# 向量函数的导数

2021 年 2 月 7 日

## 1 一元函数的导数

导数是对于函数而言的，具体而言，它是一个极限：

**定义**1.1（导数） 对于函数 $f : D \to \mathbb{R}$，设 $x_0$ 是 $D$ 的一个内点，若极限

$$\lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \tag{1.1}$$

存在，则称 $f$ 在 $x_0$ 处可导，将这个极限记为 $f'(x_0)$，并称 $f'(x_0)$ 是 $f$ 在 $x_0$ 处的导数.

对于一元函数，有一个有用的事实：

**定理**1.1（复合函数的导数） 对于函数 $f : A \to B$, $g : C \to \mathbb{R}$，设 $x \in A$，记 $y = f(x)$，那么只要 $f$ 在 $x$ 可导并且 $g$ 在 $y$ 可导，就一定可以说明复合函数 $g \circ f$ 在 $x$ 可导，并且我们有

$$(g \circ f)'(x) = g'(y)f'(x) \tag{1.2}$$

对于复合函数的写法，我们约定 $g \circ f : x \mapsto g(f(x))$.

**证明**： 设 $x'$ 在 $x$ 某邻域内：

$$f(x') = f(x) + f'(x)(x'-x) + o(x'-x) \tag{1.3}$$

由此可知当 $x'$ 足够接近 $x$ 时，存在常数 $C$，使得

$$|f(x')-f(x)| \le C\,|x'-x| \tag{1.4}$$

于是有

$$\begin{aligned} g(f(x')) &= g(f(x)) + g'(f(x))\,(f(x')-f(x)) + o\,(f(x')-f(x)) \\ &= g(f(x)) + g'(f(x))f'(x)\,(x'-x) + g'(f(x))o\,(x'-x) + o(x'-x) \\ &= g(f(x)) + g'(f(x))f'(x)\,(x'-x) + o(x'-x) \end{aligned} \tag{1.5}$$

观察式 1.5我们发现：$g'(f(x))f'(x)$ 就是 $g \circ f$ 在 $x$ 处的导数. □

## 2 向量函数及其导数

首先介绍向量：

**定义**2.1（向量） 若干个数写成一行叫做**行向量**，若干个数写成一列叫做**列向量**，行向量和列向量统称**向量**.

**例**2.1 我们称

$$\begin{bmatrix} 109 \\ 24 \end{bmatrix}$$

是一个列向量，并且说它是 2 维的．称

$$\begin{bmatrix} 10 & 201 & 34 & 75 \end{bmatrix}$$

是一个行向量，并且说它是 4 维的．它有几个元素就说它是几维的．我们暂不涉及无穷维的向量．

那么输出值（也可能有多个）依赖于多个输入值（受其影响）的函数就可以说是一个向量函数：

**定义**2.2（向量函数） 以向量作为输入，以向量作为输出的函数叫做向量函数．

**例**2.2 令

$$\begin{aligned} f : \mathbb{R}^2 &\longrightarrow \mathbb{R}^3 \\ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &\longmapsto \begin{bmatrix} x_1^2 + x_2^2 \\ x_1^2 - x_2^2 \\ x_1 x_2 \end{bmatrix} \end{aligned}$$

则这里的 $f$ 就是一个堂堂正正的向量函数．

由于向量函数有多个输出，以及多个输入，所以要完全刻画它的导数性质，非得要用到一个矩阵不可，具体地，设 $f : \mathbb{R}^m \to \mathbb{R}^n$，对于 $\mathbb{R}^m$ 中的一点 $\boldsymbol{x}$，记 $\boldsymbol{y} = f(\boldsymbol{x})$，我们这么样描述 $f$ 的导数：

$$\boldsymbol{J}^{\mathsf{T}}(f) = \begin{bmatrix} \dfrac{\partial \boldsymbol{y}}{\partial x_1} \\ \dfrac{\partial \boldsymbol{y}}{\partial x_2} \\ \vdots \\ \dfrac{\partial \boldsymbol{y}}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial y_1}{\partial x_1} & \dfrac{\partial y_2}{\partial x_1} & \cdots & \dfrac{\partial y_n}{\partial x_1} \\ \dfrac{\partial y_1}{\partial x_2} & \dfrac{\partial y_2}{\partial x_2} & \cdots & \dfrac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial y_1}{\partial x_m} & \dfrac{\partial y_2}{\partial x_m} & \cdots & \dfrac{\partial y_n}{\partial x_m} \end{bmatrix} \tag{2.1}$$

