2.Training Procedure Outline

Model Architecture

A Single Layer of Neurons(Perceptron)

在神经网络和深度学习的语境中，单个神经元（Single Neuron）是构成神经网络的基本单元。每个神经元接收来自其他神经元的输入信号，对这些信号进行加权求和，然后通过一个激活函数（Activation Function）处理，最终产生一个输出信号。这个输出信号可以作为其他神经元的输入，或者作为整个网络的最终输出。

激活函数是神经网络中非常关键的部分，它引入了非线性因素，使得神经网络能够学习和表示复杂的函数关系。没有激活函数，神经网络将只能表示线性关系，这极大地限制了其应用范围。σ(z)是激活函数的一种表示方式，其中σ通常代表特定的激活函数，而z是神经元的加权输入和偏置（bias）的和，即z = wx + b，其中w是权重（weights），x是输入（inputs），b是偏置（bias）。

Sigmoid函数（Sigmoid Function）是激活函数的一种，它的数学表达式为：

$$\sigma(z) = 1/(1 + e^{(} - z))$$

Sigmoid函数将任意实数值映射到(0, 1)区间内，这个特性使得它非常适合用于二分类问题的输出层。当z趋近于正无穷时，σ(z)趋近于1；当z趋近于负无穷时，σ(z)趋近于0。因此，Sigmoid函数的输出可以被解释为概率值，用于表示某个事件发生的可能性。

在神经网络中，偏置项（bias term）确实可以被视为一个“始终激活”或“始终开启”的特征。这是因为偏置项不依赖于任何输入数据，而是作为一个固定的参数存在于每个神经元中。它的主要作用是调整神经元的输出阈值，从而增加模型的灵活性和表达能力。具体来说，偏置项允许神经元在没有任何输入信号的情况下也有一个非零的输出。这可以通过将偏置项加到神经元的加权输入和上来实现。因此，即使所有的输入信号都是零，由于偏置项的存在，神经元的输出也可以是非零的。这种特性使得偏置项成为神经网络中不可或缺的一部分，它有助于模型捕捉到数据中的非线性关系和复杂模式。在神经网络的训练过程中，bias会像权重一样被学习和调整。通过调整偏置项，神经网络可以更好地拟合训练数据，从而提高模型的准确性和泛化能力。

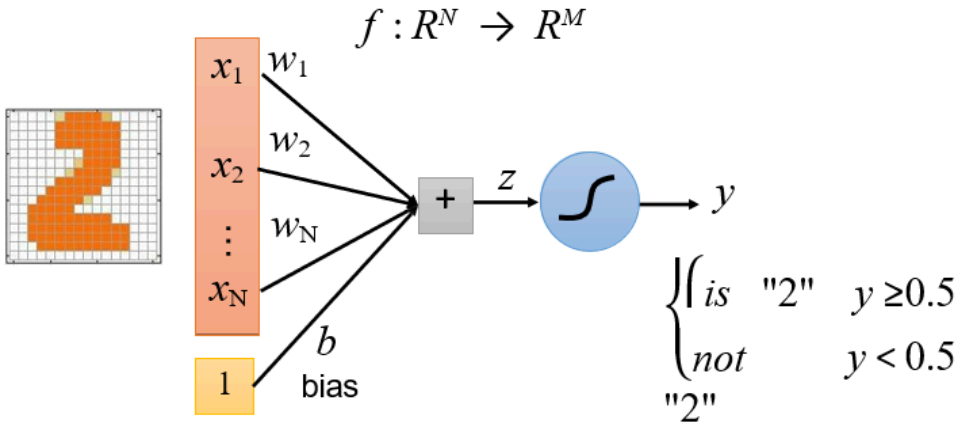
在机器学习和神经网络中，偏置项（bias term）的作用确实与“类先验”（class prior）有一定的联系，但这种联系通常是在特定的上下文或解释中才显得明显。首先，让我们澄清这两个概念：

1.偏置项（Bias Term）：在神经网络中，每个神经元都有一个或多个偏置项。偏置项是一个可学习的参数，它不与任何输入特征相乘，而是直接加到神经元的加权输入和上。偏置项的存在使得

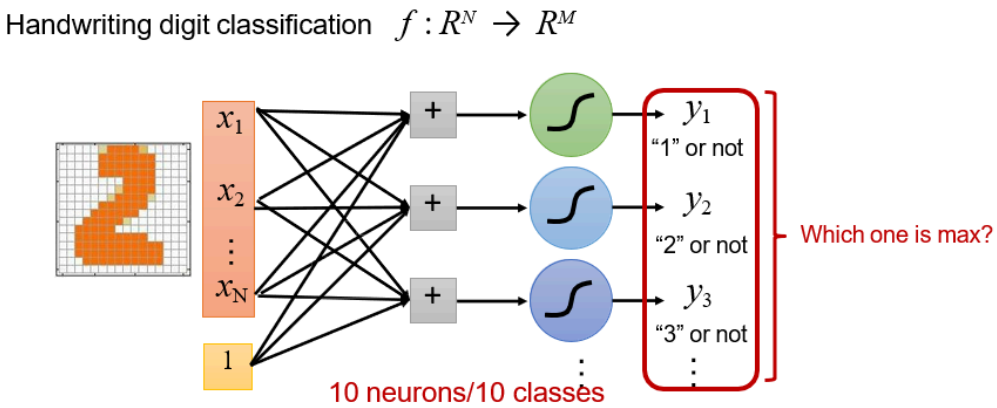
神经元的输出可以在没有输入信号的情况下也有一个非零的基准值。

2.类先验（Class Prior）：在分类任务中，类先验通常指的是每个类别在训练数据集中出现的相对频率。例如，在一个二分类问题中，如果正面类别的样本占60%，负面类别的样本占40%，那么正面类别的先验概率就是0.6，负面类别的先验概率就是0.4。

