深度学习中的矩阵微积分

Terence Parr and Jeremy Howard

July 3, 2018
译者：being
2022-05-25

(We teach in University of San Francisco's MS in Data Science program and have other nefarious projects underway. You might know Terence as the creator of the ANTLR parser generator. For more material, see Jeremy's fast.ai courses and University of San Francisco's Data Institute in person version of the deep learning course.) HTML version (The PDF and HTML were generated from markup using bookish)

摘　要

本文试图解释所有你需要的矩阵微积分，以理解深度学习的训练过程。我们假设你已经具有微积分 1 的数学基础，请注意在你开始学习训练和使用深度学习之前，你不需要了解本材料。这份资料是为那些已经熟悉神经网络基础，并希望加深对基础数学的理解的人准备的。不要担心如果你在学习过程中被卡住了--只要回头重读一下上一节，并且试着写下一些例子并加以研究。如果你仍然被卡住，我们很乐意在论坛.fast.ai 的理论类别中回答你的问题。注意：在本文末尾有一个参考资料部分，总结了所有关键的矩阵计算规则和术语在此讨论。

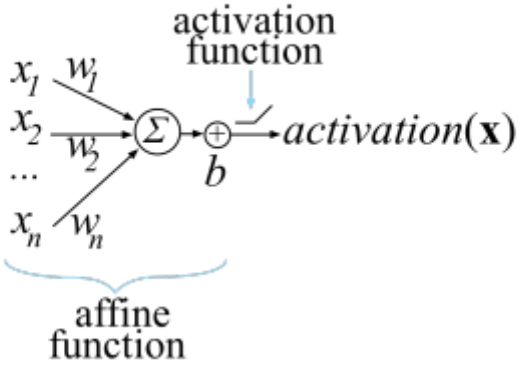
目录

一、 介绍

我们大多数人最后一次看到微积分是在学校，但导数是机器学习的一个关键部分。特别是深度神经网络，它是通过优化损失函数来进行训练的。拿起一篇机器学习论文或 PyTorch 等库的文档，微积分就会就像节日里的远房亲戚一样飞奔着来到了你的生活。而且，这可不是普通的标量微积分而是矩阵微积分，是线性代数和多变量微积分的婚礼。

好吧.....也许需要这个词并不恰当：Jeremy's 的课程展示了成为世界级的深度学习从业者，却只需要最低水平的标量微积分，这要归功于利用现代的深度学习深度学习库。但如果你真的想真正了解在这些库的引擎盖下发生了什么，并探索讨论模型训练技术的学术论文的最新进展，你就需要了解矩阵计算领域的某些部分。

例如，神经网络中单个计算单元的激活通常是通过以下方式计算的：使用边缘权重向量 w 与输入向量 x 的点积（矩阵乘法）加上一个标量偏置（阈值）：$z(x) = \Sigma_i^n w_i x_i + b = w \cdot x + b$。Z（x）函数被称作单元仿射函数，紧随其后的是一个整流的线性单元，它将负值变为零：max(0, z(x))。这样的计算单元有时被称为 "人工神经元"，看起来像这样。



神经网络由许多这样的单元组成，组织成多个神经元的集合，称为层。一个层的各个单元的激活函数的输出成为下一个层的单元的输入。各层的输出均可被称为网络输出。

训练这个神经元意味着选择权重 w 和偏移 b，以便我们在所有 N 个输入 x 的情况下得到期望的输出。要做到这一点，我们要最小化一个损失函数，将网络的预测值（$\hat{y}$）与实际值（已知的 y）进行比较。为了最小化损失，我们使用梯度下降的一些变体，如普通的随机梯度下降（SGD）、带动量的 SGD 或 Adam。动量的 SGD，或者 Adam。所有这些都需要激活（x）的部分导数（梯度）。我们的目标是逐步调整 w 和 b，使总体损失函数不断变小。

如果我们小心的话，我们可以通过微分一个普通损失的标量版本来得出梯度（均方误差）

$$\frac{1}{N}\sum_{\mathbf{x}}(target(\mathbf{x}) - activation(\mathbf{x}))^2 = \frac{1}{N}\sum_{\mathbf{x}}(target(\mathbf{x}) - max(0, \sum_{i}^{|x|} w_i x_i + b))^2$$