如果还有一个函数 $g : \mathbb{R}^n \to \mathbb{R}^q$，并且我们还知道

$$\boldsymbol{J}^{\mathsf{T}}(g) = \begin{bmatrix} \dfrac{\partial z_1}{\partial y_1} & \dfrac{\partial z_2}{\partial y_1} & \cdots & \dfrac{\partial z_q}{\partial y_1} \\ \dfrac{\partial z_1}{\partial y_2} & \dfrac{\partial z_2}{\partial y_2} & \cdots & \dfrac{\partial z_q}{\partial y_2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial z_1}{\partial y_n} & \dfrac{\partial z_2}{\partial y_n} & \cdots & \dfrac{\partial z_q}{\partial y_n} \end{bmatrix} \tag{2.2}$$

前面我们演示了：对于一元函数，求导运算的链式法则成立，现在用矩阵来表述多元函数的导数，我们发现链式法则对于多元函数的求导运算也成立，具体地：设 $f$ 将 $\mathbb{R}^m$ 中的 $\boldsymbol{x}$ 映射为 $\mathbb{R}^n$ 中的 $\boldsymbol{y}$，而后 $g$ 又将 $\boldsymbol{y}$ 映到 $\mathbb{R}^q$ 中的 $\boldsymbol{z}$，类似于：

$$\boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \xrightarrow{f} \boldsymbol{y} := \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \xrightarrow{g} \boldsymbol{z} := \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_q \end{bmatrix} \tag{2.3}$$

那么，我们有

$$\boldsymbol{J}_{\boldsymbol{x}}^{\mathsf{T}}(g \circ f) = \boldsymbol{J}_{\boldsymbol{x}}^{\mathsf{T}}(f) \boldsymbol{J}_{\boldsymbol{y}}^{\mathsf{T}}(g) \tag{2.4}$$

也就是说

$$\begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_2}{\partial x_1} & \cdots & \frac{\partial z_q}{\partial x_1} \\ \frac{\partial z_1}{\partial x_2} & \frac{\partial z_2}{\partial x_2} & \cdots & \frac{\partial z_q}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_m} & \frac{\partial z_2}{\partial x_m} & \cdots & \frac{\partial z_q}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \frac{\partial y_2}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix} \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_2}{\partial y_1} & \cdots & \frac{\partial z_q}{\partial y_1} \\ \frac{\partial z_1}{\partial y_2} & \frac{\partial z_2}{\partial y_2} & \cdots & \frac{\partial z_q}{\partial y_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial y_n} & \frac{\partial z_2}{\partial y_n} & \cdots & \frac{\partial z_q}{\partial y_n} \end{bmatrix} \tag{2.5}$$

我们可以通过回忆矩阵乘法的运算规则和多元函数的求导法则来理解这个式子.

# 3 梯度下降

梯度下降是指：在给定的学习率 $\alpha$ 下，按照梯度的指引，一步步地调整函数的输入，使得其输出减少的一种函数最小化的方法. 所谓的梯度其实就是导数，除了笔划数量不同并无差别. 那么既然要「按照梯度的指引」肯定首先要知道梯度，而经过我们刚才对向量函数的导数知识的学习，相信这不在话下.

## 3.1 一个简单的例子

考虑函数

$$f(x_1, x_2) = (x_1 x_2 - 3)^2 + 1 \tag{3.1}$$

找到一点 $(x_1, x_2)$ 使得 $f(x_1, x_2)$ 尽可能地小.

