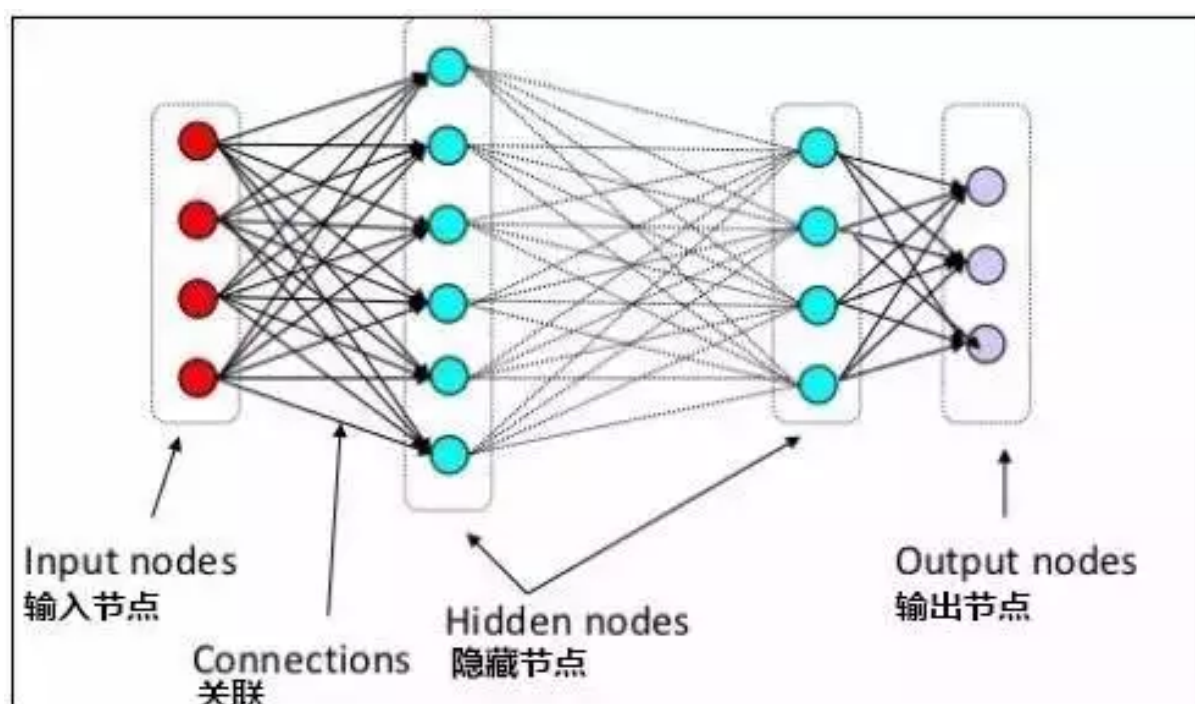


Pytorch逻辑回归

人工神经网络是一种模仿生物神经网络(动物的中枢神经系统，特别是大脑)的结构和功能的数学模型或计算模型。

神经网络是人工智能深度学习的基础，也是目前主流网络模型应用最多的基础架构。



通常学习神经网络最好的入门方法也是先学习逻辑回归，而后进一步认识神经元和整体的网络训练。

线性回归

线性回归的目标就是找到一个线性函数来拟合数据，使得预测值与真实值之间的误差尽可能小，而参数估计就是确定这个线性函数中参数的过程。

简单线性回归的参数估计

简单线性回归模型的表达式为： $y = \beta_0 + \beta_1 x + \epsilon$ ，其中 y 是因变量， x 是自变量， β_0 是截距， β_1 是斜率， ϵ 是误差项（服从均值为 0 的正态分布）。

自变量 是我们认为对因变量 有影响的变量，是模型中用来预测或解释因变量变化的因素。

因变量 是受到一个或多个自变量影响的变量，也被称为响应变量或被解释变量。

截距 是线性回归模型中的一个常数项，截距的值反映了除自变量之外的其他所有因素对因变量的综合影响（当 $x = 0$ 时）。在机器学习领域也被称为 **偏差 (bias)**。

斜率 表示直线的倾斜程度，它衡量了自变量每变动一个单位时，因变量的平均变动量。在机器学习和优化算法中，为了更简洁通用地表示这些参数，常常用 θ 来统一表示模型的参数向量。

参数估计常用的方法是最小二乘法（Least Squares Method），其基本思想是找到一组参数 β_0 和 β_1 ，使得观测值 y_i 与预测值 \hat{y}_i 之间的误差平方和最小。预测值 $\hat{y}_i = \beta_0 + \beta_1 x_i$ ，误差平方和 $S(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2$ 。

为了找到使

$S(\beta_0, \beta_1)$ 最小的 β_0 和 β_1 ，分别对 β_0 和 β_1 求偏导数，并令偏导数等于 0：

- $\frac{\partial S}{\partial \beta_0} = -2 \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i) = 0$
- $\frac{\partial S}{\partial \beta_1} = -2 \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i) x_i = 0$

通过求解上述方程组，可以得到参数的估计值：

- $\hat{\beta}_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$
- $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$

其中 \bar{x} 和 \bar{y} 分别是 x 和 y 的样本均值。

多元线性回归的参数估计

多元线性回归模型的表达式为： $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon$ ，其中 x_1, x_2, \cdots, x_p 是 p 个自变量。

同样使用最小二乘法，误差平方和为

$$S(\beta_0, \beta_1, \cdots, \beta_p) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \cdots - \beta_p x_{ip})^2$$

其中 x_{ip} 表示第 i 个观测值的第 p 个自变量的值。

为了找到使

S 最小的参数 $\beta_0, \beta_1, \cdots, \beta_p$ ，对每个参数求偏导数并令其等于 0，组成一个包含 $p + 1$ 个方程的方程组。通过矩阵运算，可以得到参数的估计值为： $\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ ，其中 $\hat{\beta}$ 是包含 $\hat{\beta}_0, \hat{\beta}_1, \cdots, \hat{\beta}_p$ 的参数向量， \mathbf{X} 是包含自变量观测值的设计矩阵（第一列通常为 1，对应截距项）， \mathbf{y} 是因变量的观测值向量。

逻辑回归

逻辑回归就是将线性回归模型映射为概率的模型。

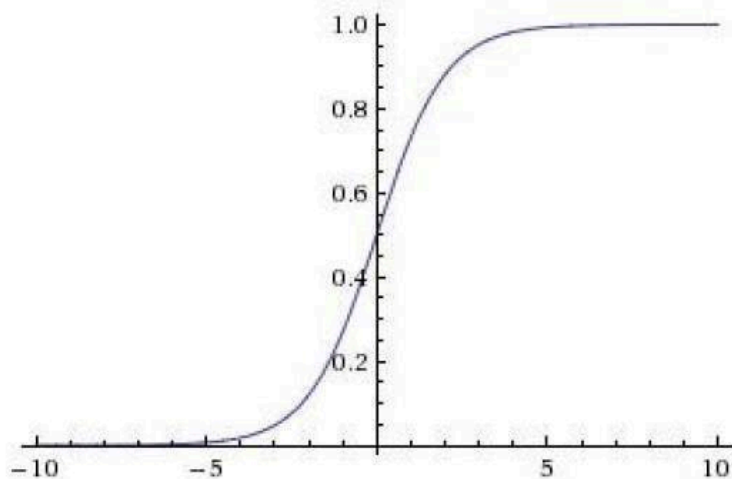
把 \hat{y} 实数空间的输出 $[-\infty, +\infty]$ 映射到取值为 $[0, 1]$ 的区间，从而把模型的输出值转换为概率值。

