。

# ▶ 损失函数：回归任务损失函数比较

损失函数名称	公式	特点与应用
均方误差 (MSE)	$\frac{1}{n} \sum (y - \hat{y})^2$	对大误差敏感，常用于回归
平均绝对误差 (MAE)	$\frac{1}{n} \sum  y - \hat{y} $	常用于回归
Huber Loss	$L_\delta(a) = \begin{cases} \frac{1}{2}(a)^2 & \text{if }  a  \leq \delta \\ \delta( a  - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$	平衡 MSE 与 MAE 的优点
Log-Cosh Loss	$\sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i))$	平滑、对离群值不敏感
Quantile Loss	$\sum_{i=1}^n [q \cdot \max(y_i - \hat{y}_i, 0) + (1 - q) \cdot \max(\hat{y}_i - y_i, 0)]$	用于分位数回归 (如 90% 分位)



误差在增大时（模拟异常值）不同损失函数的增长趋势

- 总体特点：1) MSE 增长最快，对离群点最敏感；2) MAE 增长线性，鲁棒；3) Huber 在小误差时像 MSE，大误差时像 MAE；3) Log-Cosh 类似 MSE，但更平滑；5) Quantile 非对称时也可用于倾斜分布预测。

# ▶ 损失函数：回归任务损失函数代码实现（numpy）

损失函数名称	公式	特点与应用
均方误差 (MSE)	$\frac{1}{n} \sum (y - \hat{y})^2$	对大误差敏感，常用于回归
平均绝对误差 (MAE)	$\frac{1}{n} \sum  y - \hat{y} $	常用于回归
Huber Loss	$L_\delta(a) = \begin{cases} \frac{1}{2}(a)^2 & \text{if }  a  \leq \delta \\ \delta( a  - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$	平衡 MSE 与 MAE 的优点
Log-Cosh Loss	$\sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i))$	平滑、对离群值不敏感
Quantile Loss	$\sum_{i=1}^n [q \cdot \max(y_i - \hat{y}_i, 0) + (1 - q) \cdot \max(\hat{y}_i - y_i, 0)]$	用于分位数回归 (如 90% 分位)

```
import numpy as np

# Loss Functions for Regression Tasks

# Mean Squared Error (MSE)
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Mean Absolute Error (MAE)
def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# Huber Loss
def huber_loss(y_true, y_pred, delta=1.0):
    error = y_true - y_pred
    is_small_error = np.abs(error) <= delta
    squared_loss = 0.5 * error**2
    linear_loss = delta * (np.abs(error) - 0.5 * delta)
    return np.mean(np.where(is_small_error, squared_loss, linear_loss))

# Log-Cosh Loss
def log_cosh_loss(y_true, y_pred):
    error = y_true - y_pred
    return np.mean(np.log(np.cosh(error)))

# Quantile Loss
def quantile_loss(y_true, y_pred, quantile=0.5):
    error = y_true - y_pred
    return np.mean(np.maximum(quantile * error, (quantile - 1) * error))

# Example usage
y_true = np.array([3.0, 5.0, 7.0])
y_pred = np.array([2.5, 5.0, 8.0])

print("MSE:", mse(y_true, y_pred))
print("MAE:", mae(y_true, y_pred))
print("Huber:", huber_loss(y_true, y_pred))
print("Log-Cosh:", log_cosh_loss(y_true, y_pred))
print("Quantile (q=0.7):", quantile_loss(y_true, y_pred, quantile=0.7))

✓ 0.0s
```

```
MSE: 0.4166666666666667
MAE: 0.5
Huber: 0.2083333333333334
Log-Cosh: 0.18463177914710152
Quantile (q=0.7): 0.2166666666666667
```

# ▶ 损失函数：回归任务损失函数代码实现练习（pytorch）

损失函数名称	公式	特点与应用
均方误差 (MSE)	$\frac{1}{n} \sum (y - \hat{y})^2$	对大误差敏感，常用于回归
平均绝对误差 (MAE)	$\frac{1}{n} \sum  y - \hat{y} $	常用于回归
Huber Loss	$L_\delta(a) = \begin{cases} \frac{1}{2}(a)^2 & \text{if }  a  \leq \delta \\ \delta( a  - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$	平衡 MSE 与 MAE 的优点
Log-Cosh Loss	$\sum_{i=1}^n \log(\cosh(y_i - \hat{y}_i))$	平滑、对离群值不敏感
Quantile Loss	$\sum_{i=1}^n [q \cdot \max(y_i - \hat{y}_i, 0) + (1 - q) \cdot \max(\hat{y}_i - y_i, 0)]$	用于分位数回归 (如 90% 分位)

```
# 练习：使用pytorch实现这些损失函数
import torch

# Mean Squared Error (MSE) in PyTorch
def mse_torch(y_true, y_pred):
    pass

# Mean Absolute Error (MAE) in PyTorch
def mae_torch(y_true, y_pred):
    pass

# Huber Loss in PyTorch
def huber_loss_torch(y_true, y_pred, delta=1.0):
    pass

def log_cosh_loss_torch(y_true, y_pred):
    pass

def quantile_loss_torch(y_true, y_pred, quantile=0.5):
    pass

# Example usage in PyTorch
# 不要修改以下代码
y_true_torch = torch.tensor([3.0, 5.0, 7.0])
y_pred_torch = torch.tensor([2.5, 5.0, 8.0])

print("MSE (PyTorch):", mse_torch(y_true_torch, y_pred_torch))
print("MAE (PyTorch):", mae_torch(y_true_torch, y_pred_torch))
print("Huber (PyTorch):", huber_loss_torch(y_true_torch, y_pred_torch))
print("Log-Cosh (PyTorch):", log_cosh_loss_torch(y_true_torch, y_pred_torch))

print("Quantile (q=0.7) (PyTorch):", quantile_loss_torch(y_true_torch, y_pred_torch, quantile=0.7))
assert torch.allclose(mse_torch(y_true_torch, y_pred_torch), torch.tensor(0.4167), atol=1e-4)
assert torch.allclose(mae_torch(y_true_torch, y_pred_torch), torch.tensor(0.5000), atol=1e-4)
assert torch.allclose(huber_loss_torch(y_true_torch, y_pred_torch), torch.tensor(0.2083), atol=1e-4)
assert torch.allclose(log_cosh_loss_torch(y_true_torch, y_pred_torch), torch.tensor(0.1846), atol=1e-4)
assert torch.allclose(quantile_loss_torch(y_true_torch, y_pred_torch, quantile=0.7), torch.tensor(0.4167), atol=1e-4)
print("恭喜你！测试通过！")
```

## ▶ 损失函数：分类任务

- 分类任务中常用的损失函数包括交叉熵（Cross Entropy）、二元交叉熵（Binary Cross Entropy）和焦点损失（Focal Loss）等，分别针对多类别、多标签及类别不平衡问题进行优化。