我们可将它看做一个 $\mathbb{R}^2$ 到 $\mathbb{R}$ 的映射，首先来计算梯度：

$$\boldsymbol{J}(f) = \begin{bmatrix} 2x_2(x_1x_2 - 3) & 2x_1(x_1x_2 - 3) \end{bmatrix} \tag{3.2}$$

然后我们首先**随机**选一点（随机就是随便的意思），比如说，我选 $\boldsymbol{x}^{(1)} = (1, 1)$，先代入看看此时的函数值是多少：

$$f(\boldsymbol{x}^{(1)}) = (1 - 3)^2 + 1 = 5 \tag{3.3}$$

设定一个学习率 $\alpha = 1 \times 10^{-2}$，然后计算 $f$ 在 $\boldsymbol{x}^{(1)}$ 的梯度：

$$\boldsymbol{J}_{\boldsymbol{x}^{(1)}}(f) = \begin{bmatrix} -4 & -4 \end{bmatrix} \tag{3.4}$$

然后更新它：

$$\begin{aligned} \boldsymbol{x}^{(2)} &= \boldsymbol{x}^{(1)} - \alpha \boldsymbol{J}_{\boldsymbol{x}^{(1)}}(f) \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix} - 1 \times 10^{-2} \begin{bmatrix} -4 & -4 \end{bmatrix} \\ &= \begin{bmatrix} 1.04 & 1.04 \end{bmatrix} \end{aligned} \tag{3.5}$$

再看此时的函数值：

$$f(\boldsymbol{x}^{(2)}) \approx 4.68025856 \tag{3.6}$$

再做一次更新：

$$\begin{aligned} \boldsymbol{x}^{(3)} &= \boldsymbol{x}^{(2)} - \alpha \boldsymbol{J}_{\boldsymbol{x}^{(2)}}(f) \\ &= \begin{bmatrix} 1.04 & 1.04 \end{bmatrix} - 1 \times 10^{-2} \begin{bmatrix} -1.995136 & -1.995136 \end{bmatrix} \\ &= \begin{bmatrix} 1.05995136 & 1.05995136 \end{bmatrix} \end{aligned} \tag{3.7}$$

再看此时的函数值：

$$f(\boldsymbol{x}^{(3)}) = 4.5212639385 \tag{3.8}$$

这个过程的重复就是梯度下降. 有时候学习率选得太大，或者初始点选得不合适，可能会导致效果不好.

## 3.2 神经网络

神经网络就可以看做是多个向量函数的复合，可以认为，神经网络的基本构成 (building-block) 是层，与之对应的数学对象也就是向量函数，为此，现在我们就来定义几个「层」，第一层添加哑节点并且对输入做线性组合：

$$\begin{aligned} f_1 : M[4,3] \times \mathbb{R}^2 &\longrightarrow \mathbb{R}^4 \\ \left(\boldsymbol{\theta}^{(1)}, \boldsymbol{x}^{(1)}\right) &\longmapsto \boldsymbol{x}^{(2)} := \boldsymbol{\theta}^{(1)} \begin{bmatrix} 1 \\ \boldsymbol{x}^{(1)} \end{bmatrix} \end{aligned} \tag{3.9}$$

其中 $M[4,3]$ 表示全体 4 行 3 列矩阵构成的集合，第三层对第而层的输出做按元素 Sigmoid 处理，并且添加一个偏置：

$$\begin{aligned} f_2 : \mathbb{R}^4 &\longrightarrow \mathbb{R}^5 \\ \boldsymbol{x}^{(2)} &\longmapsto \boldsymbol{x}^{(3)} := \begin{bmatrix} 1 \\ \text{Sigmoid}(\boldsymbol{x}^{(2)}) \end{bmatrix} \end{aligned} \tag{3.10}$$