以下是perceptron在实现分类任务，最后按照outputs的值来划分分类结果。



A single neuron can only handle binary classification



A layer of neurons can handle multiple possible output, and the result depends on the max one

---

Limitations of Perceptron

感知器 (Perceptron)

感知器是所有神经网络中最基本的模型之一，由 Frank Rosenblatt 在 1957 年提出。它是一种**单层神经网络**，通常用于二分类的线性分类任务。感知器的结构非常简单，只包含输入层和输出

层，没有隐藏层。

结构

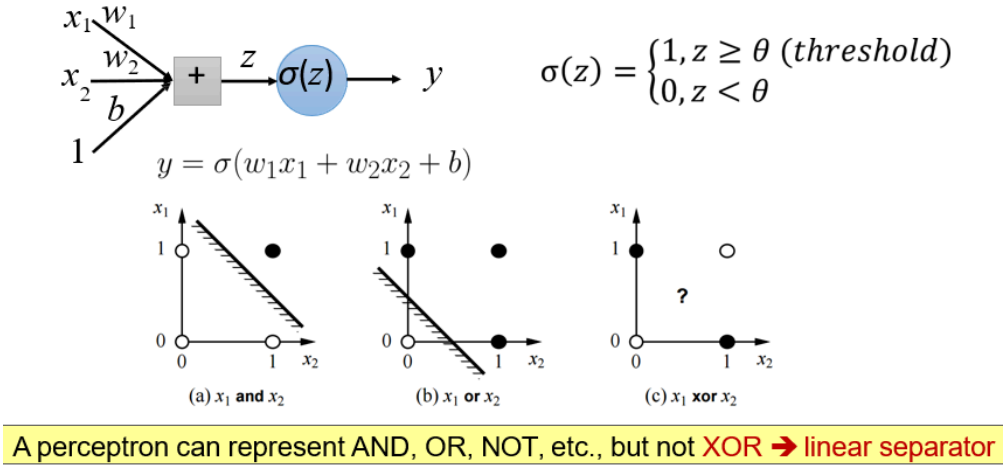
1. **输入层**
   - 接收外部输入信号
   - 不进行计算，仅传递信号到输出层
2. **输出层**
   - 对输入信号进行加权求和并加上偏置值
   - 通过激活函数（如阶跃函数）生成输出

数学表达

感知器可以表示 **AND**、**OR**、**NOT** 等线性可分逻辑运算，但：

❌ 无法直接表示 XOR 运算

- 原因：XOR 是非线性可分问题（如下图）
- 解决方案：需使用多层感知机（MLP）实现： A XOR B= A'B + AB'



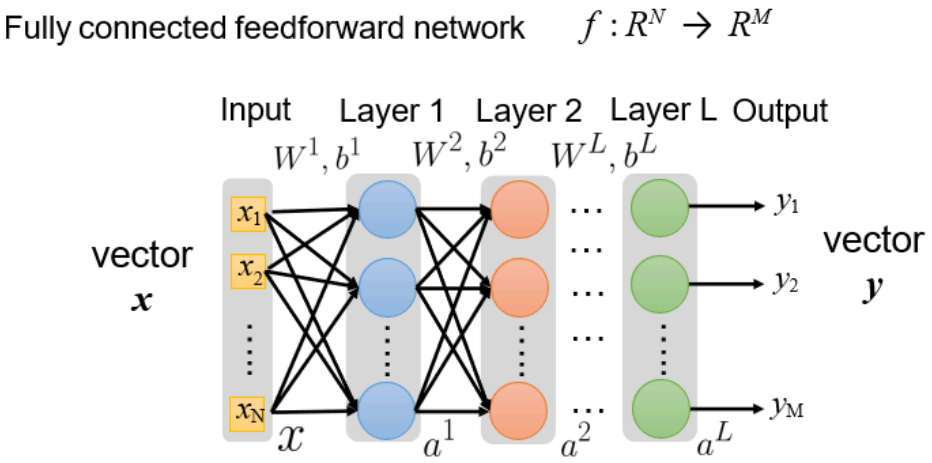
图：XOR 问题的线性不可分性

Neural Network Model (Multi-Layer Perceptron)

多层感知器的结构

1.输入层：接收输入数据，不进行计算，只是将信号传递给下一层。
2.隐藏层：一个或多个隐藏层，每个隐藏层包含多个神经元。每个神经元对输入信号进行加权求和，加上偏置值，然后通过激活函数进行非线性变换。
3.输出层：接收隐藏层的输出，进行加权求和，加上偏置值，然后通过激活函数输出最终结果。

结构如图：



$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) ... + b^L)$$

激活函数（Activation Function）

1.定义（Activation Function）：
在神经网络中，激活函数是用来决定一个节点（或神经元）是否应该被激活的函数。它给网络的输出增加了非线性因素，使得神经网络能够学习和模拟复杂的函数关系。常见的激活函数包括Sigmoid、Tanh、ReLU（Rectified Linear Unit）等。激活函数的选择对网络的性能和训练效率有很大影响。