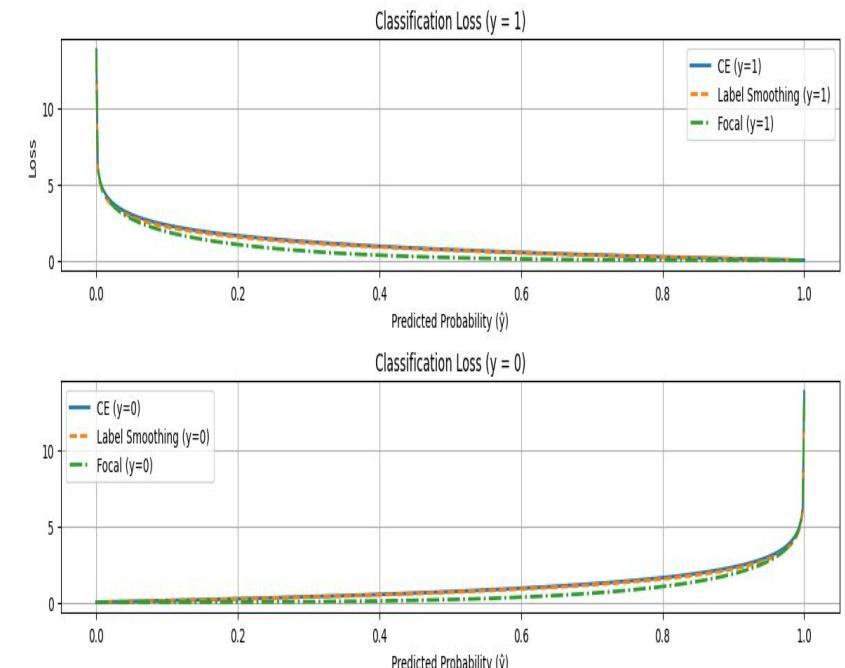
损失函数名称	公式	特点与应用	
交叉熵（Cross Entropy）	$-\sum y \log(\hat{y})$	多分类标准，配合 softmax	$K$ 是类别总数。 $y_i$ 是真实标签的独热编码（通常为0或1）。 $\hat{y}_i$ 是模型预测的类别概率。
二元交叉熵（Binary CE）	$-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$	二分类任务	$\epsilon$ 是平滑因子（如0.1），将部分概率均匀分配到非真实类别上，防止过拟合。
标签平滑交叉熵	$-\sum_{i=1}^K \left( (1 - \epsilon) \cdot y_i + \frac{\epsilon}{K} \right) \log(\hat{y}_i)$	防止过拟合，提升泛化	
Focal Loss	$-\alpha_t (1 - p_t)^\gamma \log(p_t)$	处理类别不平衡（如目标检测）	$p_t$ 是模型对正类或负类的预测概率。 $\alpha_t$ 是平衡因子，用于处理类别不平衡。 $\gamma$ 是调节难易样本的聚焦参数。
KL 散度	$\sum_i P(i) \log \frac{P(i)}{Q(i)}$	衡量两个分布差异，用于生成模型	$P$ 是真实分布， $Q$ 是预测分布。

不同损失函数对异常值的敏感程度不同，根据需求选择合适的函数以提高模型鲁棒性和精度。

# ▶ 损失函数：分类任务损失函数比较

损失函数名称	公式	特点与应用
交叉熵 (Cross Entropy)	$-\sum y \log(\hat{y})$	多分类标准，配合 softmax
二元交叉熵 (Binary CE)	$\sum [-y \log \hat{y} - (1 - y) \log(1 - \hat{y})]$	二分类任务
标签平滑交叉熵	$-\sum_{i=1}^K \left( (1 - \epsilon) \cdot y_i + \frac{\epsilon}{K} \right) \log(\hat{y}_i)$	防止过拟合，提升泛化
Focal Loss	$-\alpha_t (1 - p_t)^\gamma \log(p_t)$	处理类别不平衡（如目标检测）
KL 散度	$\sum_i P(i) \log \frac{P(i)}{Q(i)}$	衡量两个分布差异，用于生成模型

Classification Loss Functions Comparison



概率变化时候不同损失函数的增长趋势

- ▶ 总体特点：1) 交叉熵最常用，惩罚错误预测；2) 标签平滑用于缓解过拟合，鼓励概率分布更“软”；3) Focal Loss专注于难分类样本，常用于目标检测。

# ▶ 损失函数：分类任务损失函数实现（numpy）

损失函数名称	公式	特点与应用
交叉熵 (Cross Entropy)	$-\sum y \log(\hat{y})$	多分类标准，配合 softmax
二元交叉熵 (Binary CE)	$\sum [-y \log \hat{y} - (1 - y) \log(1 - \hat{y})]$	二分类任务
标签平滑交叉熵	$-\sum_{i=1}^K \left( (1 - \epsilon) \cdot y_i + \frac{\epsilon}{K} \right) \log(\hat{y}_i)$	防止过拟合，提升泛化
Focal Loss	$-\alpha_t (1 - p_t)^\gamma \log(p_t)$	处理类别不平衡（如目标检测）
KL 散度	$\sum_i P(i) \log \frac{P(i)}{Q(i)}$	衡量两个分布差异，用于生成模型

```

import numpy as np

# Binary Cross Entropy
def binary_cross_entropy(y_true, y_pred, eps=1e-12):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

# Categorical Cross Entropy (One-hot labels)
def categorical_cross_entropy(y_true, y_pred, eps=1e-12):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))

# Label Smoothing Cross Entropy
def label_smoothing_cross_entropy(y_true, y_pred, epsilon=0.1, eps=1e-12):
    K = y_true.shape[1] # number of classes
    y_pred = np.clip(y_pred, eps, 1 - eps)
    y_smooth = (1 - epsilon) * y_true + epsilon / K
    return -np.mean(np.sum(y_smooth * np.log(y_pred), axis=1))

# Focal Loss (Binary classification only)
def focal_loss(y_true, y_pred, gamma=2.0, eps=1e-12):
    y_pred = np.clip(y_pred, eps, 1 - eps)
    pt = np.where(y_true == 1, y_pred, 1 - y_pred)
    return -np.mean((1 - pt) ** gamma * np.log(pt))

# KL Divergence (One-hot true vs predicted probability)
def kl_divergence(y_true, y_pred, eps=1e-12):
    y_true = np.clip(y_true, eps, 1 - eps)
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return np.mean(np.sum(y_true * np.log(y_true / y_pred), axis=1))

# Example usage
y_true = np.array([[1, 0], [0, 1], [1, 0]])
y_pred = np.array([[0.9, 0.1], [0.2, 0.8], [0.6, 0.4]])
print("Binary Cross Entropy:", binary_cross_entropy(y_true[:, 0], y_pred[:, 0]))
print("Categorical Cross Entropy:", categorical_cross_entropy(y_true, y_pred))
print("Label Smoothing Cross Entropy:", label_smoothing_cross_entropy(y_true, y_pred))
print("Focal Loss:", focal_loss(y_true[:, 0], y_pred[:, 0]))
print("KL Divergence:", kl_divergence(y_true, y_pred))

```

✓ 0.0s

```

Binary Cross Entropy: 0.2797765635793423
Categorical Cross Entropy: 0.2797765635793423
Label Smoothing Cross Entropy: 0.3462596310220802
Focal Loss: 0.03057048233723506
KL Divergence: 0.2797765635520409

```

# ▶ 损失函数：分类任务损失函数代码实现练习（pytorch）

损失函数名称	公式	特点与应用
交叉熵 (Cross Entropy)	$-\sum y \log(\hat{y})$	多分类标准，配合 softmax
二元交叉熵 (Binary CE)	$\sum [-y \log \hat{y} - (1 - y) \log(1 - \hat{y})]$	二分类任务
标签平滑交叉熵	$-\sum_{i=1}^K \left( (1 - \epsilon) \cdot y_i + \frac{\epsilon}{K} \right) \log(\hat{y}_i)$	防止过拟合，提升泛化
Focal Loss	$-\alpha_t (1 - p_t)^\gamma \log(p_t)$	处理类别不平衡（如目标检测）
KL 散度	$\sum_i P(i) \log \frac{P(i)}{Q(i)}$	衡量两个分布差异，用于生成模型

```
# 练习：使用pytorch实现这些损失函数
import torch

# Mean Squared Error (MSE) in PyTorch
def binary_cross_entropy_torch(y_true, y_pred):
    pass

# Categorical Cross Entropy in PyTorch
def categorical_cross_entropy_torch(y_true, y_pred):
    pass

# Label Smoothing Cross Entropy in PyTorch
def label_smoothing_cross_entropy_torch(y_true, y_pred, epsilon=0.1):
    pass

# Focal Loss in PyTorch
def focal_loss_torch(y_true, y_pred, gamma=2.0):
    pass

# KL Divergence in PyTorch
def kl_divergence_torch(y_true, y_pred):
    pass

# Example usage in PyTorch
# 不要修改以下代码
y_true_torch = torch.tensor([[1.0, 0.0], [0.0, 1.0], [1.0, 0.0]])
y_pred_torch = torch.tensor([[0.9, 0.1], [0.2, 0.8], [0.6, 0.4]])
print("Binary Cross Entropy (PyTorch):", binary_cross_entropy_torch(y_true_torch[:, 0], y_pred_torch))
print("Categorical Cross Entropy (PyTorch):", categorical_cross_entropy_torch(y_true_torch, y_pred))
print("Label Smoothing Cross Entropy (PyTorch):", label_smoothing_cross_entropy_torch(y_true_torch))
print("Focal Loss (PyTorch):", focal_loss_torch(y_true_torch[:, 0], y_pred_torch[:, 0]))
print("KL Divergence (PyTorch):", kl_divergence_torch(y_true_torch, y_pred_torch))

assert torch.allclose(binary_cross_entropy_torch(y_true_torch[:, 0], y_pred_torch[:, 0]), torch.tensor(0.0))
assert torch.allclose(categorical_cross_entropy_torch(y_true_torch, y_pred_torch), torch.tensor(0.0))
assert torch.allclose(label_smoothing_cross_entropy_torch(y_true_torch, y_pred_torch), torch.tensor(0.0))
assert torch.allclose(focal_loss_torch(y_true_torch[:, 0], y_pred_torch[:, 0]), torch.tensor(0.030))
assert torch.allclose(kl_divergence_torch(y_true_torch, y_pred_torch), torch.tensor(0.2798), atol=1e-05)
print("恭喜你！测试通过！")
```