第三层对第二层的输出做线性组合并且进行 Sigmoid 处理：

$$\begin{aligned} f_3 : M[1,5] \times \mathbb{R}^5 &\longrightarrow \mathbb{R} \\ \left(\boldsymbol{\theta}^{(2)}, \boldsymbol{x}^{(3)}\right) &\longmapsto x^{(4)} := \text{Sigmoid}\left(\left(\boldsymbol{\theta}^{(2)}\right)^{\mathsf{T}} \boldsymbol{x}^{(3)}\right) \end{aligned} \tag{3.11}$$

如果将这个神经网络的神经元一个个画出来，我们会看到图 3-1：

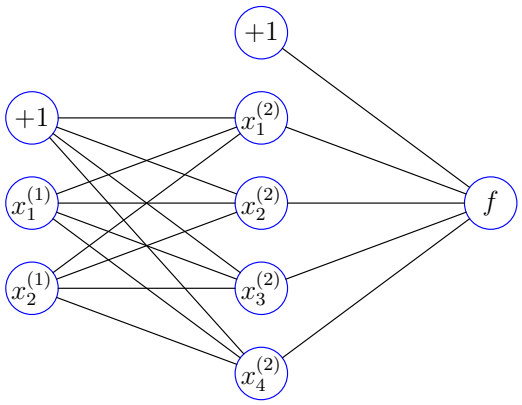


图 3-1: 神经网络示意图

如果仅仅是做预测，即假设正确的参数 $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}$ 是已知的，那么对于一个数据点 $\boldsymbol{x}^{(1)}$，仅仅计算到 $f_3$ 的输出值，是足够的，然而如果我们不知道参数 $\boldsymbol{\theta}$，就要再加一层用来计算误差，当然这仅仅是用来计算误差和做梯度下降，做预测的时候用不到这一层：

$$\begin{aligned} f_4 : \mathbb{R} &\longrightarrow \{1, 0\} \\ x^{(4)} &\longmapsto \frac{1}{2}\left(x^{(4)} - t(\boldsymbol{x}^{(1)})\right)^2 \end{aligned} \tag{3.12}$$

其中 $t(\cdot)$ 对应真实标签.

为了更加清楚地搞清楚对于这种多层的神经网络，梯度下降（也就是后馈传播）应该怎么做，我们可以按照数值与数值之间的影响关系画出一张图（图 3-2）：

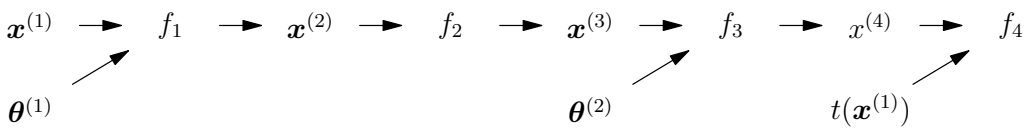


图 3-2: 计算图

从图 3-2中，我们可以看到：$\boldsymbol{x}^{(1)}$ 和 $\boldsymbol{\theta}^{(1)}$ 共同透过 $f_1$ 影响着 $\boldsymbol{x}^{(2)}$，而 $\boldsymbol{x}^{(2)}$ 透过 $f_2$ 影响 $x^{(3)}$，由于 $f_2$ 其实是按元素 Sigmoid，所以没有参数，然后 $\boldsymbol{x}^{(3)}$ 和 $\boldsymbol{\theta}^{(2)}$ 共同影响 $x^{(4)}$，也就是神经网络最终输出的概率，而 $t(\boldsymbol{x}^{(1)})$（也就是真实标签）和 $x^{(4)}$（也就是输出概率值）共同影响着 $f_4$ 的计算结果（也就是损失值）. 我们希望通过链式法则计算出 $f_4$ 的输出值分别关于 $\boldsymbol{\theta}^{(2)}$ 与 $\boldsymbol{\theta}^{(1)}$ 的导数（也就是梯度），然后根据这个梯度和学习率一起，去微调之，从而使得 $f_4$ 的输出值降低.