但这只是一个神经元，神经网络必须同时训练所有层的所有神经元的权重和偏差。因为有多个输入和（潜在的）多个网络输出，我们确实需要一个相对于一个向量的导数的一般规则，甚至是矢量值函数相对于矢量的导数的规则。

本文将介绍一些重要规则的推导：用于计算相对于向量的偏导数的，特别是那些对训练神经网络有用的规则。一个“被广为人知”领域--矩阵微积分，好消息是，我们只需要该领域的一小部分。虽然网上有很多关于多元微积分和线性代数的资料，但它们通常是作为两门独立的本科课程来教授的。所以大多数材料都是孤立地对待它们。那些讨论矩阵微积分的网页往往只是列出了一些规则，而没有什么解释或者只是部分解释。同时由于他们使用了密集的符号，几乎不对基础概念进行讨论(见最后的资源注释列表)，除了一小部分数学家之外，这些往往对所有的人都很晦涩。

与此相反，我们将重拾一些关键的矩阵微积分规则，以试图解释它们。事实证明，矩阵微积分真的没有那么难！并没有一打的新规则需要去学习，只有一些关键的概念。我们希望这篇短文能让你快速入门迅速进入矩阵微积分的世界，因为它与训练神经网络有关。我们的假设是你已经熟悉了神经网络结构和训练的基础知识。如果你不熟悉。去 Jeremy’s 的课程，完成其中的第一部分。当你完成后，我们会在这里见到你。(请注意，与许多学术方法不同的是，我们强烈建议首先学习如何训练并在实践中学会使用神经网络，然后再学习基础数学。因为在有前置知识背景的情况下，数学会更容易理解；此外，没有必要为了成为一个有效的实践者而摸索所有这些微积分）。

关于记号的说明：Jeremy’s 的课程完全使用代码，而不是数学符号，来解释概念，因为代码中不熟悉的函数很容易被搜索和实验。但在本文中中，我们的做法正好相反：有很多数学符号，因为本文的目标之一是帮助你理解你将在深度学习论文和书籍中看到的符号。在本文的最后的末尾，你会发现一个关于所使用的符号的简表，包括你可以用一个单词或短语来搜索更多的细节。

二、 矢量微积分和偏导数的介绍

神经网络层不是单一参数 f(x)的单一函数。因此，让我们继续讨论多参数的函数，如 f(x, y)。例如，什么是 xy 的导数（即 x 和 y 的乘法）？换句话说，当我们改变变量时，乘积 xy 是如何变化的？嗯，这取决于我们是改变 x 还是 y。我们一次计算一个变量（参数）的导数，得到两个不同的偏导数（一个是 x，一个是 y）。注意偏导运算符是 $\frac{\partial}{\partial x}$。所以，$\frac{\partial}{\partial x} f(x,y)$ 和 $\frac{\partial}{\partial x} f(x,y)$ 是 xy 的偏导数。通常，这些都被称为偏导。对于单一参数的函数，算子 $\frac{\partial}{\partial x}$ 是相当于 $\frac{d}{dx}$（对于足够平滑的函数）。然而，最好使用 $\frac{d}{dx}$ 来表明你指的是一个标量导数。

关于 x 的偏导只是通常的标量导数，只是把方程中的其他变量当作常数。为了明确我们是在做矢量微积分，而不仅仅是多变量微积分，让我们考虑如何计算 f(x, y)的偏导数 $\frac{\partial f(x,y)}{\partial x}$ 和 $\frac{\partial f(x,y)}{\partial y}$。把它们组织成一个水平向量，而不只是单独存在。我们称这个向量为 f(x, y)的梯度，并将其写为。

$$\nabla f(x,y) = [\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}] = [6yx, 3x^2]$$

所以 f(x, y)的梯度只是一个它参数的向量。梯度是矢量微积分世界的一部分，它处理的是将 n 个标量参数映射到一个标量的矩阵。现在，让我们疯狂一下，同时考虑向量函数的导数。