而这其中的转换，我们使用的就是sigmoid函数

$$\hat{p} = \text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

```
def sigmoid(z):  
    return 1 / (1 + np.exp(-z))
```

$\text{sigmoid}(z)$ 的取值范围



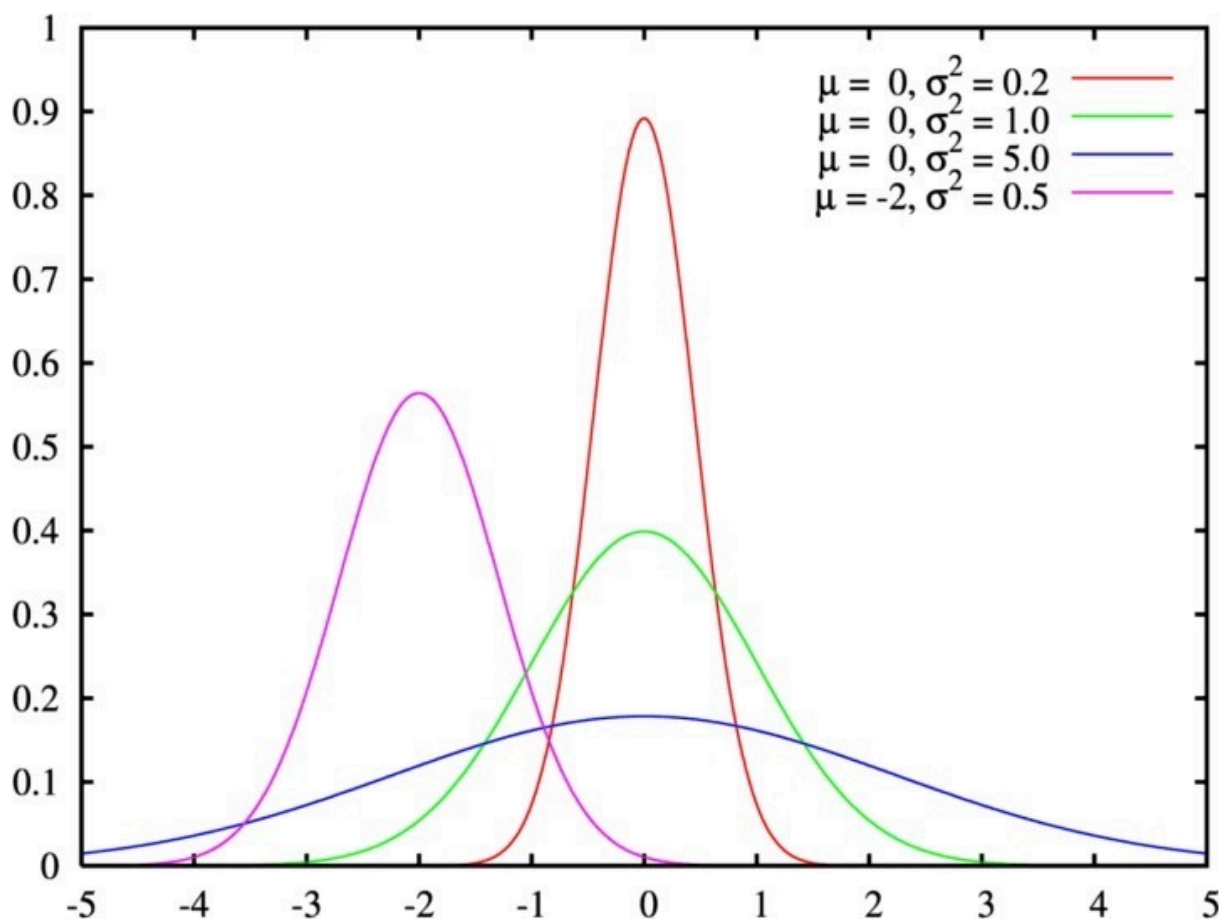
Sigmoid

最大似然估计和损失函数

回顾一下正态分布的概率密度函数

$$X \sim N(\mu, \sigma^2), \quad f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

其中 μ 是均值, σ 是标准差。



正态分布的概率在均值处概率最高, 以均值为中心两边是对称的, σ 是标准差, 标准差控制着概率分布偏离均值的程度, 标准差越大概率分布越扁平, 越小的话, 概率分布越集中。

概率和似然的区别

概率 (Probably), 是在已知一些概率分布参数的情况下, 预测观测值的结果;

似然 (Likelihood), 则是用于在已知某些观测值所得到的结果时, 对观测结果所属的概率分布参数进行估计。

当我们讨论具体的样本数据时, x 常被称为**观测值**。在实际研究中, 我们会收集一系列样本点的数据, 对于每个样本点, 都有对应的自变量 x 的取值和因变量 y 的取值。例如, 我们收集了 n 个房屋的相关数据, 对于第 i 个房屋, 它的面积 x_i 就是一个观测值 (同时 y_i 表示第 i 个房屋的价格, 也是一个观测值)。这里的观测

值是实际收集到的数据点，是具体的数值，用于对模型进行参数估计和评估等操作。

$$P(x|\theta)$$

x 表示某一个具体的数据； θ 表示模型的参数。

- 如果 θ 是已知确定的， x 是变量，这个函数叫做**概率函数(probability function)**，它描述对于不同的样本点 x ，其出现概率是多少。
- 如果 x 是已知确定的， θ 是变量，这个函数叫做**似然函数(likelihood function)**，它描述对于不同的模型参数，出现 x 这个样本点的概率是多少。