## 3.3 梯度的计算

我们的目的是找到 $\boldsymbol{\theta}^{(1)}$ 和 $\boldsymbol{\theta}^{(2)}$ 使得对于 $\boldsymbol{x}^{(1)}$ 和 $t(\boldsymbol{x}^{(1)})$ 的取值分别如表 3-1所示的时候

表 3-1: 自变量和标签的取值

| $\boldsymbol{x}^{(1)}$ | $t(\boldsymbol{x}^{(1)})$ |
|---|---|
| $(0,0)$ | 0 |
| $(0,1)$ | 1 |
| $(1,0)$ | 1 |
| $(1,1)$ | 0 |

$f_4$ 的输出都比较小. 为此，我们这样做：固定 $\boldsymbol{x}^{(1)}$ 和 $t(\boldsymbol{x}^{(1)})$，然后 $f_4$ 关于各 $\boldsymbol{\theta}$ 的梯度，然后根据这些梯度去微调 $\boldsymbol{\theta}$ 使得 $f_4$ 下降（梯度下降），然后让 $\boldsymbol{x}^{(1)}$ 和 $t(\boldsymbol{x}^{(1)})$ 都取到表格的下一行（如果已是最后一行就返回第一行），重复这些步骤直到 $f_4$ 的输出相当小.

观察图3-2，我们发现：顺着箭头方向的计算都是比较容易进行，稍微有点麻烦的其实是求导，如果只是求 $f$ 对于 $\boldsymbol{x}$ 的导数还好说，稍微有点麻烦的是对于那些 $\boldsymbol{\theta}$ 的导数，$\boldsymbol{\theta}$ 是矩阵，而我们只知道在「以向量作为输入」且「以向量作为输出」的情况下如何求导，那么如果是「以矩阵作为」输入的时候该怎么办呢？我们主要是将矩阵也视作一个折叠起来的向量，要将矩阵写成一个向量，我们首先取矩阵的第一行，然后取第二行接在第一行的后头，然后第三行接在第二行的后头，这样一来，就建立了有限维矩阵和向量之间的一一对应关系. 譬如说，对于矩阵

$$\begin{bmatrix} 14 & 32 & 66 \\ 72 & 86 & 97 \end{bmatrix} \tag{3.13}$$

我们就将它写成一个行向量

$$\begin{bmatrix} 14 & 32 & 66 & 72 & 86 & 97 \end{bmatrix} \tag{3.14}$$

有了这种对应关系，就可以求一个函数的向量输出关于矩阵输入的导数，例如说，对于函数

$$\begin{gathered} \boldsymbol{g} : M[2,3] \longrightarrow \mathbb{R}^2 \\ \boldsymbol{x} := \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} \longmapsto \begin{bmatrix} g_1(\boldsymbol{x}) \\ g_2(\boldsymbol{x}) \end{bmatrix} \end{gathered} \tag{3.15}$$

它关于 $\boldsymbol{x}$ 的导数就是

$$\boldsymbol{J}_{\boldsymbol{x}}(\boldsymbol{g}) = \begin{bmatrix} \dfrac{\partial g_1}{\partial x_{11}} & \dfrac{\partial g_1}{\partial x_{12}} & \dfrac{\partial g_1}{\partial x_{13}} & \dfrac{\partial g_1}{\partial x_{21}} & \dfrac{\partial g_1}{\partial x_{22}} & \dfrac{\partial g_1}{\partial x_{23}} \\ \dfrac{\partial g_2}{\partial x_{11}} & \dfrac{\partial g_2}{\partial x_{12}} & \dfrac{\partial g_2}{\partial x_{13}} & \dfrac{\partial g_2}{\partial x_{21}} & \dfrac{\partial g_2}{\partial x_{22}} & \dfrac{\partial g_2}{\partial x_{23}} \end{bmatrix} \tag{3.16}$$

如此一来就没有什么本质上的困难了.