2.布尔值（Boolean）：
布尔值是一种逻辑数据类型，只有两个可能的值：真（True）和假（False）。布尔值在计算机科学和编程中广泛使用，尤其是在条件判断和控制流程语句中。例如，在if语句中，可以根据布尔值来决定是否执行某段代码。

3.线性（Linear）：
线性关系指的是两个变量之间的关系可以表示为一条直线。在数学上，线性函数可以表示为y = ax + b的形式，其中a和b是常数，x和y是变量。线性关系简单且易于分析，但在很多实际问题中，变量之间的关系往往是非线性的。

4.非线性（Non-linear）：
非线性关系指的是两个变量之间的关系不能用一条直线来表示。非线性关系可能表现为曲线、曲面或其他更复杂的形状。非线性系统通常比线性系统更难分析，但在实际生活中，很多现象都是非线性的。在机器学习中，非线性激活函数使得神经网络能够捕捉和表示数据中的复杂关系。

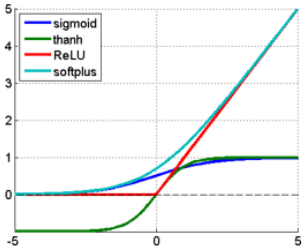
常见的非线性激活函数（Non-Linear Activation Function）

- Sigmoid

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

- Tanh

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$



- Rectified Linear Unit (ReLU)

$$\text{ReLU}(x) = \max(x, 0)$$

Non-linear functions are frequently used in neural networks

非线性激活函数的作用

1.表达能力增强：

线性函数只能表示输入数据的线性组合，这意味着无论神经网络的层数有多少，其最终输出都将是输入的线性变换。然而，现实世界中的数据往往呈现出复杂的非线性关系。通过引入非线性激活函数，神经网络能够捕捉到这些非线性关系，从而极大地增强了其表达能力。

2.分层特征提取：

在深度学习中，每一层神经网络都可以被视为对数据的一种特征提取器。非线性激活函数使得每一层都能够学习到数据的不同非线性特征，这些特征在后续层中可以被进一步组合和抽象，以形成更高级别的表示。这种分层特征提取的能力是深度学习模型能够处理复杂任务的关键。

3.避免退化：

如果没有非线性激活函数，多层神经网络将退化为单层线性模型。这是因为多层线性模型的输出仍然是输入的线性组合，而线性组合可以通过单层模型轻松实现。非线性激活函数的引入打破了这种退化，使得每一层都能够为最终输出贡献独特的非线性特征。

4.梯度消失与爆炸的缓解：

虽然非线性激活函数本身并不直接解决梯度消失或爆炸问题，但某些激活函数（如ReLU及其变体）在设计中考虑到了这些问题，并试图通过其特性来减轻它们的影响。例如，ReLU函数在输入为正时具有恒定的梯度，这有助于缓解梯度消失问题。

5.无非线性时的情况&有非线性时的情况

当深度神经网络中没有非线性激活函数时，整个网络将退化为一个线性变换器。具体来说，如果每一层的输出都是输入的线性组合（即没有非线性激活函数进行转换），那么无论网络有多少层，其最终输出都可以表示为输入的线性组合。这种情况下，网络的表达能力将非常有限，无法捕捉数据中的复杂非线性关系。

当深度神经网络中引入非线性激活函数时，网络的表达能力将显著增强。非线性激活函数使得每一层的输出不再是输入的线性组合，而是经过非线性变换后的结果。这种非线性变换允许网络捕捉到数据中的复杂非线性关系，并逐层抽象出更高层次的特征。随着网络层数的增加，其能够近似的函数复杂度也将增加。具体来说，具有更多层的网络可以组合更多的非线性特征，从而近似更复杂的函数。这种能力使得深度神经网络能够处理各种复杂的任务，如图像识别、语音识别、自然语言处理等。

Loss Function Design

Function = Model Parameters

具体解释如图：

$$y = f(x) = \sigma(W^L \cdots \sigma(W^2\sigma(W^1 x + b^1) + b^2) \ldots + b^L)$$

function set　　different parameters W and b ➔ different functions

- Formal definition

$f(x;\theta)$ model parameter set

$$\theta = \left\{W^1, b^1, W^2, b^2, \cdots W^L, b^L\right\}$$

pick a function f = pick a set of model parameters θ

Model Parameter Measurement

模型参数衡量

模型参数是指在数学模型中用于描述系统特性和行为的数值。这些参数可以是固定的常数，也可以是随时间或其他变量变化的函数。在机器学习中，模型参数是通过训练数据来估计的，目标是找到一组最优的参数，使得模型能够最好地拟合数据并具有良好的泛化能力。那么如何衡量模型参数的好坏？

Define a function to measure the quality of a parameter set θ

- Evaluating by a loss/cost/error function $C(\theta)$ ➔ how bad θ is
- Best model parameter set

$$\theta^* = \arg\min_{\theta} C(\theta)$$

- Evaluating by an objective/reward function $O(\theta)$ ➔ how good θ is
- Best model parameter set

$$\theta^* = \arg\max_{\theta} O(\theta)$$

常用损失函数C(θ)来衡量模型训练的好坏，就是训练模型常提到的**loss**。损失函数C(θ)是一个衡量模型预测与实际观测值之间差异的函数。它的作用是量化参数集θ的“坏”的程度，即模型在给定参数下的预测误差。损失函数的选择取决于具体的任务和数据特点，常见的损失函数包括均方误差（MSE）、平均绝对误差（MAE）、交叉熵损失等。不同的任务需要不同的损失函数。