最大似然估计

最大似然估计 (Maximum Likelihood Estimation) 就是一种可以生成拟合数据任何分布的参数的最可能估计的技术。

简单解释就是：最大似然估计的目的就是找到一个最符合当前观测数据的概率分布。

似然函数

假设我们有一组观测到的数据， $X = [x_1, x_2, \dots, x_N]$ 共有 N 个。我们假设这组数据属于正态分布，它的概率密度函数为 $f(x; \mu, \sigma)$ ，我们把 X 的值带入到 $f(x; \mu, \sigma)$ 里，得到的是每个数据点 x 在假设的概率分布中出现的可能性。

$$P(x_1 | \mu, \sigma), P(x_2 | \mu, \sigma), \dots, P(x_N | \mu, \sigma)$$

那么这组数据 X 在假设的概率分布中的出现的可能性就是它们所有概率的乘积：

$$L(\mu, \sigma | X) = P(X | \mu, \sigma) = \prod_{i=1}^N P(x_i | \mu, \sigma)$$

$L(\mu, \sigma | X)$ 就是似然函数，利用已知观测值 X ，来估计参数 μ, σ 的可能性。所以说，似然函数也是一种条件概率函数。

对数似然函数

我们对似然函数取 \log 对数，就是对数似然函数

$$\mathcal{L}(\mu, \sigma | X) = \sum_{i=1}^N \log P(x_i | \mu, \sigma)$$

对数似然函数是在神经网络优化中 **最常用的损失函数**



为什么要取对数？

原始的似然函数是很多条件概率的乘积, 在计算极大值的时候需要求似然函数的导数。而乘积的导数计算很麻烦, 所以取对数可以把乘法变成加法。

损失函数

对于上面的逻辑回归, 我们引入假设函数, 预测

Y 等于1的概率: $h_{\theta}(X) = \frac{1}{1+e^{-z}} = P(Y = 1 | X; \theta)$

Y 等于0的概率: $P(Y = 0 | X; \theta) = 1 - h_{\theta}(X)$

假设函数 (hypothesis) 本质上是一个模型, 用于描述自变量和因变量之间的映射关系。

它根据给定的输入数据 (自变量的值), 通过特定的数学运算和参数设置, 预测或估计相应的输出值 (因变量的值)。

例如, 在简单线性回归中, 假设函数的形式为

$h_{\theta}(x) = \theta_0 + \theta_1 x$, 其中 x 是自变量, $h_{\theta}(x)$ 是基于参数 θ_0 和 θ_1 对因变量的预测值。这里的 $h_{\theta}(x)$ 就是一个假设函数, 它假设因变量与自变量之间存在线性关系。

通过最大似然估计, 定义似然函数, 将 $h_{\theta}(X)$ 代入到伯努利分布的概率质量函数中

$$\begin{aligned}
 L(\theta | x) &= P(Y | X; \theta) \\
 &= \prod_i^m P(y_i | x_i; \theta) \\
 &= \prod_i^m h_{\theta}(x_i)^{y_i} (1 - h_{\theta}(x_i))^{(1-y_i)}
 \end{aligned}$$

伯努利分布 (Bernoulli distribution) 是一个离散型概率分布, 用于描述只有两种可能结果的随机试验, 例如抛硬币 (正面或反面)、产品是否合格 (合格或不合格) 等。

设随机变量

X 服从伯努利分布, 若试验成功的概率为 $p(0 \leq p \leq 1)$, 试验失败的概率为 $1 - p$, 则伯努利分布的概率质量函数为:

$$P(X = k) = p^k (1 - p)^{1-k}, \text{ 其中 } k = 0, 1$$

当

$k = 1$ 时, 表示试验成功, 此时 $P(X = 1) = p$; 当 $k = 0$ 时, 表示试验失败, 此时 $P(X = 0) = 1 - p$ 。

以方便计算, 上式取 \log , 转为对数似然 (乘法转加法)

$$L(\theta | x) = \log(P(Y | h_{\theta}(X))) = \sum_{i=1}^m y_i \log(h_{\theta}(x_i)) + (1 - y_i) \log(1 - h_{\theta}(x_i))$$

上面的公式前缀加个负号, 就可把求最大转换为求最小。设 $h_{\theta}(X) = \hat{Y}$, 得出损失函数 $J(\theta)$

只要最小化这个函数的结果值, 就可得出我们想要的 θ

$$J(\theta) = - \sum_i^m Y \log(\hat{Y}) - (1 - Y) \log(1 - \hat{Y})$$


```
def loss_function(y, y_hat):  
    e = 1e-8 # 防止y_hat计算值为0, 添加的极小值epsilon  
    return - y * np.log(y_hat + e) - (1 - y) * np.log(1 - y_hat + e)
```

梯度下降法

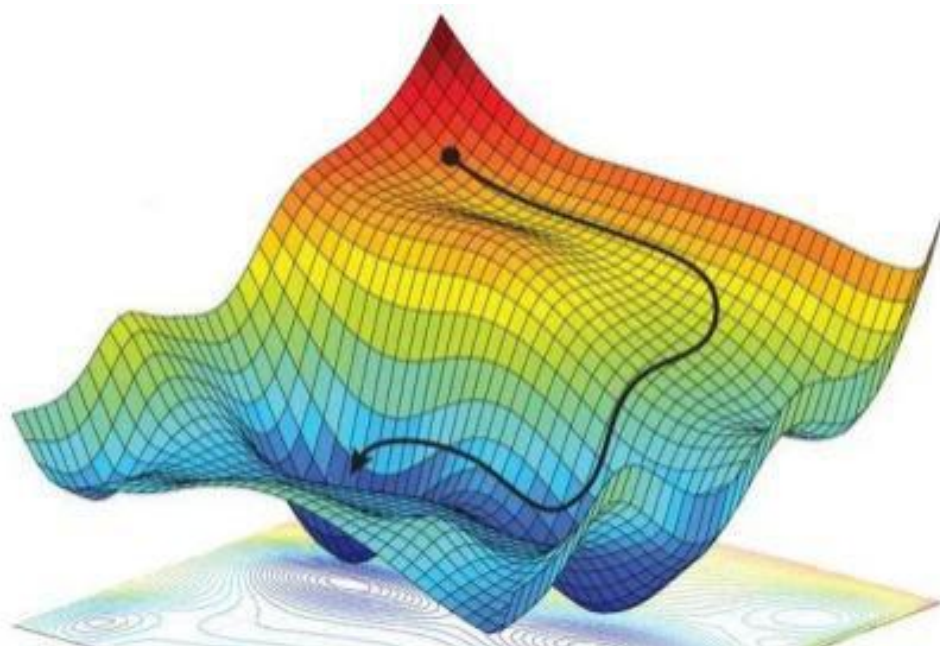
梯度下降法并不是一个机器学习的算法，它既不能解决回归问题，也不能解决分类问题。

那么它是什么呢？梯度下降是一种基于搜索的最优化的方法，它的目标是用于优化一个目标函数。

在机器学习中，**梯度下降法的作用就是：最小化一个损失函数。**

下山问题

梯度下降的思想其实就是一个下山的过程，假设我们爬到了山顶，然后感觉很累想尽快下山休息。但是这个时候起雾了，山上都是浓雾，能见度很低。那么我们下山的路径是无法确定的，这时候我们要怎么下山呢？聪明的你肯定能想到利用周围能看见的环境去找下山的路径，这个时候梯度下降的思想就能体现了，我们首先要找到周围最陡峭的地方，然后朝着地势往下的方向走，走完一段距离发现最陡峭的方向变了，那么我们调整一下方向接着沿地势往下的方位走。这样不断的进行下去，理论上最后一定会到达山脚，如下图所示