三、 矩阵微积分

当我们从一个函数的导数转移到多个函数的导数时，我们就从矢量微积分的世界转移到了矩阵微积分。让我们来计算两个函数的部分导数，这两个函数都有两个参数。我们可以保留上一节中的 $f(x, y) = 3x^2y$，但我们也可以引入 $g(x, y) = 2x + y^8$。g 的梯度有两个元素，是关于参数的偏导数。

$$\frac{\partial g(x,y)}{\partial x} = \frac{\partial 2x}{\partial x} + \frac{\partial y^8}{\partial x} = 2\frac{\partial x}{\partial x} + 0 = 2 \times 1 = 2$$

and

$$\frac{\partial g(x,y)}{\partial y} = \frac{\partial 2x}{\partial y} + \frac{\partial y^8}{\partial y} = 0 + 8y^7 = 8y^7$$

因此梯度$\nabla g(x, y) = [2, 8y^7]$ 。

梯度向量一个特定标量函数的所有偏导数构成。如果我们有两个函数，我们也可以通过堆叠梯度把它们的梯度组织成一个矩阵。当我们这样做的时候，我们就会得到雅各布矩阵（或者仅仅是雅各布矩阵），其中梯度是行。

$$J = \begin{bmatrix} \nabla f(x,y) \\ \nabla g(x,y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} 6yx & 3x^2 \\ 2 & 8y^7 \end{bmatrix}$$

欢迎来到矩阵微积分!。

3.1 雅各布矩阵的一般化

到目前为止，我们已经看了一个雅各布矩阵的具体例子。为了更广泛地定义雅各布矩阵的定义，让我们把多个参数合并成一个矢量参数：$f(x, y, z) \Rightarrow f(\mathbf{x})$。我们还必须为向量 x 定义一个方向。我们假设所有的向量都是列向量，大小为 n×1。

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

对于多个标量值的函数，我们可以将它们全部合并成一个向量，就像我们对参数做的一样。$\mathbf{y} = f(\mathbf{x})$是一个由 m 个标量值函数组成的向量，每个标量值函数取一个 $\mathbf{x}$（长度为 n）。其中每个 f_i 函数都返回一个标量，就像上一节中所说的那样。

$$\begin{aligned} y_1 &= f_1(\mathbf{x}) \\ y_2 &= f_2(\mathbf{x}) \\ &\vdots \\ y_m &= f_m(\mathbf{x}) \end{aligned}$$

所以，雅各布矩阵是所有 m×n 个可能的偏导的集合（m 行和 n 列）。也就是关于 **x** 的 m 个梯度的堆叠。（译者注：请注意这里是讲解的是向量的微积分）

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \cdots \\ \nabla f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} f_1(\mathbf{x}) \\ \frac{\partial}{\partial \mathbf{x}} f_2(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial \mathbf{x}} f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}) \\ & & \cdots & \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \frac{\partial}{\partial x_2} f_m(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix}$$

3.2 矢量元素之间二元运算的导数

向量上的逐元素二元运算，如加法 **w+x**。这是很重要的，因为我们可以将许多常见的向量运算，如向量与标量的乘法，表示为逐元素二元运算，这是指将一个运算符应用于两个向量的所有相同位置元素，这些基本数学运算符在 numpy 或 tensorflow 中都是默认应用的（例如矩阵主元素相乘*）。例如 w>x（返回一个 1 和 0 的向量）。我们可以用 y=f(w)○g(x)的符号来概括元素间的操作，这个符号表示任何元素的运算符（如+）。下面是方程 **y = f(w)○g(x)**，当我们将其写成标量方程时，是这样的。

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x}) \\ f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x}) \end{bmatrix}$$

利用上一节的思路，我们可以看到，关于 **w** 的雅各布矩阵：

$$J_{\mathbf{w}} = \frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \cdots & \frac{\partial}{\partial w_n}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial w_1}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \cdots & \frac{\partial}{\partial w_n}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ & \cdots & & \\ \frac{\partial}{\partial w_1}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \cdots & \frac{\partial}{\partial w_n}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

关于 **x** 的雅各布矩阵：

$$J_{\mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial}{\partial x_1}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \cdots & \frac{\partial}{\partial x_n}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial x_1}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \cdots & \frac{\partial}{\partial x_n}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ & \cdots & & \\ \frac{\partial}{\partial x_1}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \cdots & \frac{\partial}{\partial x_n}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