常见损失函数汇总（Frequent Loss Function）

以下是优化后的内容，格式清晰，公式美观，可直接复制到 Obsidian 中使用（支持 Markdown 和 LaTeX 公式渲染）：

---

1. 平方损失（Square Loss）

平方损失是最常用的损失函数之一，也称为**均方误差**（Mean Squared Error, MSE），适用于回归任务（预测连续值）。

定义

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- y_i：真实值
- $\hat{y}_i$：预测值
- N：样本数量

特点

优点：
✅ 计算简单，导数易求
✅ 线性回归问题中存在唯一全局最优解

缺点：
❌ 对异常值敏感（平方会放大异常值影响）
❌ 不适用于分类任务

---

2. Hinge Loss

主要用于**支持向量机（SVM）**的二分类任务。

定义

$$\text{Hinge Loss} = \max(0, 1 - y_i \cdot \hat{y}_i)$$

- y_i：真实标签（取值为 -1 或 1）
- $\hat{y}_i$：预测值（实数）

特点

优点：
✅ 适合 SVM，能找到最大间隔超平面
✅ 对正确分类样本损失为 0

缺点：
❌ 仅适用于二分类
❌ 多分类需扩展

---

3. Logistic Loss（对数损失）

适用于**二分类任务**，通过 Sigmoid 函数输出概率。

定义

$$\text{Logistic Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\sigma(\hat{y}_i)) + (1-y_i)\log(1-\sigma(\hat{y}_i))\right]$$

- y_i：真实标签（0 或 1）
- $\sigma(x) = \frac{1}{1+e^{-x}}$：Sigmoid 函数

特点

优点：
✅ 输出概率，解释性强
✅ 鼓励模型输出准确概率

缺点：
❌ 计算复杂度高（涉及对数运算）
❌ 多分类需扩展

---

4. Cross Entropy Loss（交叉熵损失）

Logistic Loss 的多分类版本，适用于**多分类任务**。

定义

$$\text{Cross Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

- y_{ic}：真实标签（0 或 1，表示样本 i 是否属于类别 c）

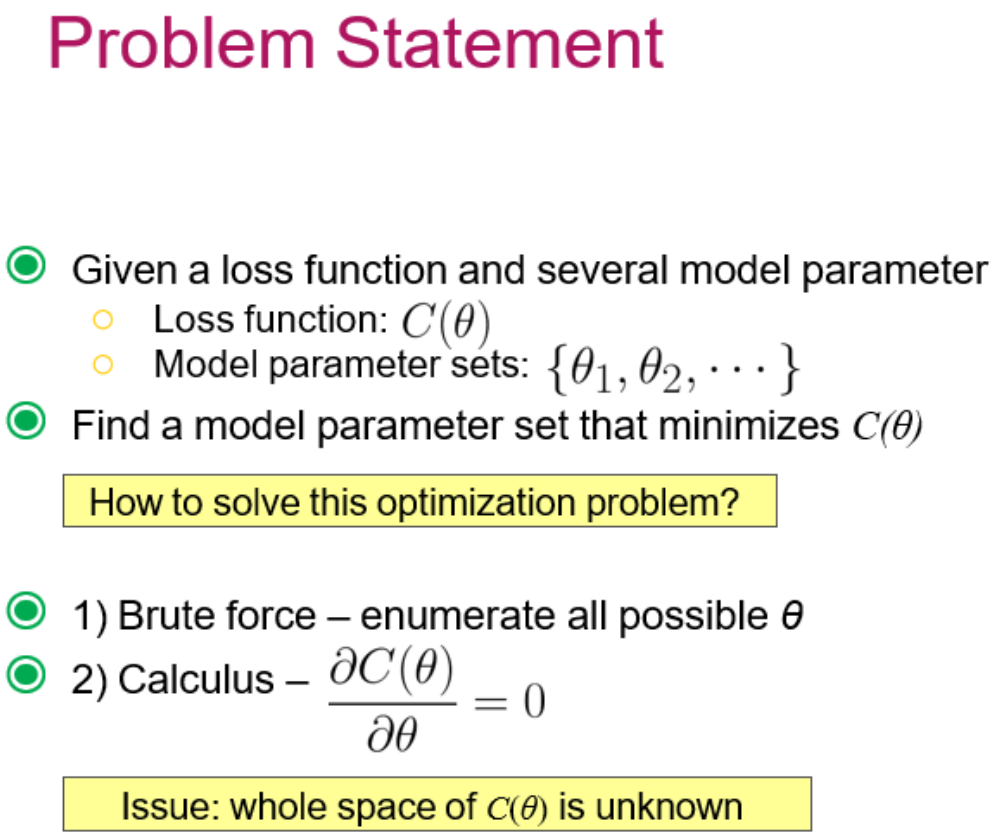
- $\hat{y}_{ic}$：预测概率（样本 i 属于类别 c 的概率）
- C：类别数量