梯度下降图示

假设这座山有的坡特别平缓，那么几个方向的陡峭程度很接近，我们无法用肉眼测量出最陡峭的，假如恰好我们带了一个测量仪，然后每走一段距离我们就会测量一次，这时候就会出现一个问题，如果我们每走几步就测量一次，那么可能走到明天我们都下不去山，但是我们如果减少测量次数，那么可能就会偏离下山的最快的路径。这里我们就会需要一个合适的测量频率，保证我们下山路径足够快，然后又不需要太多时间去测量。

这个例子对应到模型的训练过程中的梯度下降算法又是怎么回事呢？学完前面的内容都知道，我们将模型带入观测值 x 会计算出一个输出 \hat{y} ，输出 \hat{y} 与真实的数据标签 y 输出对比会产生一个误差，然后根据误差会设定一个损失函数，那么损失函数与之前的参数能形成一个复杂的函数，这个函数就是在刚才举例中的那一座山，那么山脚对应损失函数的最小值。最快的下山方式就是在当下位置找最陡峭的地方然后沿着这个方向往下走，对应到上述复杂函数中，就是根据当前的模型参数找到梯度的方向，然后朝梯度的反向走，这样函数值就能下降最快，因为梯度方向是函数变化最快的方向。

因此，我们不断的使用这种方法，反复的求取梯度，函数就能达到相对的最小值。为什么是相对而不是绝对呢？我们下山过程中可能会遇到这种情况，根据地势往下走，不小心进入了一个很深的洼地，这时四周都是地势向上，人是很聪明的，我们可以爬出来继续找个其它地势向下的方向，但是计算机可没有那么聪明，它可能会深陷其

中无论怎么求梯度都爬不出来了。这就是函数可能只能够达到局部的最小值，而不是全局的最小值。

梯度与学习率

- 梯度

梯度下降的梯度是什么？函数在某一点的梯度是这样一个向量，它的方向与取得最大方向导数的方向一致，而它的模为方向导数的最大值。在存在多个变量的函数中，梯度是一个向量，向量有方向，梯度的方向就指出了函数在给定点的上升最快的方向。这就意味着我们需要到达山底，那么在我们下山测量中，梯度就告诉我们下山的方向。梯度的方向是函数在给定点上升最快的方向，那么反方向就是函数在给定点下降最快的方向，所以我们只要沿着梯度的反方向一直走，就能走到局部的最低点！对于梯度的理解可能不太容易，我们需要搜索一些梯度与导数的资料，然后再去看一部分拟牛顿法的资料来进行深刻的体会理解。

由于神经网络模型中有众多的参数，也称为**权重参数 (weight parameter)** 所以我们常常需要处理的是多元复合函数，要想知道某一个权重参数对损失函数的影响，那么就要求它的偏导数。因此对于权重参数，我们是可以确定向量的方向的，就是求它的导数值就可以了（所以说，要想理解梯度下降的数学原理必须要明白导数微分的概念）。

权重参数 是模型中用于衡量输入特征重要程度的数值参数。以线性模型为例，如简单线性回归模型 $y = \beta_0 + \beta_1 x$ ，其中 β_1 就是权重参数（这里 x 是自变量），它表示自变量 x 对因变量 y 的影响程度和方向。在更复杂的神经网络模型中，权重参数存在于神经元之间的连接上，每个连接都有一个对应的权重值。

下面我们给出梯度下降的描述方程：

$$x_{n+1} = x_n - \eta \frac{\partial f(x, y)}{\partial x}$$

在这解释一下上式，例如原函数为 $z = f(x, y) = x^2 + y^2$ ，那么上述公式是一次迭代更新 x 求解的过程。 x_{n+1} 是更新后的 x ， x_n 是当前的 x ， $\frac{\partial f(x, y)}{\partial x}$ 代表在 $f(x, y)$ 函数中对 x 求偏导， η 代表学习率。

• 学习率

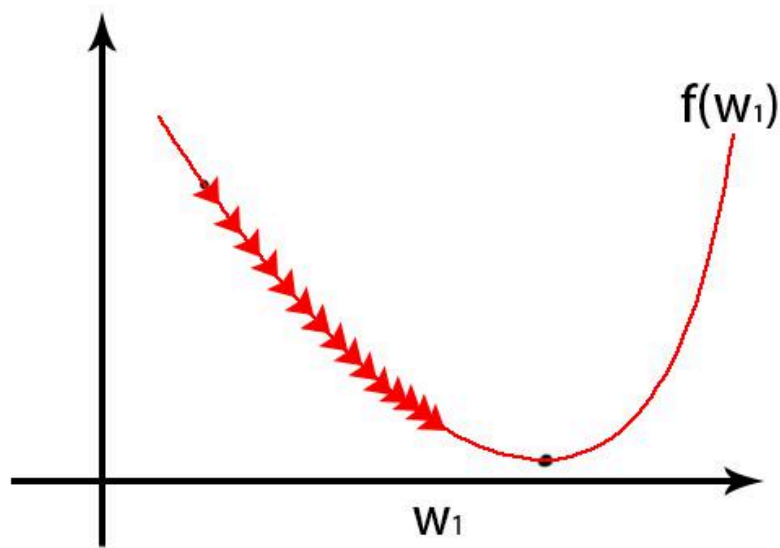
学习率看起来很陌生，但结合着下山问题来解释就很好理解了。比如说，我们要找一个合适的测量频率，保证我们下山又快，测量次数又少。那么这个学习率就是影响我们测量频率的因素。可以将其理解为梯度下降过程中的步长。

在我们训练模型的时候，学习率是里面很重要的一个超参数

(Hyperparameters)，它决定着我们的损失函数能否收敛到最小值，还需要多长时间才能收敛到最小值。一个合适的学习率能够让我们的损失函数在合适的时间内收敛到局部最小值。

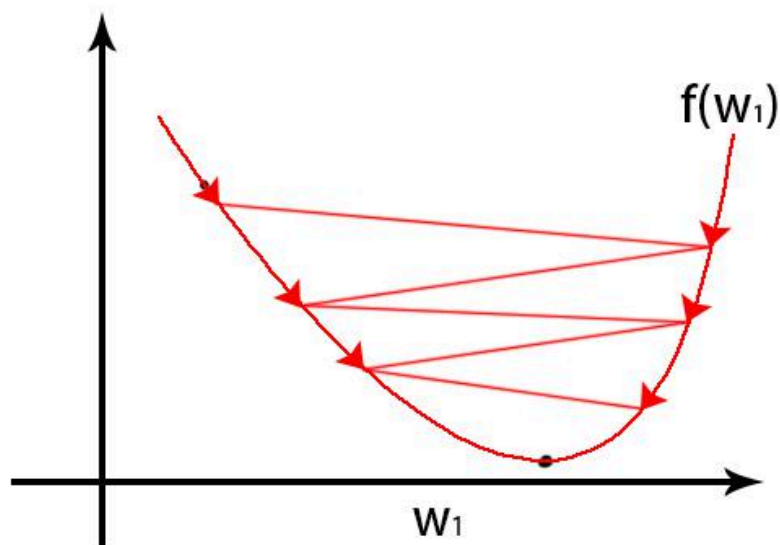
超参数 (Hyperparameters) 是指那些在训练过程开始之前就需要手动设置的参数，它们不能通过模型从数据中学习得到，而是需要通过人为的经验、实验或者特定的搜索方法来进行选择和调整。