现在回到图 3-2，为了计算 $\boldsymbol{J}(f_1)$，我们主要关注 $\boldsymbol{x}^{(2)}$，它等于

$$\boldsymbol{x}^{(2)} := \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \\ x_4^{(2)} \end{bmatrix} = \boldsymbol{\theta}^{(1)} \begin{bmatrix} 1 \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} \theta_{1,0}^{(1)} & \theta_{1,1}^{(1)} & \theta_{1,2}^{(1)} \\ \theta_{2,0}^{(1)} & \theta_{2,1}^{(1)} & \theta_{2,2}^{(1)} \\ \theta_{3,0}^{(1)} & \theta_{3,1}^{(1)} & \theta_{3,2}^{(1)} \\ \theta_{4,0}^{(1)} & \theta_{4,1}^{(1)} & \theta_{4,2}^{(1)} \end{bmatrix} \begin{bmatrix} 1 \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{2} \theta_{1,j}^{(1)} x_j^{(1)} \\ \sum_{j=0}^{2} \theta_{2,j}^{(1)} x_j^{(1)} \\ \sum_{j=0}^{2} \theta_{3,j}^{(1)} x_j^{(1)} \\ \sum_{j=0}^{2} \theta_{4,j}^{(1)} x_j^{(1)} \end{bmatrix} \tag{3.17}$$

其中我们规定 $x_0^{(k)}$ 都等于 0．现在我们可以写出

$$\boldsymbol{J}_{\boldsymbol{x}^{(1)}}^{\mathsf{T}}(f_1) = \begin{bmatrix} \theta_{1,1}^{(1)} & \theta_{2,1}^{(1)} & \theta_{3,1}^{(1)} & \theta_{4,1}^{(1)} \\ \theta_{1,2}^{(1)} & \theta_{2,2}^{(1)} & \theta_{3,2}^{(1)} & \theta_{4,2}^{(1)} \end{bmatrix} \tag{3.18}$$

以及

$$\boldsymbol{J}_{\boldsymbol{\theta}^{(1)}}(f_1) = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & & & & & & & & & \\ & & & 1 & x_1^{(1)} & x_2^{(1)} & & & & & & \\ & & & & & & 1 & x_1^{(1)} & x_2^{(1)} & & & \\ & & & & & & & & & 1 & x_1^{(1)} & x_2^{(1)} \end{bmatrix} \tag{3.19}$$

$$= \operatorname{diag}\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}\} \tag{3.20}$$

接下来计算 $f_2$ 的梯度，首先我们知道

$$\boldsymbol{x}^{(3)} := \begin{bmatrix} x_0^{(3)} \\ x_1^{(3)} \\ x_2^{(3)} \\ x_3^{(3)} \\ x_4^{(3)} \end{bmatrix} = \begin{bmatrix} 1 \\ \text{Sigmoid}(x_1^{(2)}) \\ \text{Sigmoid}(x_2^{(2)}) \\ \text{Sigmoid}(x_3^{(2)}) \\ \text{Sigmoid}(x_4^{(2)}) \end{bmatrix} \tag{3.21}$$

记 $s := \text{Sigmoid}$

$$\boldsymbol{J}_{\boldsymbol{x}^{(2)}}(f_2) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ s(x_1^{(2)})(1 - s(x_1^{(2)})) & 0 & 0 & 0 \\ 0 & s(x_2^{(2)})(1 - s(x_2^{(2)})) & 0 & 0 \\ 0 & 0 & s(x_3^{(2)})(1 - s(x_3^{(2)})) & 0 \\ 0 & 0 & 0 & s(x_4^{(2)})(1 - s(x_4^{(2)})) \end{bmatrix} \tag{3.22}$$

接下来我们来计算 $x^{(4)}$

$$x^{(4)} = \text{Sigmoid}\left( \begin{bmatrix} \theta_{1,0}^{(2)} & \theta_{1,1}^{(2)} & \theta_{1,2}^{(2)} & \theta_{1,3}^{(2)} & \theta_{1,4}^{(2)} \end{bmatrix} \begin{bmatrix} x_0^{(3)} \\ x_1^{(3)} \\ x_2^{(3)} \\ x_3^{(3)} \\ x_4^{(3)} \end{bmatrix} \right) = \text{Sigmoid}\left( \sum_{j=0}^{4} \theta_{1,j}^{(2)} x_j^{(3)} \right) \tag{3.23}$$