特点

优点：
✅ 多分类任务的标准选择
✅ 概率输出易于解释

缺点：
❌ 需处理类别不平衡问题

Optimization

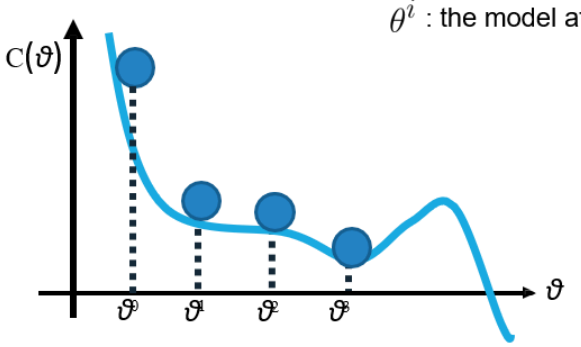


Gradient Descent

梯度下降法是一种迭代算法，用于求解无约束优化问题中的最小值。它通过计算目标函数（或损失函数）相对于其参数的梯度（或斜率），然后沿着梯度下降的方向更新参数，从而逐步逼近最小值。这个图演示得很好：

- Assume that θ has only one variable

θ^i : the model at the i-th iteration



Idea: drop a ball and find the position where the ball stops rolling (local minima)

假设目标函数为 J(θ)，其中 θ 是需要优化的参数向量。梯度下降算法的参数更新公式为：

$$\theta = \theta - \eta \cdot \nabla\theta \cdot J(\theta)$$

其中，η 是学习率，它决定了参数更新的步长。∇θJ(θ) 是目标函数 J(θ) 关于参数 θ 的梯度向量。

$$\eta$$

在梯度下降法中，梯度方向上迈出的步长大小，即学习率（learning rate），起着至关重要的作用。学习率决定了算法在每次迭代中沿梯度方向更新参数时的步幅大小。其影响主要体现在以下几个方面：

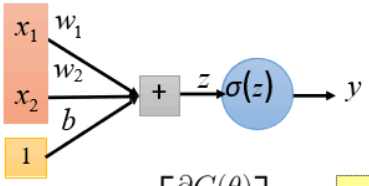
1.收敛速度：学习率的大小直接影响算法的收敛速度。如果学习率设置得过大，算法可能会跳过最优解，导致参数在最优解附近震荡而无法收敛；如果学习率设置得过小，虽然算法最终能够收敛到最优解，但收敛速度会非常慢，需要更多的迭代次数。
2.稳定性：学习率的选择还影响算法的稳定性。过大的学习率可能导致算法发散，即参数值越来越远离最优解；而过小的学习率虽然保证了算法的稳定性，但牺牲了收敛速度。

所以在MLP中是梯度下降优化的过程是什么样子的呢?

Gradient Descent for Optimization

Simple Case

$$y = f(x;\theta) = \sigma(Wx + b)$$
$$\theta = \{W, b\} = \{w_1, w_2, b\}$$



Algorithm
Initialization: start at θ^0
while($\theta^{(i+1)} \neq \theta^i$)
{
compute gradient at θ^i
update parameters
$\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta C(\theta^i)$
}

$$\nabla_\theta C(\theta) = \begin{bmatrix} \frac{\partial C(\theta)}{\partial w_1} \\ \frac{\partial C(\theta)}{\partial w_2} \\ \frac{\partial C(\theta)}{\partial b} \end{bmatrix}$$

$$\begin{bmatrix} w_1^{i+1} \\ w_2^{i+1} \\ b^{i+1} \end{bmatrix} \leftarrow \begin{bmatrix} w_1^{i} \\ w_2^{i} \\ b^{i} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C(\theta)}{\partial w_1} \\ \frac{\partial C(\theta)}{\partial w_2} \\ \frac{\partial C(\theta)}{\partial b} \end{bmatrix}$$

1.计算梯度：

- 在当前参数值 θi 处，计算损失函数 J(θ) 关于 θ 的梯度 ∇θJ(θi)。
- 梯度表示了损失函数在当前点处最陡峭的上升方向。因此，为了最小化损失函数，我们需要沿着梯度的相反方向更新参数。

2.更新参数：

- 使用梯度下降更新规则来更新参数：

$$\theta_{i+1} = \theta_i - \eta \nabla \theta J(\theta i)$$

- 这里，η 是学习率，它决定了参数更新的步长。学习率的选择非常重要，太大可能导致算法发散，太小则会使收敛速度过慢。

3.检查收敛性(减枝还在追我)：

- 检查新的参数值 θi+1 是否与上一个参数值 θi 足够接近（通常是在某个预设的容差范围内）。
- 如果足够接近，则认为算法已经收敛到最优解（或至少是一个足够好的近似解），并停止迭代。
- 如果不接近，则继续回到步骤1，使用新的参数值 θi+1 进行下一次迭代。

一个示例：

- **训练数据** (2个样本)：

样本	特征x_1	特征x_2	特征x_3	真实值y
$x^{(1)}$	1	2	3	6
$x^{(2)}$	4	5	6	15

- **模型**：线性回归

$$h\theta(x) = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

- **损失函数** (MSE)：

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2$$

- **初始参数**：

$$\theta^{(0)} = [0, 0, 0]$$

- **学习率(虽然但是，过大的学习率)**：

$$\eta = 0.1$$

- **至于梯度 = 损失函数针对θ求导**

$$\frac{\partial L}{\partial \theta} = \frac{\partial}{\partial \theta}\left(\frac{1}{n}\sum_{i=1}^{n}(\theta x^{(i)} - y^{(i)})^2\right) = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial \theta}(\theta x^{(i)} - y^{(i)})^2$$