在梯度下降算法里，例如一次权重更新中， $w_{n+1} = w_n - \eta \frac{\partial f(w)}{\partial w}$ 为我们梯度下降的方程，假设它是一元函数，如果学习率设置的过小时，函数收敛过程如下图 所示：



学习率过小时函数收敛过程

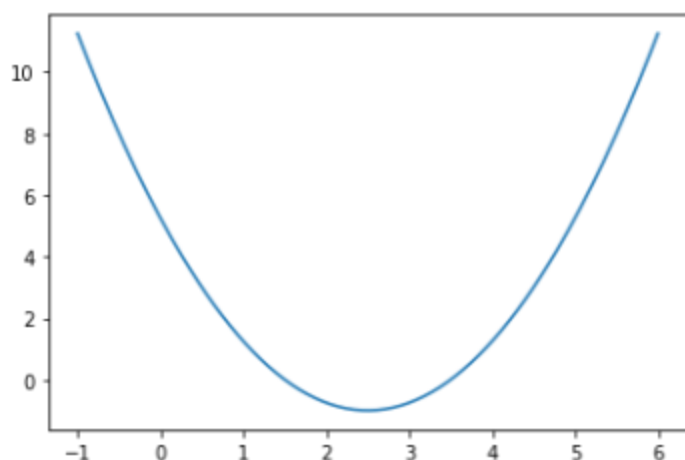
如果学习率过大的时候，函数收敛过程如下图所示：



学习率过大时函数收敛过程

可以看出来，当学习率设定太小时，收敛过程将变得十分缓慢并且可能不收敛。反而学习率设置的过大时，梯度可能会在最小值附近来回震荡，甚至可能无法收敛。学习率有一些优化方法，它可以是固定的也可以是有衰减的，对于优化方法，我们后面再详细讲解。

梯度下降法的模拟与可视化



上图描述的是一个**损失函数**在**函数不同参数**取值的情况下(x轴)，**对应变化值**(y轴)。

对于损失函数，应该有一个**最小值**。对于最小化损失函数这样一个目标，实际上就是在上图所示的坐标系中，寻找合适的参数，使得损失函数的取值最小。

在这个例子是在2维平面空间上的演示，所以参数只有1个(x轴的取值)。意味着每个参数，都对应着一个损失函数值。实际上模型的参数往往不止一个，所以这个图像描述的只是损失函数优化过程的一种简单表达。

我们用代码来绘制一张这样的图

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# 创建等差数列x,代表模型的参数取值,共150个
plot_x = np.linspace(-1., 6., 150)
print(plot_x)
```

```
[-1.      -0.95302013 -0.90604027 -0.8590604  -0.81208054 -0.76510067
 -0.71812081 -0.67114094 -0.62416107 -0.57718121 -0.53020134 -0.48322148
 -0.43624161 -0.38926174 -0.34228188 -0.29530201 -0.24832215
```

-0.20134228

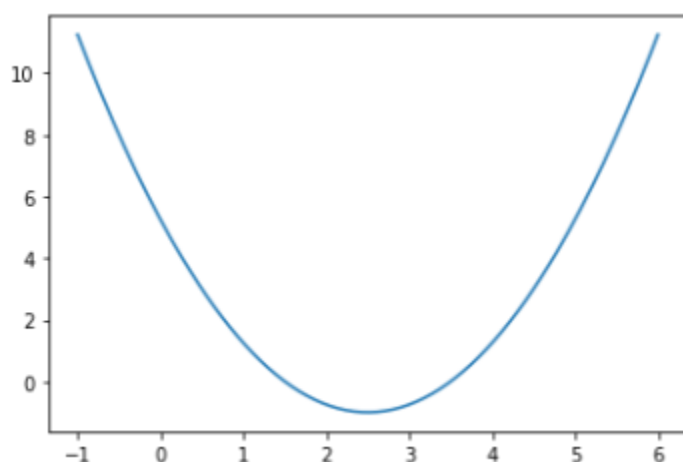
..... 略

```
5.20134228 5.24832215 5.29530201 5.34228188 5.38926174 5.43624161
5.48322148 5.53020134 5.57718121 5.62416107 5.67114094 5.71812081
5.76510067 5.81208054 5.8590604 5.90604027 5.95302013 6.    ]
```

```
# 通过二次方程来模拟一个损失函数的计算,plot_y值就是图形上弧线所对应的点
plot_y = (plot_x-2.5)**2 - 1
```

绘制图形

```
plt.plot(plot_x, plot_y)
plt.show()
```



下面，我们来尝试一下实现我们的梯度下降法。



在机器学习、统计学领域中，通常把线性函数或模型中的斜率和截距表示为 θ 和 *bias*

在一般的线性模型 $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon$ 中， β_0 是截距， $\beta_1, \beta_2, \cdots, \beta_n$ 是各个自变量 x_1, x_2, \cdots, x_n 对应的斜率（系数）。为了更简洁通用地表示这些参数，在机器学习和优化算法中，常常用 θ 来统一表示模型的参数向量。例如，令 $\theta = [\theta_0, \theta_1, \cdots, \theta_n]^T$ ，这样线性模型就可以写成 $y = \theta^T x + \epsilon$ ，其中 $x = [1, x_1, \cdots, x_n]^T$ （添加 $x_0 = 1$ 是为了将截距项也纳入到向量运算中）。这里的 θ_0 就对应原来的截距 β_0 ， $\theta_1, \cdots, \theta_n$ 对应原来的斜率 β_1, \cdots, β_n 。这种表示方式使得在推导公式、编写算法代码等过程中更加简洁和统一，便于处理多元线性模型以及进行各种数学运算和优化操作。

简单解释：把斜率和截距表示为 θ 和 *bias* 是为了在数学表达、概念理解和实际应用中提供便利，并符合该领域的习惯和约定。

首先定义一个函数，来计算损失函数对应的导数。目标就是计算参数 θ 的导数

```
def derivative(theta):  
    return 2*(theta-2.5)
```

还需要定义一个函数来计算theta值对应的损失函数

```
def loss(theta):  
    return (theta-2.5)**2 - 1
```

计算梯度值


```

# 以0作为theta的初始点
theta = 0.
eta = 0.1
epsilon = 1e-8
while True:
    # 计算当前theta对应点的梯度(导数)
    gradient = derivative(theta)