其中 $x_0^{(3)}=1$. 然后计算雅克比矩阵，首先计算关于 $\boldsymbol{x}^{(3)}$ 的

$$\boldsymbol{J}_{\boldsymbol{x}^{(3)}}(f_3)=\begin{bmatrix}\theta_{1,0}^{(2)}s(1-s) & \theta_{1,1}^{(2)}s(1-s) & \theta_{1,2}^{(2)}s(1-s) & \theta_{1,3}^{(2)}s(1-s) & \theta_{1,4}^{(2)}s(1-s)\end{bmatrix} \tag{3.24}$$

然后计算关于 $\boldsymbol{\theta}^{(2)}$ 的

$$\boldsymbol{J}_{\boldsymbol{\theta}^{(2)}}(f_3)=\begin{bmatrix}s(1-s) & x_1^{(3)}s(1-s) & x_1^{(3)}s(1-s) & x_1^{(3)}s(1-s) & x_1^{(3)}s(1-s)\end{bmatrix} \tag{3.25}$$

这里的 $s$ 等于 $\text{Sigmoid}\left(\sum_{j=0}^{4}\theta_{1,j}^{(2)}x_j^{(3)}\right)$. 可以看到梯度（雅克比矩阵）的计算主要是在前向传播时进行. 接下来我们主要关心 $f_4$ 关于 $x^{(4)}$ 的梯度，首先有

$$f_4(x^{(4)})=\frac{1}{2}\left(x^{(4)}-t(\boldsymbol{x}^{(1)})\right)^2 \tag{3.26}$$

那么就可以写出导数:

$$\frac{\mathrm{d}f_4}{\mathrm{d}x^{(4)}}=x^{(4)}-t(\boldsymbol{x}^{(1)}) \tag{3.27}$$

导数的符号计算暂时告一段落.

## 3.4 编程实现

我们并不打算给出哪怕是稍微完整一点的神经网络的实现，我们主要是想展现损失函数随着梯度下降的进行而下降的现象，这样做对于理解梯度下降有帮助.

首先是初始化一个神经网络，主要是初始化它的参数:

```
import numpy as np
from scipy.linalg import block_diag

class NeuralNet:

    def __init__(self):

        self.x1 = np.zeros(shape=(2,1,), dtype=np.float)
        self.x2 = np.zeros(shape=(4,1,), dtype=np.float)
        self.x3 = np.zeros(shape=(5,1,), dtype=np.float)
        self.x4 = np.zeros(shape=(1,1,), dtype=np.float)
        self.x5 = np.zeros(shape=(1,1,), dtype=np.float)

        self.theta1 = np.zeros(shape=(4,3,), dtype=np.float)
        self.theta2 = np.zeros(shape=(1,5,), dtype=np.float)
        self.tx1 = np.zeros(shape=(1,1), dtype=np.float)

        self.jacobian_f1_x1 = np.zeros(shape=(4,2,), dtype=np.float)
        self.jacobian_f1_theta1 = np.zeros(shape=(4,12,), dtype=np.float)
        self.jacobian_f2_x2 = np.zeros(shape=(5,4,), dtype=np.float)
        self.jacobian_f3_x3 = np.zeros(shape=(1,5,), dtype=np.float)
        self.jacobian_f3_theta2 = np.zeros(shape=(1,5,), dtype=np.float)
        self.jacobian_f4_x4 = np.zeros(shape=(1,1,), dtype=np.float)
```