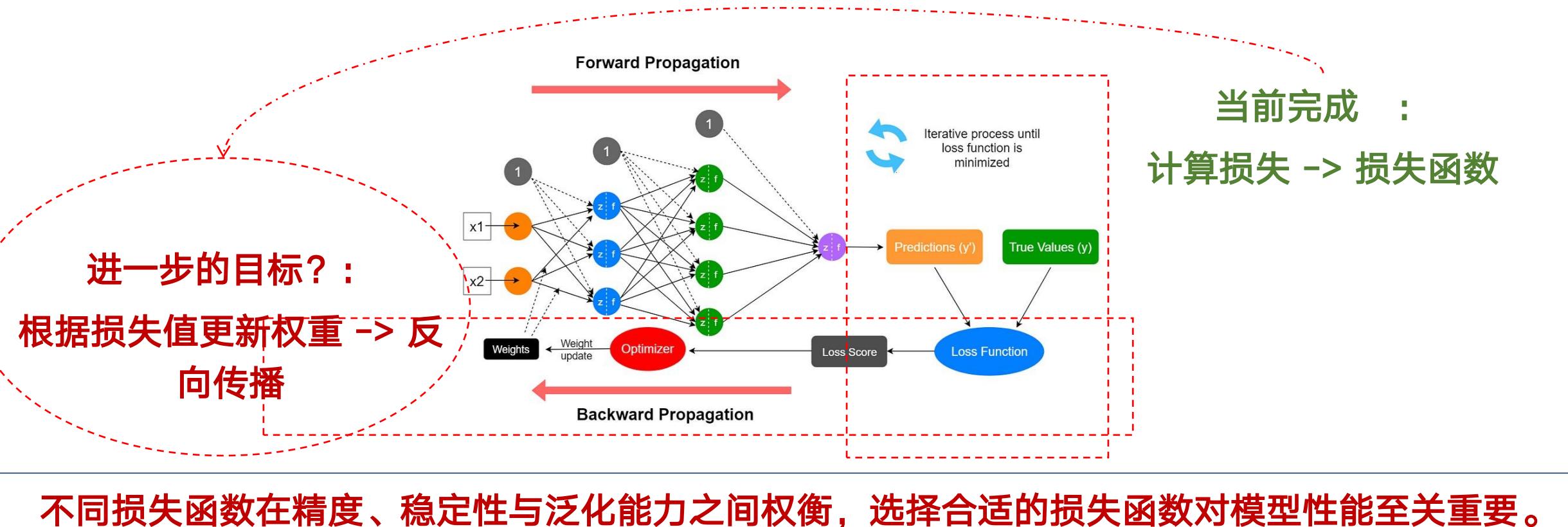
# ▶ 损失函数：扩展

大类	损失函数名称	用途	特点
对比/嵌入/度量	Contrastive Loss	判断样本对是否相似	用于Siamese网络
	Triplet Loss	拉近正样本、推远负样本	用于人类识别、图像检索
	NT-Xent (SimCLR)	对比学习中的标准损失	强化表示学习，无监督
	感知损失 (Perceptual Loss)	图像风格迁移、超分辨	在预训练网络的中间层提取特征差异
生成与图像	L1 / L2 Loss	图像生成基本损失	保证像素精度但不保证视觉感知
	对抗损失 (GAN Loss)	生成对抗网络	包括判别器损失与生成器损失
	Style Loss / Content Loss	风格迁移	分别衡量图像内容和风格的差异
序列与语言	CTC Loss	无需对齐的语音/文本序列训练	用于语音识别
	Dice Loss / IoU Loss	语义分割	处理像素不平衡
其他	Ranking Loss	推荐系统、排序任务	保证排名正确性
	Margin Loss (如 SVM)	二分类	强制类别之间保持间隔

不同损失函数对异常值的敏感程度不同，根据需求选择合适的函数以提高模型鲁棒性和精度。

## ▶ 损失函数：总结

- 损失函数衡量模型预测与真实值的偏差，是监督学习的核心优化目标。
- 回归任务常用 MSE、MAE、Huber 等，关注误差的数值大小与鲁棒性；分类任务则依赖交叉熵、Focal Loss 等，关注概率匹配与类别不平衡。



# 目录章节

---

CONTENTS

01

神经网络：当前挑战

02

神经网络：损失函数

03

神经网络：反向传播算法

04

神经网络：算法实现

05

总结

## ▶ 反向传播的历史背景

- 历史背景：20世纪70–80年代，神经网络已被提出，但多层网络训练困难，因无法高效计算深层参数的梯度。
  - 误差反向传播（Backpropagation）算法由 David E. Rumelhart、Geoffrey Hinton 和 Ronald J. Williams 等人在 1986 年系统提出并推广。

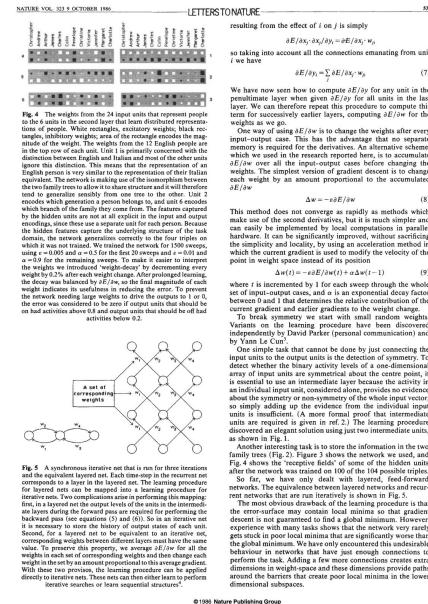
## Learning representations by back-propagating errors

**David E. Rumelhart\*, Geoffrey E. Hinton†  
& Ronald J. Williams\***

\* Institute for Cognitive Science, C-015, University of California  
San Diego, La Jolla, California 92093, USA

<sup>†</sup> Department of Computer Science, Carnegie-Mellon University  
Pittsburgh, Philadelphia 15213, USA

"Learning representations by back-propagating errors"  
David E. Rumelhart, Geoffrey E. Hinton, Ronald J.  
Williams (1986)