    # 更新theta前,先积累下上一次的theta值
    last_theta = theta

    # 更新theta,向导数的负方向移动一步,步长使用eta(学习率)来控制
    theta = theta - eta * gradient

    # 理论上theta最小值应当为0是最佳。但实际情况下, theta很难达到刚好等于0的情况
    # 所以我们可以设置一个最小值epsilon来表示我们的需要theta达到的最小值目标
    # 判断theta每次更新后,和上一个theta的差值是否已满足小于epsilon(最小值)的条件
    # 满足的话就终止运算
    if(abs(loss(theta) - loss(last_theta)) < epsilon):
        break

print(theta)
print(loss(theta))

```

2.499891109642585

-0.99999998814289

通过结果可以看出,当theta取2.5时候,损失函数的最小值正好对应着截距-1

学习率对梯度的影响

为了能够计算不同学习率下,theta的每一步变更值。我们对代码进行一些补充。

```

theta = 0.0
eta = 0.1
epsilon = 1e-8

# 添加一个记录每一步theta变更的list
theta_history = [theta]
while True:
    gradient = derivative(theta)
    last_theta = theta
    theta = theta - eta * gradient
    # 更新theta后记录它们的值
    theta_history.append(theta)

    if(abs(loss(theta) - loss(last_theta)) < epsilon):
        break

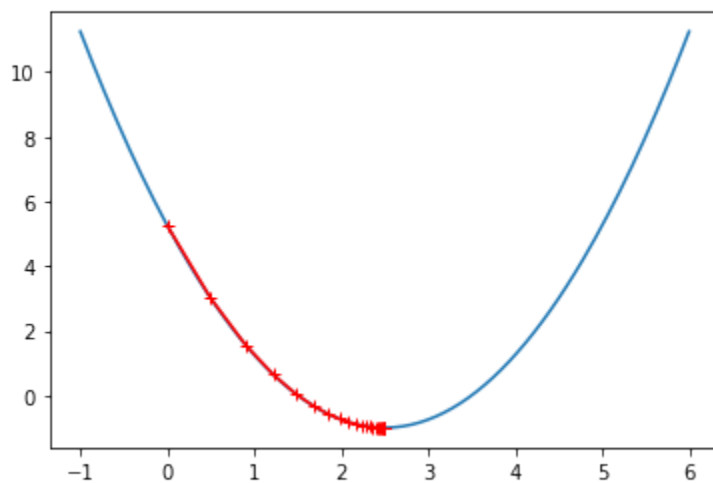
```

绘制theta每一步的变更

```

plt.plot(plot_x, loss(plot_x))
plt.plot(np.array(theta_history), loss(np.array(theta_history)),
         color="r", marker='+')
plt.show()

```



可以看到theta在下降过程中，每一步都在逐渐变小(逐渐逼近)，直到满足小于epsilon的条件。

查看一下theta_history的长度

```
len(theta_history)
```

46

我们通过梯度下降法，经过了45次查找，最终找到了theta的最小值。

为了更方便的调试eta(学习率)的值，我们对代码进行进一步的封装。把梯度下降方法和绘图分别封装为两个函数。

```
theta_history = []

def gradient_descent(initial_theta, eta, epsilon=1e-8):
    theta = initial_theta
    theta_history.append(initial_theta)

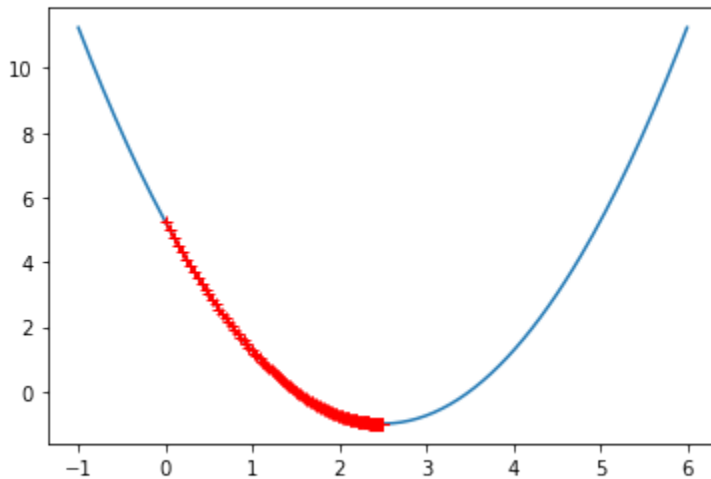
    while True:
        gradient = derivative(theta)
        last_theta = theta
        theta = theta - eta * gradient
        theta_history.append(theta)

        if(abs(loss(theta) - loss(last_theta)) < epsilon):
            break

def plot_theta_history():
    plt.plot(plot_x, loss(plot_x))
    plt.plot(np.array(theta_history), loss(np.array(theta_history)),
             color="r", marker='+')
    plt.show()
```

尝试更小的eta值

```
eta = 0.01
theta_history = []
gradient_descent(0, eta)
plot_theta_history()
```



theta下降的路径中，每一步的长度更短了，步数也更多了。

```
len(theta_history)
```

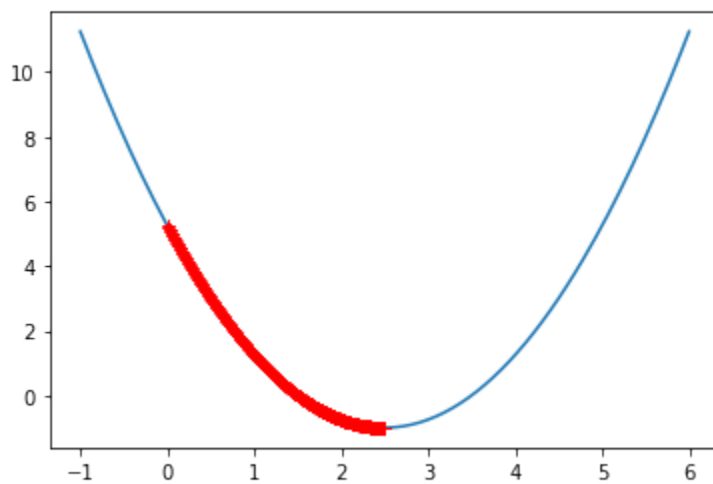
424

这次经过了424步才得到theta的最小值。

eta就是梯度下降中我们常常谈到的学习率**learn rate(lr)**。学习率的大小直接影响到梯度更新的步数。很明显，学习率越小，**theta**下降的步长越小。所需要的步数(计算次数)也就越多。

试试更小的学习率