- **化简一下**：

$$\frac{\partial L}{\partial \theta} = \frac{2}{n}\sum_{i=1}^{n}(\theta x^{(i)} - y^{(i)})x^{(i)}$$

$$h\theta(x_1) = 0\times1 + 0\times2 + 0\times3 = 0$$
$$h\theta(x_2) = 0\times4 + 0\times5 + 0\times6 = 0$$

链式求导法则求梯度：

$$\frac{\partial L}{\partial \theta_1} = \frac{1}{2}[(0-6)\cdot1 + (0-15)\cdot4] = -33$$
$$\frac{\partial L}{\partial \theta_2} = \frac{1}{2}[(0-6)\cdot2 + (0-15)\cdot5] = -43.5$$
$$\frac{\partial L}{\partial \theta_3} = \frac{1}{2}[(0-6)\cdot3 + (0-15)\cdot6] = -63$$

$$\theta 1(1) = 0 - 0.1 \times (-33) = 3.3$$

$$\theta 2(1) = 0 - 0.1 \times (-43.5) = 4.35$$

$$\theta 3(1) = 0 - 0.1 \times (-63) = 6.3$$

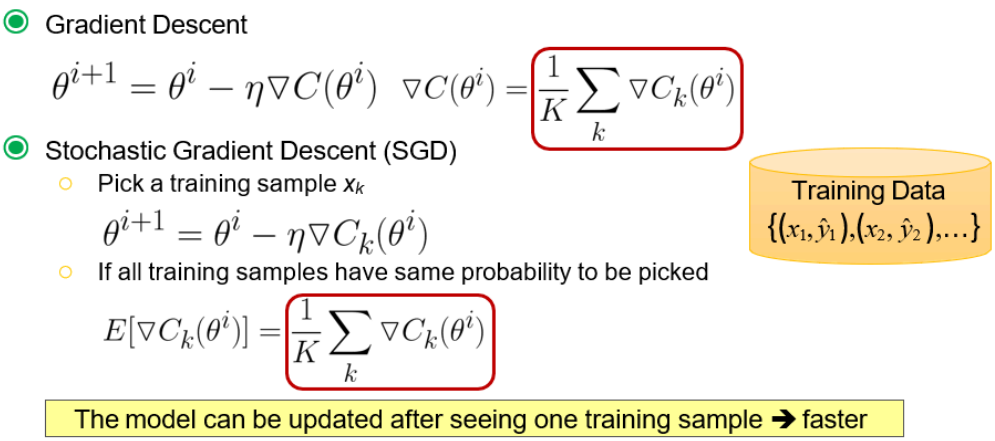
因此结果如下：

迭代次数	θ1	θ2	θ3	MSE
t=0	0	0	0	130.5
t=1	3.3	4.35	6.3	2956.28625（？？？哦哦哦这个怪学习率）

总之这就是个例子了。

Stochastic Gradient Descent (SGD)

SGD是梯度下降算法的一个变种，它的核心思想是在每次迭代中，不是使用整个数据集来计算梯度并更新模型参数，而是**随机选择一个样本**来计算梯度并更新参数。这样做的好处是大大提高了算法的效率，尤其是在处理大规模数据集时，能够快速更新参数。



算法步骤

- 1.初始化参数：设置模型的初始参数值。
- 2.计算梯度：对于随机选择的样本或mini-batch，计算损失函数关于模型参数的梯度。
- 3.更新参数：沿着梯度的反方向（即损失函数下降的方向）更新模型参数。
- 4.重复迭代：重复上述步骤，直到满足停止条件（如达到预设的迭代次数、参数变化量小于某个阈值、损失函数值收敛到某个水平等）。

特点与优势

- 1.高效性：由于每次迭代只使用部分数据，SGD可以更快地进行参数更新，这在处理大规模数据集时尤其有用。
- 2.在线学习：SGD能够在线更新模型，即可以实时地处理新数据并更新模型参数。
- 3.减少过拟合：随机选择样本可以减少模型对训练数据的过拟合，因为它引入了一定的噪声，迫使模型学习更加泛化的特征。

Epoch定义

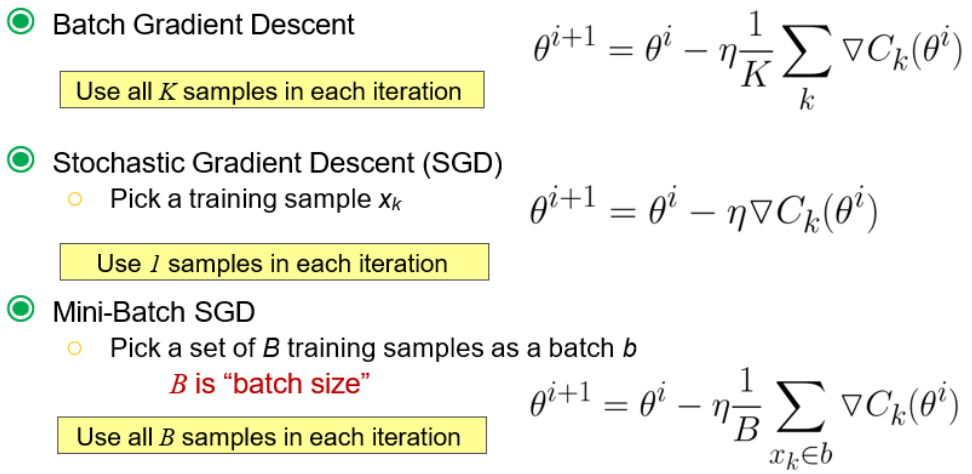
Epoch指的是在训练过程中，整个数据集被完整地传递给模型一次的过程。也可以理解为，模型对所有训练样本进行一次完整的学习。