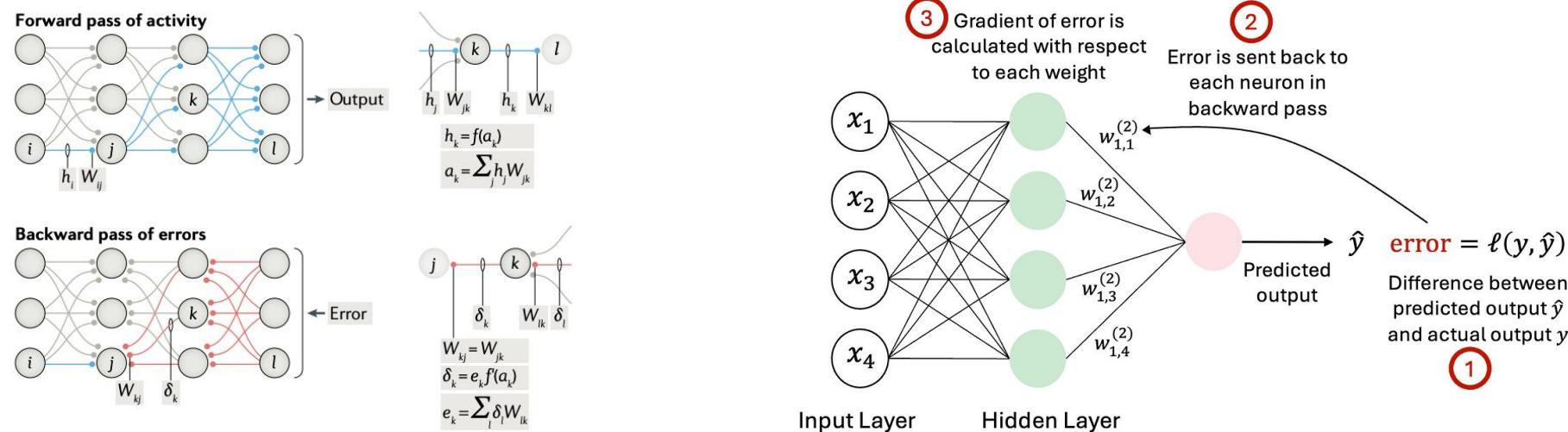


- 这篇论文首次系统性地提出并推广了反向传播算法，使得多层前馈神经网络的训练成为可能，被认为是现代深度学习的奠基之作之一。

反向传播的发明使得模型克服感知机的表达能力限制，让多层网络能自主学习非线性特征表示。

# ▶ 为什么需要反向传播?

- 神经网络的目的是通过调整权重和偏置来最小化损失函数，使预测值越来越接近真实值。
- 目前已知当前的全局误差与损失，那么就需要根据损失去动态调整整个网络的权重，这就需要一种方法来高效地计算每个权重对于当前损失的影响以及动态更新——反向传播。本质就是如果每个神经元的输出变化会让损失函数最终变化多少？反向传播的精妙之处就在于：把全局误差拆分成每一层的局部误差，让每个权重知道自己该怎么调整。



反向传播的目的是通过链式法则高效计算损失函数对每层参数的梯度，以便优化模型参数。

## ▶ 损失函数与误差（局部）

- 损失 (Loss)：把误差通过某种规则量化成一个非负的数，用于衡量模型预测好坏。
- 误差 (Error /  $\delta$ ) 【局部】：表示这一层神经元的输出对全局损失的贡献，也就是“这个神经元导致了多少错误”。通过链式法则，就能从全局误差从输出层一步步传回来。
- 在经典的反向传播推导中，每层神经元误差 $\delta$ 的计算公式定义为：

$$\delta = \frac{\partial L}{\partial z}$$

- 它其实就是损失对神经元输入 $z$ 的导数，代表“**这个神经元的输出对最终损失的影响程度**”。这个偏导越大，说明这个成员“对最终结果负的责任更大”，需要更多调整。
  - 对输出层： $\delta$  直接与损失函数的梯度相关（比如交叉熵的  $\delta = y_{\text{true}} - y_{\text{pred}}$ ）【全局】。
  - 对隐藏层： $\delta$  是通过链式法则，**把后面层的  $\delta$  反向传播过来的**【局部】。

**正是通过链式法则，把最终损失分解到每层，量化了每个神经元对全局损失的贡献。**

## ▶ 全局误差与局部误差

- ◆ 全局误差如何变成局部误差？
- ◆ 我们已经知道在反向传播中，每一层的“误差”其实就是全局损失对改层加权输入 $z$ 的偏导数：

$$\delta = \frac{\partial L}{\partial z}$$

- ◆ 针对第 $l$ 层以及第 $l+1$ 层：

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}, a = f(z^{(l)}) \quad z^{(l+1)} = W^{(l+1)}a^{(l)} + b^{(l+1)}$$

- ◆ 通过链式法则展开：

$$\frac{\partial L}{\partial z^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \odot f'(z^{(l)}) \quad \frac{\partial L}{\partial a^{(l)}} = \frac{\partial L}{\partial z^{(l+1)}} \cdot \frac{\partial z^{(l+1)}}{\partial a^{(l)}} = W^{(l+1)T} \delta^{(l+1)}$$

- ◆ 最终合并得到：

$$\delta^{(l)} = \frac{\partial L}{\partial z^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \odot f'(z^{(l)}), \quad \frac{\partial L}{\partial a^{(l)}} = W^{(l+1)^\top} \delta^{(l+1)}$$

$$\boxed{\delta^{(l)} = (W^{(l+1)^\top} \delta^{(l+1)}) \odot f'(z^{(l)})}$$

局部误差是通过链式法则，把全局误差按权重比例和激活梯度层层分摊回去的。

## ► 什么是反向传播？

- 定义：反向传播是一种用于训练前馈神经网络的高效梯度计算算法，其核心思想是利用**链式法则（chain rule）**，从损失函数对输出层的偏导数出发，逐层递归地计算损失函数关于每个可训练参数的梯度，以便于使用梯度下降等优化算法对模型进行参数更新。

◆ 形式上，设损失函数为 $L$ ，神经网络由若干函数复合组成：

$$\hat{y} = f_n(f_{n-1}(\cdots f_1(x; \theta_1) \cdots; \theta_{n-1}); \theta_n) \rightarrow L(y, \hat{y})$$

◆ 反向传播可以通过链式法则计算得到每一层参数 $\theta$ 的梯度：

$$\frac{\partial L(y, \hat{y})}{\partial \theta_i} = \frac{\partial L}{\partial f_n} \cdot \frac{\partial f_n}{\partial f_{n-1}} \cdots \frac{\partial f_{i+1}}{\partial f_i} \cdot \frac{\partial f_i}{\partial \theta_i}$$

- 为什么反向传播需要计算梯度呢？ -> 反向传播的目标是通过调整神经网络参数最小化损失函数，而**梯度恰好描述了损失函数对每个参数的变化率与方向，计算梯度后我们才能用梯度下降等方法沿着最能降低损失的方向更新参数**，从而让模型逐步逼近最优解。

**支持多层结构的高效训练，是深度学习模型（如 CNN、RNN、Transformer 等）可行的关键。**

## ▶ 为什么反向传播需要计算梯度？

- ▶ 假设我们有一个非常简单的神经网络，以及均方误差，我们的目标就是通过调整w和b让预测值更加接近真实值：

$$\hat{y} = wx + b$$
$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(wx + b - y)^2$$

- ▶ 如果不计算梯度 -> 不知道w,b到底是增大还是减小，可能改了之后误差反而还会变大，方向错误，而计算梯度之后：

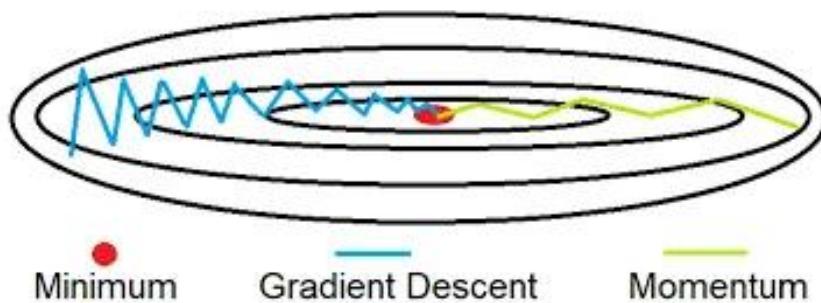
$$\frac{\partial L}{\partial w} = (wx + b - y) \cdot x \quad \frac{\partial L}{\partial b} = (wx + b - y)$$

- ▶ 该偏导数说明当前参数变化会对损失产生多大的影响（导数的绝对值），应该往哪个方向调整（导数的正负号）。如果是负数，说明需要增大w和b，反之减小w和b。
- ▶ 最后，利用梯度更新参数，这样每次参数就都会朝着让损失减少的方向前进，最终收敛到最优解附近。