这个矩阵看起来相当臃肿，但幸运的是，雅各布矩阵通常是一个对角矩阵：除了

对角线以外，其他地方都是零。因此这极大地简化了雅各布矩阵，让我们来研究一下，其如何变为对角矩阵以及当雅各布矩阵减少到对角线矩阵的时候，如何进行逐元素元素的操作。

在在什么条件下这些对角线外的元素是零？当f_i（**w**）g_i（**x**）均不是 w_j 的函数时，这些偏导为零，$\frac{\partial}{\partial w_j} f_i(w) = \frac{\partial}{\partial w_j} g_i(x) = 0$，所以无论哪种算子，如果对应偏导数为零，那么运算结果为零，$0 \bigcirc 0 = 0$。

最终，f_i（**w**）g_i（**x**）还原为f_i（w_i）g_i（x_i），我们将在后面利用这一简化。

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1}(f_1(w_1) \bigcirc g_1(x_1)) & & & \\ & \frac{\partial}{\partial w_2}(f_2(w_2) \bigcirc g_2(x_2)) & \cdots & 0 \\ 0 & & & \frac{\partial}{\partial w_n}(f_n(w_n) \bigcirc g_n(x_n)) \end{bmatrix}$$

(The large "0"s are a shorthand indicating all of the off-diagonal are 0.)

更简洁的说，我们可以写成：

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = diag\left(\frac{\partial}{\partial w_1}(f_1(w_1) \bigcirc g_1(x_1)),\ \frac{\partial}{\partial w_2}(f_2(w_2) \bigcirc g_2(x_2)),\ \ldots,\ \frac{\partial}{\partial w_n}(f_n(w_n) \bigcirc g_n(x_n))\right)$$

其中 diag(**w**)构成一个对角矩阵。

因为我们会做很多简单的向量运算，二元运算中的函数通常 f(**w**)等于向量 **w** 可以还原为 $f_i(w_i) = w_i$. 例如，矢量加法 **w**+**x** 符合我们雅克比对角矩阵条件，因为 f(**w**)+g(**x**)有标量方程 y_i=f_i(**w**)+g_i(**x**)，其偏导数为：

$$\frac{\partial}{\partial w_i}(f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial w_i}(w_i + x_i) = 1 + 0 = 1$$

$$\frac{\partial}{\partial x_i}(f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial x_i}(w_i + x_i) = 0 + 1 = 1$$

鉴于这种特殊情况的简单性，您应该能够推导出向量上常见的元素二元运算的雅可比矩阵：

Op	Partial with respect to w
$+$	$\frac{\partial(\mathbf{w}+\mathbf{x})}{\partial \mathbf{w}} = diag(\ldots \frac{\partial(w_i+x_i)}{\partial w_i} \ldots) = diag(\vec{1}) = I$
$-$	$\frac{\partial(\mathbf{w}-\mathbf{x})}{\partial \mathbf{w}} = diag(\ldots \frac{\partial(w_i-x_i)}{\partial w_i} \ldots) = diag(\vec{1}) = I$
$\otimes$	$\frac{\partial(\mathbf{w}\otimes\mathbf{x})}{\partial \mathbf{w}} = diag(\ldots \frac{\partial(w_i\times x_i)}{\partial w_i} \ldots) = diag(\mathbf{x})$
$\oslash$	$\frac{\partial(\mathbf{w}\oslash\mathbf{x})}{\partial \mathbf{w}} = diag(\ldots \frac{\partial(w_i/x_i)}{\partial w_i} \ldots) = diag(\ldots \frac{1}{x_i} \ldots)$

⊗和⊘运算符是元素间的乘法和除法。

3.3 涉及标量扩展的导数（广播）

当我们将标量与向量相乘或相加时，我们隐含地将标量扩展为一个向量然后进行一个从元素出发的二进制运算。例如，将标量 z 加入到向量 **x** 中。**y = x** + z，实际上是 y = f(x) + g(z) 其中 f(x) = x，g(z) = **1**z。一个适当长度的 1 的向量）。z 是任何不依赖 x 的标量，这很有用,因为这样 g(z)对于任意 x 的偏导数为 0，这将简化我们的偏导计算。