```
eta = 0.001
theta_history = []
gradient_descent(0, eta)
plot_theta_history()
```



步长已经短到几乎连成一条粗线

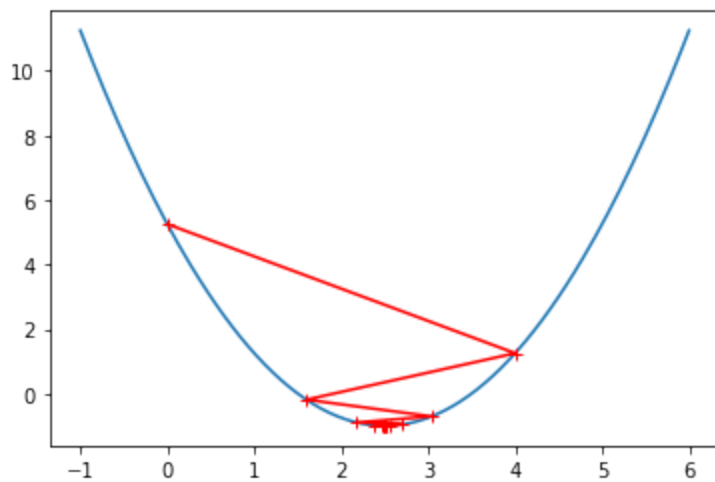
```
len(theta_history)
```

3682

共经过了3682步

那么学习率调大一些是不是更好呢？

```
eta = 0.8
theta_history = []
gradient_descent(0, eta)
plot_theta_history()
```



theta 变成了从曲线的左侧跳到了右侧，由于eta还是够小。最终我们还是计算得到了theta 的最小值。

如果更大一些的学习率，会是什么效果呢？

```
eta = 1.1
theta_history = []
gradient_descent(0, eta)
```

```
-----
OverflowError                                Traceback (most recent call last)
<ipython-input-104-89c87a003181> in <module>
      1 eta = 1.1
      2 theta_history = []
----> 3 gradient_descent(0, eta)

<ipython-input-98-46124331ca03> in gradient_descent(initial_theta, eta, epsilon)
     11     theta_history.append(theta)
     12
--> 13     if(abs(J(theta) - J(last_theta)) < epsilon):
     14         break
     15

<ipython-input-95-09073b555d68> in J(theta)
      1 def J(theta):
----> 2     return (theta-2.5)**2 - 1.
      3
      4 def dJ(theta):
      5     return 2*(theta-2.5)

OverflowError: (34, 'Result too large')
```

直接报错！

错误描述信息是：结果值太大了。

为什么会产生这样的原因呢？

原因是eta太大所导致的，theta在每次更新也会变得越来越大。梯度非但没有收敛，反而在向着反方向狂奔。直到结果值最终超出了计算机所能容纳的最大值，错误类型OverflowError(计算溢出错误)。

为避免计算值的溢出，我们改造损失值计算方法

```
def loss(theta):  
    try:  
        return (theta-2.5)**2 - 1.  
    except:  
        return float('inf') # 计算溢出时，直接返回一个float的最大值
```

给梯度更新方法添加一个最大循环次数

```
def gradient_descent(initial_theta, eta, n_iters = 1e4, epsilon=1e-8):  
  
    theta = initial_theta  
    i_iter = 0 # 初始循环次数  
    theta_history.append(initial_theta)  
  
    while i_iter < n_iters: # 小于最大循环次数  
        gradient = derivative(theta)  
        last_theta = theta  
        theta = theta - eta * gradient  
        theta_history.append(theta)  
  
        if(abs(loss(theta) - loss(last_theta)) < epsilon):  
            break  
  
        i_iter += 1 # 循环次数+1
```

重新执行刚刚会报错的

```
eta = 1.1
theta_history = []
gradient_descent(0, eta)
```

没有错误了~

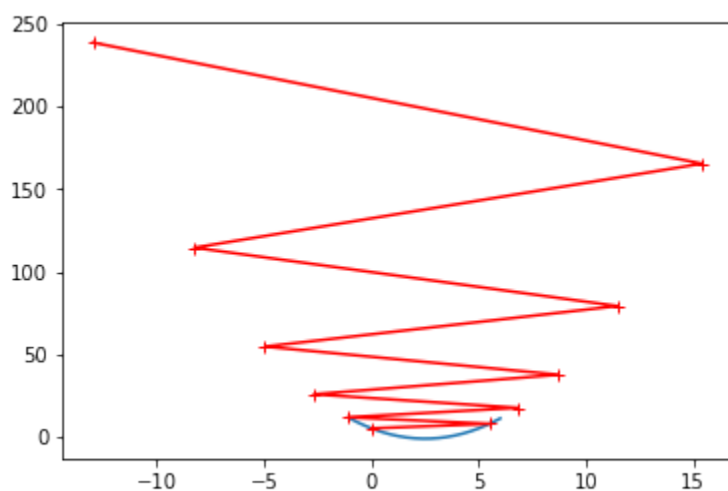
```
len(theta_history)
```

10001

达到最大循环次数后，梯度计算就停止了。

为了观察 $\eta=1.1$ 时， θ 到底是如何更新的。我们指定一个有限的梯度更新次数(10次)

```
eta = 1.1
theta_history = []
gradient_descent(0, eta, n_iters=10)
plot_theta_history()
```



theta的值从曲线出发，逐渐向外。最终越变越大.....

学习率的最佳取值

学习率 η 的取值，是不是1就是极限值呢？

实际上 η 的取值是和损失函数相关的，或者说和theta的导数相关的。所以没有一个固定标准。

所以，学习率对于梯度下降法来说也是一个超参数。也需要网格搜索来寻找。

保险的方法，是把eta先设置为0.01，然后逐渐寻找它的最佳取值。大多数函数，都是可以胜任的。

如果出现了参数值过大，也可以用上面的方法绘制图形来查看哦~

梯度更新

在得到了逻辑回归的损失函数后，我们就可以使用**梯度下降法**来寻找损失函数极小值。直白的说，就是要求出, 当 θ 取什么值时, 损失函数可以到达极小值。

要求出梯度, 我们要求 $J(\theta)$ 对于 θ_j 的偏导数 $\frac{\partial J(\theta)}{\partial \theta_j}$ 。求偏导的意义在于, 描述函数在某一点的变化率。梯度越小, 说明变化率越小, 趋近于0时, 就说明参数已经收敛了。

θ 的偏导

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_i^m \left(\frac{1}{1+e^{-\theta^T x_i}} - y_i \right) x_{ij} = \frac{1}{m} \sum_i^m (\hat{y}_i - y_i) x_{ij}$$

```
delta_theta = np.dot((y_hat-y),X) / m
```

bias 的偏导

$$\frac{\partial}{\partial bias} J(bias) = \frac{1}{m} \sum_i^m (\hat{y}_i - y_i)$$

```
delta_b = np.mean(y_hat-y)
```

实际运算中，我们带入的是向量 X 。所以最终的梯度会在求和后取均值。

逻辑回归模型构建及训练流程

1. 数据准备，参数初始化

```
from sklearn.datasets import make_classification  
  
X, y = make_classification(n_features=10)
```

关于逻辑回归的数据，有很多学习用的示例样本。这里我们使用scikit learn提供的数据集生成函数来创建。

Scikit-learn 是用 Python 开发的开源机器学习库，广泛用于数据挖掘和数据分析。

- **特点：**易用、高效，具有丰富的工具，涵盖分类、回归、聚类等多种机器学习算法。
- **功能：**提供数据预处理、模型选择、评估等功能，便于构建完整的机器学习工作流。
- **优势：**有详细文档和示例，社区活跃，能降低开发成本、提高效率。

<https://scikit-learn.org/stable/api/index.html>