**总结：梯度告诉我们损失函数对每个参数的变化方向和大小，反向传播正是用它来指导参数更新，从而让模型朝降低损失的方向收敛。**

## ► 扩展：为什么要在损失最大的方向下降？

- 在数学上可以证明：在任意一点，**梯度的方向就函数上升最快的方向，它的反方向就是函数下降最快的方向**。如果你站在山坡上，梯度就是最陡上坡的方向，而负梯度就是最陡下坡的方向，选择这个方向能保证单位步长下降最多。
- **能不能偏离梯度一点？** 可以，很多优化方法（比如共轭梯度法、牛顿法、动量法、Adam等）就是在负梯度附近调整方向，利用曲率或历史信息让下降更快、更稳定。但如果你完全随便偏离，可能出现两种问题：下降效率变差（走了不陡的方向，损失下降得很慢）；可能震荡或跑偏（如果方向偏得太多，甚至会离开下降区域）。

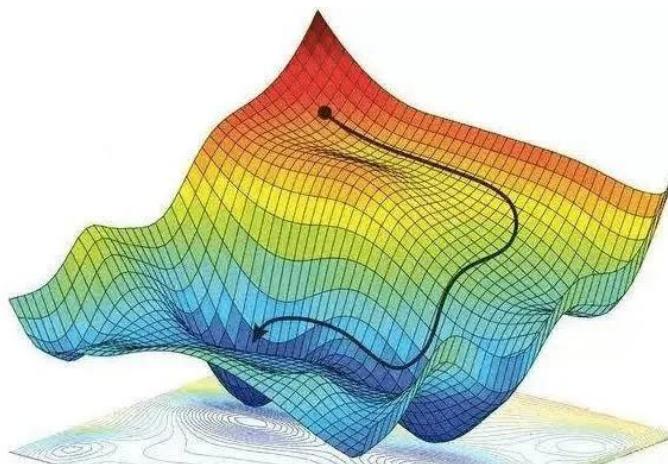


最简单可靠的策略就是：**先沿着负梯度走（最快下降），再通过学习率和改进优化器控制步幅和方向。**

**负梯度方向是最快下降的方向，它能保证每一步损失减少最多；如果想“稍微偏离”，那就需要更复杂的优化方法来保证效率和收敛性。**

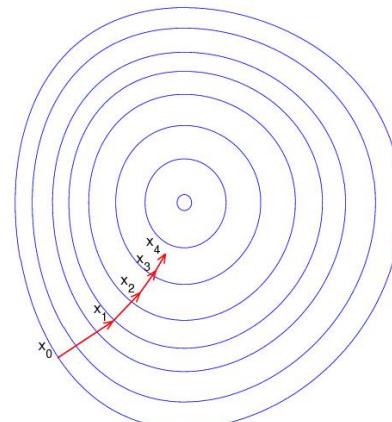
## ► 梯度下降法：定义

- 我们目前已知这些梯度，我们该怎么用它们来调整参数，使模型更准确呢？
- 注意：反向传播的核心是通过链式法则，快速计算出模型中每个参数对损失的敏感程度，也就是梯度。但计算出梯度只是第一步，我们还需要根据梯度来调整参数。
- 梯度下降法——一种根据梯度信息不断更新参数，逐步降低损失的优化方法。梯度下降方法基于以下的观察：如果实值函数 $F(x)$  在某点处可微分且有定义，那么函数 $F(x)$ 在该点沿着梯度相反的方向。



source:

<https://zh.wikipedia.org/wiki/%E6%A2%AF%E5%BA%A6%E4%B8%8B%E9%99%8D%E6%B3%95>



因而，如果：

$$b = a - \gamma \nabla F(a)$$

对于一个任意小的 $\gamma > 0$ 时成立，那么：

$$F(a) > F(b)$$

从函数的某一点除法，并考虑如下序列 $x_0, x_1, x_2 \dots$ 使得：

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n), n \geq 0$$

就可以得到：  $F(x_0) \geq F(x_1) \geq F(x_2) \geq \dots$

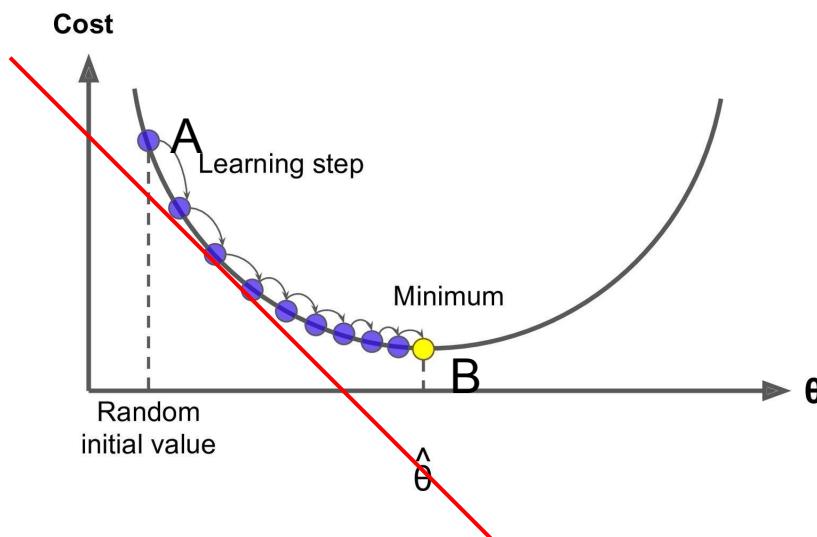
如果顺利的话序列 $x_n$ 就会收敛到期望的**局部极小值**。

注意每次迭代步长 $\gamma$ 可以改变（后续提到的优化器）。 31

## ► 梯度下降法：直观理解

- 直观理解：先随机“猜一个值”，然后根据反馈进行调整。类比下山过程，我们把 loss 看出是曲线就是山谷，如果走过来就再往回调，所以是一个迭代的过程。
- 这里的  $x_{n+1}$  就根据  $x_n$ 、梯度以及  $\gamma$  所迭代。 $\gamma$  又称为学习率（learning rate） $[\gamma > 0]$ ，可以看作是下山迈的步子的大小，步子迈的大下山就快，即调整的幅度、震荡力度。