符号与我们上面的数学推导所用到的基本一致. 然后定义前馈过程，主要是计算雅克比矩阵和误差:

```
def feed_forward(self, x: np.ndarray, y: np.ndarray) -> np.ndarray:
    if self.x1.shape == x.shape and self.tx1.shape == y.shape:
        self.x1 = x
```

```
        self.tx1 = y

        self.x2 = self.x1
        self.x2 = np.concatenate(
            ([[1]], self.x2),
            axis=0
        )
        self.x2 = self.theta1 @ self.x2
        self.jacobian_f1_x1 = self.theta1[:, [1,2]].copy()

        temp = np.concatenate(([[1]], self.x1,), axis=0).T
        self.jacobian_f1_theta1 = block_diag(temp, temp, temp, temp)

        self.x3 = self.x2
        self.x3 = 1 / (1 + np.exp(0-self.x3))
        self.x3 = np.concatenate(
            ([[1]], self.x3,),
            axis=0
        )

        self.jacobian_f2_x2 = (self.x3 * (1 - self.x3)).flatten()
        self.jacobian_f2_x2 = np.diag(self.jacobian_f2_x2)[:, 1:]

        temp = self.theta2 @ self.x3
        self.x4 = 1 / (1 + np.exp(0-temp))

        self.jacobian_f3_x3 = self.theta2 * self.x4 * (1-self.x4)
        self.jacobian_f3_theta2 = self.x3.T * self.x4 * (1-self.x4)

        self.x5 = (1/2)*np.power(self.x4-self.tx1, 2)
        self.jacobian_f4_x4 = self.x4 - self.tx1

        return self.x5
    else:
        raise ValueError()
```

算好了雅克比矩阵就可以进行误差逆传播了：

```
def back_propagate(self) -> np.ndarray:
    theta2_gradient = self.jacobian_f4_x4 @ self.jacobian_f3_theta2
    theta1_gradient = self.jacobian_f4_x4 @ \
    self.jacobian_f3_x3 @ \
    self.jacobian_f2_x2 @ \
    self.jacobian_f1_theta1
    theta1_gradient = theta1_gradient.reshape(4,3)

    return (theta1_gradient, theta2_gradient,)
```

误差逆传播给出的其实是梯度，所以我们还要定义一个过程去更新参数：

```
def update_parameter(self, lr: float = 0.02) -> None:
    theta1_gradient, theta2_gradient = self.back_propagate()
    self.theta1 = self.theta1 - lr * theta1_gradient
```

```
    self.theta2 = self.theta2 - lr * theta2_gradient
```

当然了，在真正开始训练之前，还有随机设置它的参数，通过一个参数设置方法：

```
def set_parameter(self, theta1: np.ndarray, theta2: np.ndarray):
    self.theta1 = theta1
    self.theta2 = theta2
```

现在可以调用这些函数了：

```
net = NeuralNet()
theta1 = np.random.rand(4,3,)
theta2 = np.random.rand(1,5,)
net.set_parameter(theta1, theta2)

for i in range(100000):
    x1 = np.array([[0], [0]])
    y = np.array([[0]])
    e1 = net.feed_forward(x1, y)[0,0]
    net.back_propagate()
    net.update_parameter()

    x1 = np.array([[0], [1]])
    y = np.array([[1]])
    e2 = net.feed_forward(x1, y)[0,0]
    net.back_propagate()
    net.update_parameter()

    x1 = np.array([[1], [0]])
    y = np.array([[1]])
    e3 = net.feed_forward(x1, y)[0,0]
    net.back_propagate()
    net.update_parameter()

    x1 = np.array([[1], [1]])
    y = np.array([[0]])
    e4 = net.feed_forward(x1, y)[0,0]
    net.back_propagate()
    net.update_parameter()

    if (i+1) % 10000 == 0:
        error = e1 + e2 + e3 + e4
        print(error)
```

如果不收敛，建议多试几组随机参数.

# 4 总结

在本篇文章中，我们主要讲了向量函数的链式求导法则及其应用，尤其是，通过以神经网络为例，讲解了向量函数的求导在误差逆传播算法中的体现，我们由衷地希望您通过读懂这篇文章使得自己无论是在对微积分知识的理解上还是在对神经网络梯度下降算法的理解上都能够有看得见的提升！