```
train_X, test_X, train_y, test_y = \
    train_test_split(X, y, test_size=0.2,
                    shuffle=True, random_state=123)
```

模型训练的数据，通常还需要拆分为训练集和测试集。目的是为了防止数据泄露。

数据泄露 (Data Leakage) 指的是训练数据中包含了不应该有的信息，这些信息会使模型在评估时表现得比在真实应用场景中更好，导致对模型性能的高估，使得模型的泛化能力被错误判断。

```
# weight parameters
theta = np.random.randn(1, 10)
bias = 0

# hyper parameters
lr = 1e-3
epoch = 5000
```

模型训练中的参数分类两类，一类是权重参数，一类是超参数。

前者对模型的最终计算结果有影响，而后者往往在模型训练过程中起作用。

2. 模型运算

```
def forward(x, theta, bias):
    # linear
    z = np.dot(theta, x.T) + bias
    # sigmoid
    y_hat = 1 / (1 + np.exp(-z))
    return y_hat
```

我们把自变量，权重参数导入模型，最终计算得到结果 \hat{y} 。

这个过程也可以称为**前向运算**或**前向传播 (Forward Propagation)**。

3. 计算损失

```
def loss_function(y, y_hat):
    e = 1e-8 # 防止y_hat计算值为0, 添加的极小值epsilon
    return - y * np.log(y_hat + e) \
           - (1 - y) * np.log(1 - y_hat + e)
```

损失既是上面我们依据似然函数整理出的负对数**损失函数 (loss function)**，有些地方也称损失函数为**代价函数 (cost function)**。

注意，为防止 \hat{y} 计算结果出现0而产生错误 ($\log 0$ 无意义)，添加一个极小值 ϵ 来避免这种问题的发生。

4. 计算梯度

```
def calc_gradient(x, y, y_hat):
    m = x.shape[-1]
    delta_w = np.dot(y_hat - y, x) / m
    delta_b = np.mean(y_hat - y)
    return delta_w, delta_b
```

直接带入参数的梯度计算公式。

5. 更新参数


```
theta -= lr * dw
bias -= lr * db
```

对应公式 $w_{n+1} = w_n - \eta \frac{\partial f(w,y)}{\partial w}$

6. 重复2至5步，观察损失函数值，调整学习率

```
for i in range(epoch):
    #正向
    y_hat = forward(train_X,theta,bias)
    #计算损失
    loss = np.mean(loss_function(train_y,y_hat))
    if i % 100 == 0:
        print('step:',i,'loss:',loss)
    #梯度下降
    dw,db = calc_gradient(train_X,train_y,y_hat)
    #更新参数
    theta -= lr * dw
    bias -= lr * db
```

完整代码



```

from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.datasets import make_classification

# 生成观测值数据
X, y = make_classification(n_features=10)

# 拆分训练和测试集
train_X, test_X, train_y, test_y = \
    train_test_split(X, y, test_size=0.2,
                    shuffle=True, random_state=123)

# 初始化参数模型参数
theta = np.random.randn(1, 10)
bias = 0
# 学习率
lr = 1e-3
# 模型训练的轮数
epoch = 5000

# 前向计算
def forward(x, theta, bias):
    # linear
    z = np.dot(theta, x.T) + bias
    # z = np.dot(theta.T, x.T) + bias
    # sigmoid
    y_hat = 1 / (1 + np.exp(-z))
    return y_hat

# 损失函数
def loss_function(y, y_hat):
    e = 1e-8 # 防止y_hat计算值为0, 添加的极小值epsilon
    return - y * np.log(y_hat + e) \
        - (1 - y) * np.log(1 - y_hat + e)

# 计算梯度
def calc_gradient(x, y, y_hat):
    m = x.shape[-1]
    delta_w = np.dot(y_hat - y, x) / m
    delta_b = np.mean(y_hat - y)
    return delta_w, delta_b

for i in range(epoch):
    #正向
    y_hat = forward(train_X, theta, bias)
    #计算损失
    loss = np.mean(loss_function(train_y, y_hat))
    if i % 100 == 0:
        print('step:', i, 'loss:', loss)
    #梯度下降
    dw, db = calc_gradient(train_X, train_y, y_hat)
    #更新参数
    theta -= lr * dw
    bias -= lr * db

```

模型测试

```
# 测试模型
idx = np.random.randint(len(test_X))
x = test_X[idx]
y = test_y[idx]

def predict(x):
    pred = forward(x, theta, bias)[0]
    if pred > 0.5:
        return 1
    else:
        return 0

pred = predict(x)
print(f'预测值: {pred} 真实值: {y}')
```

Pytorch实现逻辑回归



```
"""
1. 数据准备, 参数初始化
2. 前向计算
3. 计算损失
4. 计算梯度
5. 更新参数
"""

import torch
from sklearn.datasets import load_iris
from sklearn.datasets import make_classification

# 超参数: 学习率
learn_rate = 1e-3

# 1.1 数据准备
X, y = make_classification(n_features=10)
# 创建张量
tensor_x = torch.tensor(X, dtype=torch.float)
tensor_y = torch.tensor(y, dtype=torch.float)

# 1.2 创建参数并初始化
# x [100,10] * w[1,10]
w = torch.randn(1, 10, requires_grad=True) # 初始化参数w
b = torch.randn(1, requires_grad=True) # 初始化参数b

for i in range(5000):

    # 2. 前向运算
    r = torch.nn.functional.linear(tensor_x, w, b)
    r = torch.sigmoid(r)

    # 3. 计算损失
    loss = torch.nn.functional.binary_cross_entropy(
        r.squeeze(1), tensor_y, reduction='mean')

    # 4. 计算梯度
    loss.backward()

    # 5. 参数更新
    with torch.autograd.no_grad(): # 关闭梯度计算跟踪
        # 更新权重梯度
        w -= learn_rate * w.grad
        # 清空本次计算的梯度 (因为梯度是累加计算, 不清空就累加)
        w.grad.zero_()
        # 更新偏置项梯度
        b -= learn_rate * b.grad
        # 清空本次计算的梯度
        b.grad.zero_()

    # item()张量转换python基本类型
    print(f'train loss:{loss.item():.4f}')
```


对比numpy实现逻辑回归，由于pytorch中封装了大量的函数运算实现，所以代码可以进一步简化。

- pytorch实现模型权重参数初始化
- 线性函数和sigmoid函数的pytorch实现
- pytorch提供的预定义损失函数



由于pytorch计算图中包含了自动求导的计算操作实现。所以在这个简易版本中，我们暂时不使用自动求导功能，主要是通过手动计算并更新梯度，以便于和numpy版本的代码进行对比。