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n), n \geq 0$$



source:  
<https://www.cnblogs.com/ngiws719/p/16424497.html>

问题：这个公式是怎么控制一定是朝向局部最小值逼近迭代的呢？

假设初始点为 A，目标点 B 为局部最小值点，我们可以知道，A 点向 B 点移动的过程中，梯度（斜率）始终小于 0，即：

$$\nabla F(x_n) < 0$$

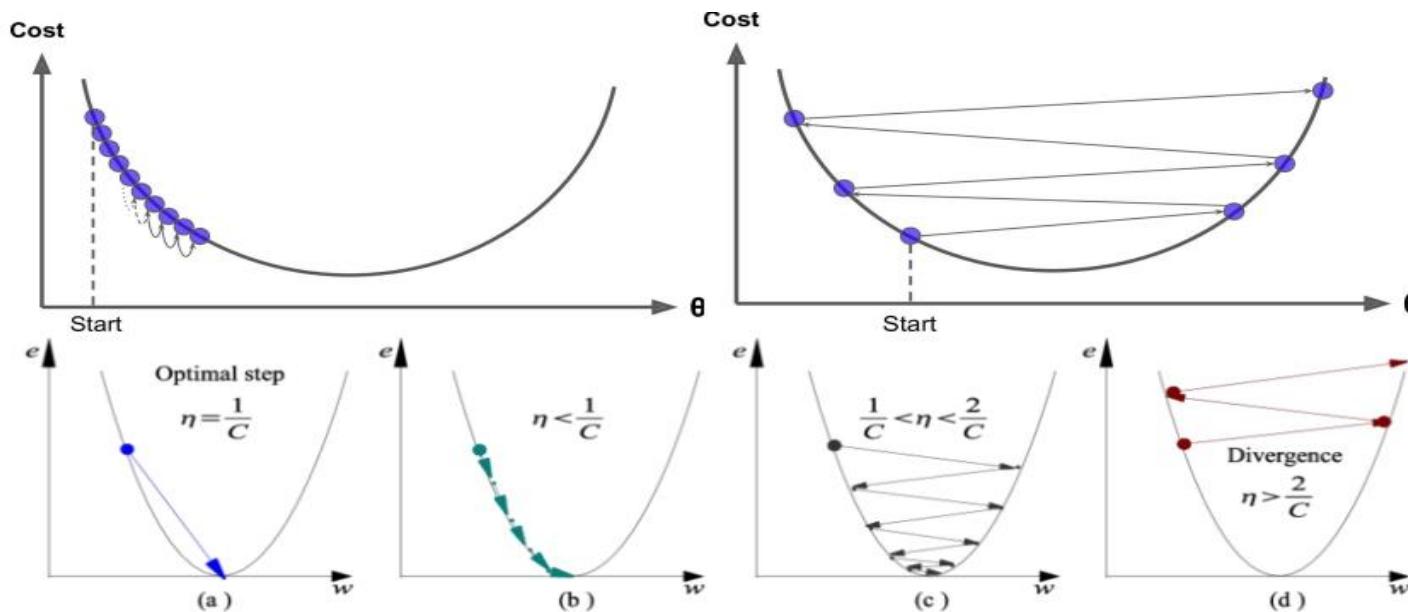
由于  $\gamma > 0$ ，那么：

$$x_{n+1} > x_n$$

说明迭代的过程是向右进行的，是在往局部最小值的方向逼近。反之亦然。

## ▶ 梯度下降法：学习率

- 学习率如何调整？考虑每次迭代过程中固定 $\gamma$ 不变的情况，每次调整的幅度、迈出的步伐一定，但是如果 $\gamma$ 过大或者过小会出现什么情况？
- $\gamma$ 过大  $\rightarrow$  矫枉过正，迭代过程会来回震荡，甚至 越来越偏离局部最小值。
- $\gamma$ 过小  $\rightarrow$  矫枉不够，迭代幅度太小，相同迭代次数情况下，靠近局部最小值较远。



问题：为什么需要学习率？  
梯度本身只是告诉我们“**方向和相对大小**”，它**并没有规定我们走多远**。因此引入一个超参数 $\gamma$ （学习率）来控制更新步长。

**学习率决定了参数更新的步幅：过大会导致震荡甚至发散，过小则收敛缓慢，因此需在稳定性与收敛速度之间权衡，并常结合自适应或动态调整策略。**

## ► 权重更新（梯度下降的意义）

➤ 有了梯度下降法之后，权重应该怎么更新呢？

● 权重更新公式：

$$w' = w - \gamma \frac{\partial L}{\partial w}$$

● 用链式法则展开后：

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w} = \delta \cdot a_{\text{prev}} \quad \frac{\partial z}{\partial w} = \frac{\partial(w \cdot a_{\text{prev}} + b)}{\partial w} = a_{\text{prev}}$$

● 这里的 $\delta$ 就是梯度在该层神经元的误差， $a_{\text{prev}}$ 是前一层的激活值。

● 所以更新公式变成：

$$w' = w - \gamma \delta a_{\text{prev}}$$

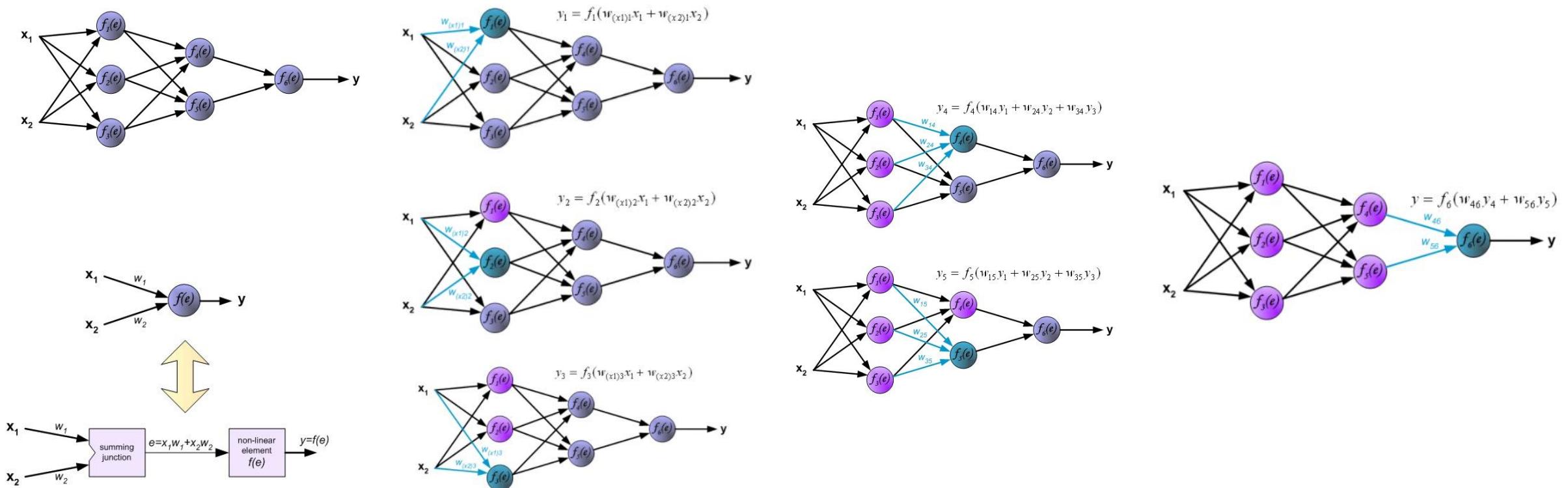
注：所谓的“误差  $\delta$ ”就是梯度的一部分：它是损失对神经元输入的偏导。

我们平时说的“误差反传”，本质上就是把梯度分解（链式法则）后层层传递，每一层根据  $\delta$  和前一层的激活去算自己的梯度。

权重更新 = 学习率  $\times$  当前神经元的误差  $\times$  前一层的激活值， $\gamma$ 像“调整的速度”：控制收敛节奏。 $\delta$  像“责任分摊”：这个神经元要改多少。 $a_{\text{prev}}$ 像“传话力度”：输入信号越强，连接它的权重越值得调整。

# ▶ 反向传播：完整案例

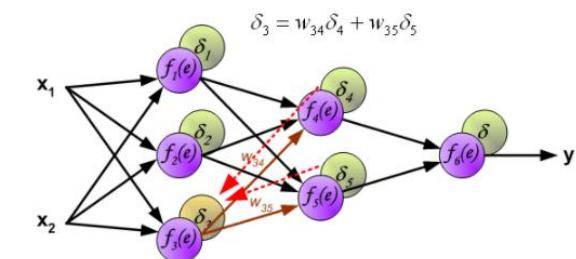
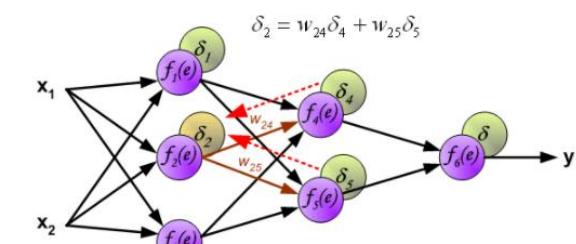
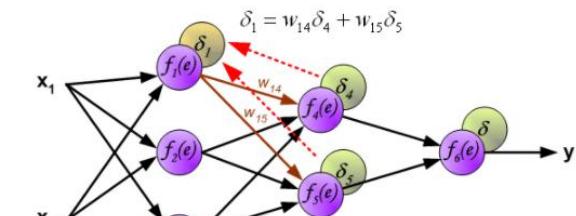
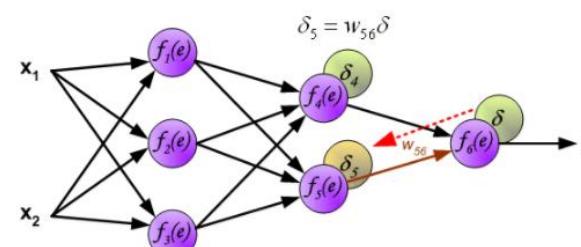
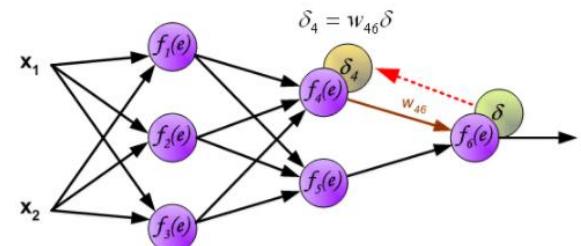
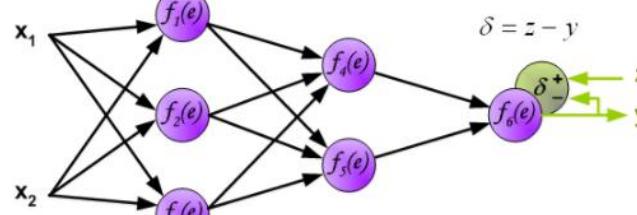
- 目标：用一个简单的 2 层神经网络，观察前向传播后怎么通过反向传播进行权重更新。
- 1) 前向传播过程：



source: <https://www.jiqizhixin.com/graph/technologies/7332347c-8073-4783-bfc1-1698a6257db3>