- 1.完整性：一个Epoch意味着模型已经看到了数据集中的所有样本至少一次。这是通过前向传播和（可能的）反向传播过程实现的，其中模型根据当前参数预测输出，并根据损失函数计算误差，然后通过反向传播算法调整参数以减小误差。
- 2.迭代与批次：在实际训练中，由于数据集可能非常大，无法在一次迭代中处理整个数据集。因此，数据集通常被分成多个小批次（Batches），每个批次包含一定数量的样本。模型在每个Epoch中逐个处理这些批次，直到完成对整个数据集的遍历。
- 3.参数更新：在每个批次（Batches）处理后，模型通常会根据该批次的损失函数梯度来更新其参数。这意味着在每个Epoch结束时，模型的参数已经根据整个数据集中的所有样本进行了多次更新。

Mini-Batch SGD

一种比较稳定的解决方案，介于BGD和SGD之间，使用一个batch size来训练（几十或几百样本）。

对比如下：



批量梯度下降（Batch Gradient Descent）

- 1.在每次迭代中，使用**整个数据集**来计算梯度并更新参数。
- 2.优点：对于凸误差曲面，可以保证收敛到全局最小值；对于非凸误差曲面，可以收敛到局部最小值。
- 3.缺点：计算量大，更新速度慢，且不适用于大数据集。

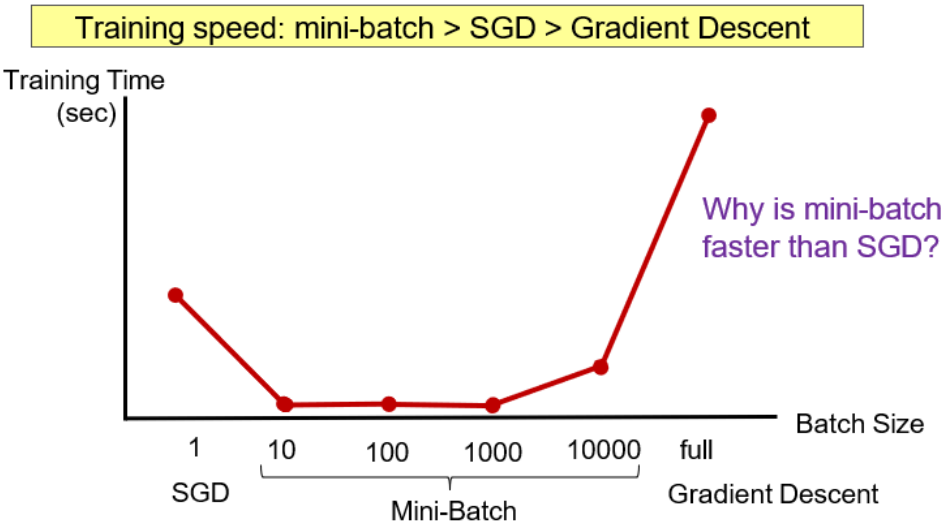
随机梯度下降（Stochastic Gradient Descent, SGD）

- 1.在每次迭代中，仅使用**一个**训练样本来计算梯度并更新参数。
- 2.优点：计算速度快，可以在线更新模型，适用于大数据集。
- 3.缺点：目标函数波动严重，可能导致收敛不稳定，且可能陷入局部最小值。

小批量梯度下降（Mini-batch Gradient Descent）

- 1.在每次迭代中，使用一个**小批量**（通常是几十到几百个）的训练样本来计算梯度并更新参数。
- 2.优点：结合了批量梯度下降和随机梯度下降的优点，既保证了计算的稳定性，又提高了更新速度。
- 3.缺点：需要选择一个合适的小批量大小，以及学习率。
- **在每个epoch之前打乱训练样本！以避免模型记住样本的输入顺序**