# ▶ 反向传播：完整案例

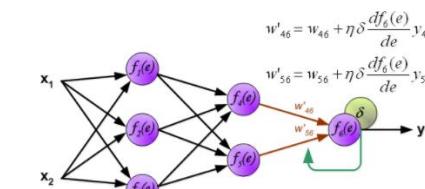
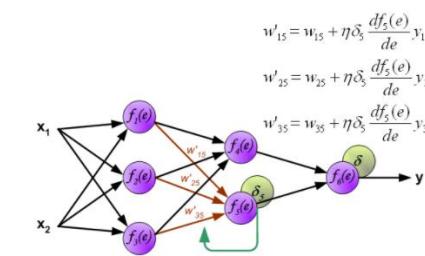
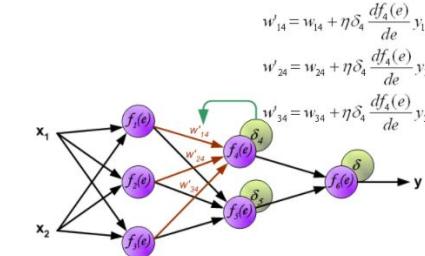
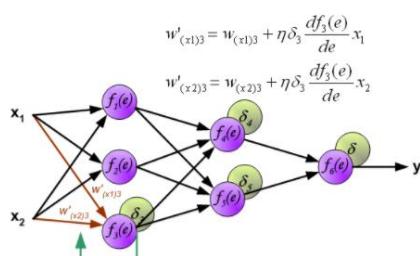
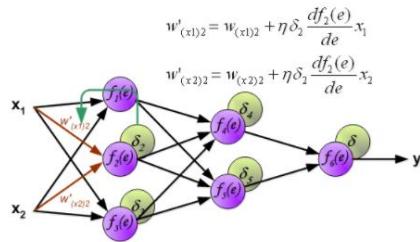
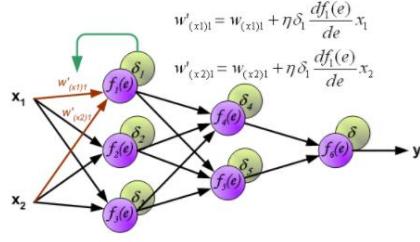
- 目标：用一个简单的 2 层神经网络，观察前向传播后怎么通过反向传播进行权重更新。
- 2) 反向传播计算损失过程：



source: <https://www.jiqizhixin.com/graph/technologies/7332347c-8073-4783-bfc1-1698a6257db3>

# ▶ 反向传播：完整案例

- 目标：用一个简单的 2 层神经网络，观察前向传播后怎么通过反向传播进行权重更新。
- 3) 权重更新过程：



source: <https://www.jiqizhixin.com/graph/technologies/7332347c-8073-4783-bfc1-1698a6257db3>

## ▶ 前向传播 vs 反向传播

- 前向传播是预测阶段，将输入数据一层层处理得到输出。
- 反向传播是学习阶段，根据损失反向计算梯度，更新模型参数。

项目	前向传播 (Forward Propagation)	反向传播 (Backpropagation)
作用	计算网络输出（预测）	计算损失对参数的梯度，用于学习
方向	输入 -> 输出	输出 -> 输入
输入内容	输入数据（特征）	损失函数的导数（对输出误差的敏感度）
输出内容	模型的预测结果	各层参数的梯度
依赖函数	激活函数	激活函数的导数
关键步骤	逐层线性变换 + 激活函数	使用链式法则逐层反向计算梯度
是否更新参数	否	是（结合优化器如SGD、Adam等）
是否用到损失函数	否	是（从损失开始计算梯度）

前向传播用于计算模型输出和损失，反向传播用于计算损失对参数的梯度以指导优化。

# 目录章节

---

CONTENTS

01

神经网络：当前挑战

02

神经网络：损失函数

03

神经网络：反向传播算法

04

神经网络：算法实现

05

总结

## ► 算法实现 (numpy)

- 目标：理解神经网络的训练步骤。
- 大致流程：前向传播 → 计算损失 → 反向传播（求梯度）→ 更新权重。
- 1) 前向传播（激活函数为Sigmoid）：

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}, a = f(z^{(l)})$$

- 2) 计算损失函数 (MSE)：

$$L(y, \hat{y}) = \frac{1}{n} \sum (y - \hat{y})^2$$

- 3) 反向传播：

$$\delta^{(L)} = \nabla_a L \odot f'(z^{(L)})$$

计算输出层误差

$$\delta^{(l)} = (W^{(l+1)T} \delta^{(l+1)}) \cdot f'(z^{(l)})$$

逐层传播

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

梯度计算

- 4) 更新权重：

$$W^{(l)} \leftarrow W^{(l)} - \gamma \frac{\partial L}{\partial W^{(l)}}$$

注：偏置值b的计算也类似

# ▶ 算法实现 (numpy)

## ➤ 算法伪装代码：

输入：训练数据  $X, y$ , 学习率  $\gamma$ , 迭代次数 epochs  
初始化：权重  $W$  和偏置  $b$

循环 epoch:

1. 前向传播：计算每层  $z$ 、 $a$
2. 计算损失： $L(y, \hat{y})$
3. 反向传播：

- 计算输出层  $\delta$
- 逐层传播  $\delta$
- 计算梯度  $dW, db$

4. 更新参数：

$$W \leftarrow W - \gamma * dW$$
$$b \leftarrow b - \gamma * db$$

输出：训练好的权重  $W, b$

```
import numpy as np

# 设置随机种子
np.random.seed(43)

# 激活函数及其导数
def sigmoid(x): return 1 / (1 + np.exp(-x))
def sigmoid_deriv(x): return sigmoid(x) * (1 - sigmoid(x))

# 初始化参数
W1 = np.random.randn(2, 3) # shape: (2, 3) 输入层->隐藏层
b1 = np.zeros((1, 3)) # 广播机制
W2 = np.random.randn(3, 1) # shape: (3, 1) 隐藏层->输出层
b2 = np.zeros((1, 1)) # 广播机制
lr = 0.1

# 前向传播
def forward(X):
    z1 = X @ W1 + b1
    a1 = sigmoid(z1)
    z2 = a1 @ W2 + b2
    a2 = sigmoid(z2)
    return z1, a1, z2, a2

# 损失 (MSE)
def mse(y, y_hat): return np.mean((y - y_hat)**2)

# 反向传播
def backward(X, y, z1, a1, z2, a2):
    m = y.shape[0]
    delta2 = (a2 - y) * sigmoid_deriv(z2)
    dW2 = a1.T @ delta2 / m
    db2 = np.sum(delta2, axis=0, keepdims=True) / m

    delta1 = (delta2 @ W2.T) * sigmoid_deriv(z1)
    dW1 = X.T @ delta1 / m
    db1 = np.sum(delta1, axis=0, keepdims=True) / m
    return dW1, db1, dW2, db2

# 训练
X = np.array([[0,0],[0,1],[1,0],[1,1]]) # shape: (4, 2)
y = np.array([[0],[1],[1],[0]]) # XOR
for epoch in range(10000):
    z1,a1,z2,a2 = forward(X)
    dW1,db1,dW2,db2 = backward(X,y,z1,a1,z2,a2)
    W1 -= lr*dW1; b1 -= lr*db1
    W2 -= lr*dW2; b2 -= lr*db2
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}: Loss = {mse(y,a2):.4f}")
```

$$\begin{aligned}\sigma(x) &= \frac{1}{1 + e^{-x}} \\ \frac{\partial \sigma(x)}{\partial x} &= \frac{\partial}{\partial x} (1 + e^{-x})^{-1} \\ &= -1 \cdot (1 + e^{-x})^{-2} \cdot \frac{\partial}{\partial x} (1 + e^{-x}) \\ &= -1 \cdot (1 + e^{-x})^{-2} \cdot (-e^{-x}) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{1 + e^{-x} - 1}{(1 + e^{-x})^2} \\ &= \frac{1}{1 + e^{-x}} \left( 1 - \frac{1}{1 + e^{-x}} \right) \\ &= \sigma(x)(1 - \sigma(x))\end{aligned}$$

Epoch 0: Loss = 0.3741  
Epoch 1000: Loss = 0.2462  
Epoch 2000: Loss = 0.2399  
Epoch 3000: Loss = 0.2296  
Epoch 4000: Loss = 0.2139  
Epoch 5000: Loss = 0.1918  
Epoch 6000: Loss = 0.1599  
Epoch 7000: Loss = 0.1168  
Epoch 8000: Loss = 0.0758  
Epoch 9000: Loss = 0.0487

# ► 算法实现练习 (pytorch)

## ➤ 算法伪装代码：

输入：训练数据  $X, y$ , 学习率  $\gamma$ , 迭代次数 epochs  
初始化：权重  $W$  和偏置  $b$

循环 epoch:

1. 前向传播：计算每层  $z, a$
2. 计算损失： $L(y, \hat{y})$
3. 反向传播：
  - 计算输出层  $\delta$
  - 逐层传播  $\delta$
  - 计算梯度  $dW, db$
4. 更新参数：
$$W \leftarrow W - \gamma * dW$$
$$b \leftarrow b - \gamma * db$$

输出：训练好的权重  $W, b$

```
import torch

# 设置随机种子
torch.manual_seed(666)

# 激活函数及其导数
def sigmoid_torch(x):
    pass
def sigmoid_deriv_torch(x):
    pass

W1 = torch.randn((2, 3), dtype=torch.float32) # shape: (2, 3)
b1 = torch.zeros((1, 3), dtype=torch.float32) # shape: (1, 3)
W2 = torch.randn((3, 1), dtype=torch.float32) # shape: (3, 1)
b2 = torch.zeros((1, 1), dtype=torch.float32) # shape: (1, 1)
lr = 0.1

# 前向传播
def forward(x):
    pass

# 损失 (MSE)
def mse(y, y_hat):
    pass

# 反向传播
def backward(X, y, z1, a1, z2, a2):
    pass

# 训练数据
X = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]], dtype=torch.float32)
y = torch.tensor([[0.], [1.], [1.], [0.]], dtype=torch.float32)

# 训练
for epoch in range(10000):
    z1, a1, z2, a2 = forward(X)
    dW1, db1, dW2, db2 = backward(X, y, z1, a1, z2, a2)
    W1 -= lr * dW1; b1 -= lr * db1
    W2 -= lr * dW2; b2 -= lr * db2
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}: Loss = {mse(y, a2).item():.4f}")
```

# ► 算法实现 (pytorch封装)

## ➤ 算法伪装代码：

输入：训练数据  $X, y$ , 学习率  $\gamma$ , 迭代次数 epochs  
初始化：权重  $W$  和偏置  $b$

循环 epoch:

1. 前向传播：计算每层  $z, a$
2. 计算损失： $L(y, \hat{y})$
3. 反向传播：

- 计算输出层  $\delta$
- 逐层传播  $\delta$
- 计算梯度  $dW, db$

4. 更新参数：

$$W \leftarrow W - \gamma * dW$$
$$b \leftarrow b - \gamma * db$$

输出：训练好的权重  $W, b$

```
import torch
import torch.nn as nn

# 设置随机种子
torch.manual_seed(666)

# 数据
X = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]], dtype=torch.float32)
y = torch.tensor([[0.], [1.], [1.], [0.]], dtype=torch.float32)

# 定义模型
class XORNet(nn.Module):
    def __init__(self):
        super(XORNet, self).__init__()
        self.hidden = nn.Linear(2, 3)
        self.output = nn.Linear(3, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.hidden(x))
        x = self.sigmoid(self.output(x))
        return x

model = XORNet()

# 损失函数
criterion = nn.MSELoss()
lr = 0.1 # 学习率
# optimizer = torch.optim.SGD(model.parameters(), lr=lr) # 使用SGD优化器

# 训练 (手动更新参数)
for epoch in range(10000):
    y_pred = model(X)
    loss = criterion(y_pred, y)

    # 反向传播
    model.zero_grad()
    loss.backward()

    # 手动更新参数
    with torch.no_grad():
        for param in model.parameters():
            param -= lr * param.grad

    # 反向传播 & 参数更新
    # optimizer.zero_grad()
    # loss.backward()
    # optimizer.step()

    if epoch % 1000 == 0:
        print(f"Epoch {epoch}: Loss = {loss.item():.4f}")

# 预测
print("Predictions:", model(X).detach().numpy())
```

# 目录章节

---

CONTENTS

01

神经网络：当前挑战

02

神经网络：损失函数

03

神经网络：反向传播算法

04

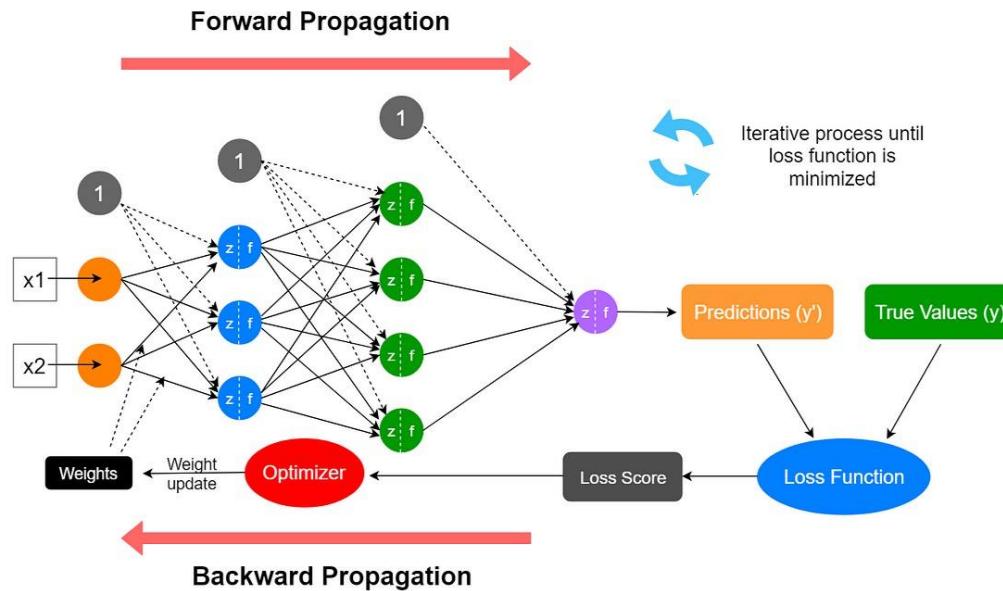
神经网络：算法实现

05

总结

## ▶ 总结

- 损失函数：衡量模型预测与真实结果的差距，指导参数优化方向。常用的损失函数：均方误差（MSE）、交叉熵损失等。
- 反向传播算法：基于链式法则，逐层计算梯度，高效更新神经网络参数，使模型逐步逼近最优解。



损失函数定义了优化目标，反向传播计算梯度实现目标优化。

# 感谢聆听



Personal Website: <https://www.miaopeng.info/>



Email: [miaopeng@stu.scu.edu.cn](mailto:miaopeng@stu.scu.edu.cn)



Github: <https://github.com/MMeowhite>



Youtube: <https://www.youtube.com/@pengmiao-bmm>