以下是**训练速度**比对：

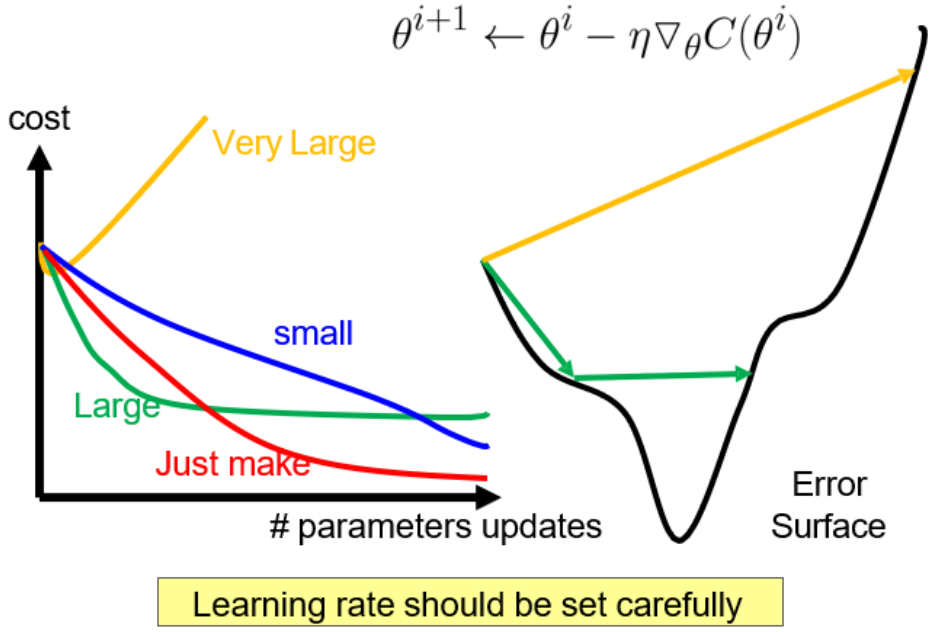


Practical Tips

第一个就是**学习率**的选择！

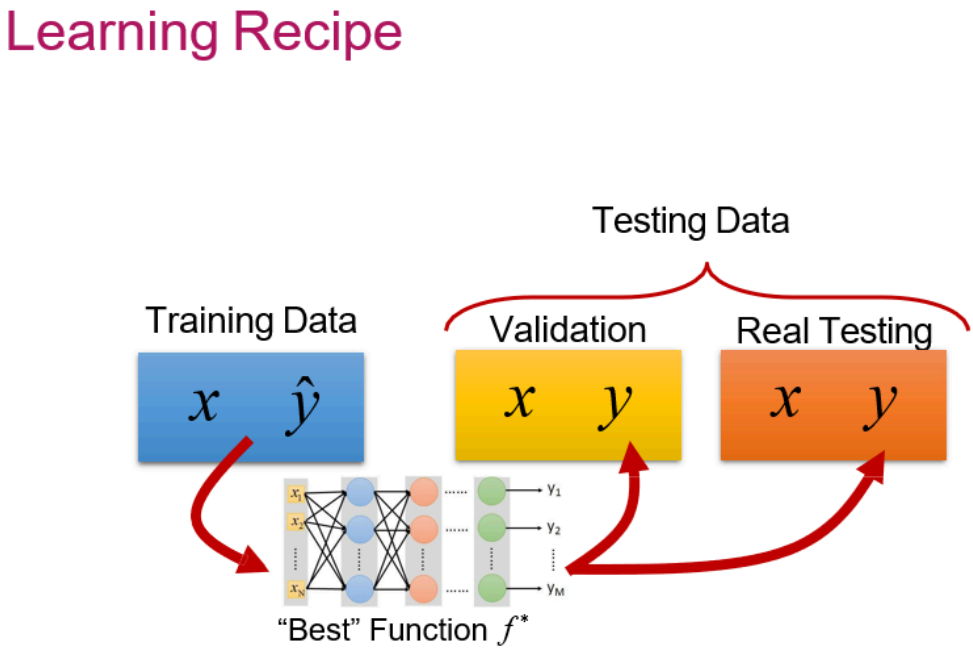
- 根据批次大小和数据分布，调整学习率以优化训练过程。

- 可以使用学习率衰减策略，如指数衰减、余弦衰减等，来逐步降低学习率。

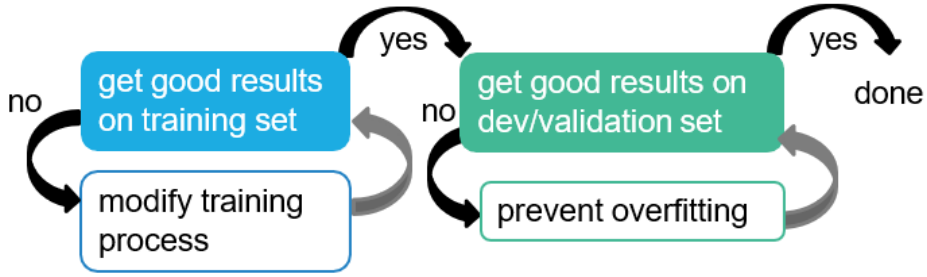


再就是**学习策略**?

- 数据集划分，使用training data进行训练，和testing data 比重一般1:9.
- 训练过程中监控验证集的性能，当验证集性能不再提升时停止训练，以防止过拟合
- Validation data 能迅速知道模型效果
- Real Testing do not know the performance until submission.



如果训练效果不理想怎么办?



可能原因一：不存在好的函数（即假设函数集不佳）

- 模型在训练集和验证集上的表现都很差。
- 即使增加训练时间或数据，模型性能也没有显著提升。
- 调整策略：重构模型架构：尝试使用不同的模型架构，如从线性模型切换到深度神经网络，或者从卷积神经网络（CNN）切换到循环神经网络（RNN）等，以适应数据的特性。

可能原因二：无法找到好的函数（即陷入局部最优解）

- 模型在训练集上表现良好，但在验证集上表现不佳。
- 训练过程中损失函数下降缓慢或停滞。

Summary PPT上的
Q1: What is the model?

- **Model Architecture.**
  Q2: What does a "good" function mean?
- **Loss Function Design.**
  Q3: How do we pick the "best" fuction?
- **Optimization.**