3.4 以标量形式还原矢量求和

将一个向量的元素相加是深度学习中的一个重要操作，比如说网络损失函数，但我们也可以用它来简化计算向量点积的导数和其他将向量简化为标量的操作。让 y = sum(**f(x)**)（译者注：请注意这里 y 是一个标量）。矢量求和的梯度（1×n 雅各布矩阵）是：

$$\begin{aligned}
\frac{\partial y}{\partial \mathbf{x}} &= \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}\right] \\
&= \left[\frac{\partial}{\partial x_1}\sum_i f_i(\mathbf{x}),\ \frac{\partial}{\partial x_2}\sum_i f_i(\mathbf{x}),\ \ldots,\ \frac{\partial}{\partial x_n}\sum_i f_i(\mathbf{x})\right] \\
&= \left[\sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_1},\ \sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_2},\ \ldots,\ \sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_n}\right] \qquad \text{(move derivative inside } \textstyle\sum\text{)}
\end{aligned}$$

让我们来看看简单的 y = sum(**x**)（**f(x)=x**）的梯度。

$$\nabla y = \left[\frac{\partial x_1}{\partial x_1},\ \frac{\partial x_2}{\partial x_2},\ \ldots,\ \frac{\partial x_n}{\partial x_n}\right] = [1, 1, \ldots, 1] = \vec{1}^T$$

注意，结果是一个充满 1 的水平向量，而不是一个垂直向量，所以梯度是 $\mathbf{1}^T$。保持你所有的向量和矩阵的形状是非常重要的，否则就不可能计算出复数函数的导数。

3.5 向量链式法则

我们准备解决向量值函数和向量的链式规则。令人惊讶的是，这个更一般的连锁规则和标量的单变量连锁法则一样简单。与其说是在介绍矢量连锁法则，不如让我们

自己重新发现它，这样我们就能牢牢掌握它了。我们可以从以下方面入手计算一个向量函数相对于标量的导数，**y=f**(x)，看看是否能抽象出一个一般的公式。

$$\begin{bmatrix} y_1(x) \\ y_2(x) \end{bmatrix} = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} ln(x^2) \\ sin(3x) \end{bmatrix}$$

让我们引入两个中间变量，g1 和 g2，这样 y 看起来更像是 **y = f(g**(x))。

$$\begin{bmatrix} g_1(x) \\ g_2(x) \end{bmatrix} = \begin{bmatrix} x^2 \\ 3x \end{bmatrix}$$

$$\begin{bmatrix} f_1(\mathbf{g}) \\ f_2(\mathbf{g}) \end{bmatrix} = \begin{bmatrix} ln(g_1) \\ sin(g_2) \end{bmatrix}$$

矢量 y 相对于标量 x 的导数是一个垂直矢量，其元素的计算方法是使用多变量链式法则计算。

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial f_1(\mathbf{g})}{\partial x} \\ \frac{\partial f_2(\mathbf{g})}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2}\frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2}\frac{\partial g_2}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1}2x + 0 \\ 0 + cos(g_2)3 \end{bmatrix} = \begin{bmatrix} \frac{2x}{x^2} \\ 3cos(3x) \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3cos(3x) \end{bmatrix}$$

好了，现在我们只用标量规则就可以得到答案了。让我们试着从这个结果中抽象出它在矢量形式中的样子。我们的目标是将标量运算的向量转换为向量运算。

$$\begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x} \\ \frac{\partial g_2}{\partial x} \end{bmatrix} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial x}$$

这意味着，雅各布矩阵是另外两个雅各布矩阵的矩阵乘法，这有点酷。让我们检查一下我们的结果。

$$\frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial x} = \begin{bmatrix} \frac{1}{g_1} & 0 \\ 0 & cos(g_2) \end{bmatrix} \begin{bmatrix} 2x \\ 3 \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1}2x + 0 \\ 0 + cos(g_2)3 \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3cos(3x) \end{bmatrix}$$

哇！我们得到的答案与标量法相同。这个适用于的向量的链式法则似乎是正确的，事实上，它反映了单变量的链式法规。

为了使这个公式适用于多个参数或矢量 x，我们只需将方程中的 x 改为矢量 **x** 的方程式。其效果是，$\frac{\partial \mathbf{g}}{\partial \mathbf{x}}$ $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ 现在均是雅各布矩阵（$\frac{\partial \mathbf{g}}{\partial \mathbf{x}}$ 不再是列向量了）。我们完整的矢量链规则是：

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \mathbf{x}}$$

矢量公式比单变量链规则的好处是，它自动考虑了总导数，同时保持了同样的符号简单性。

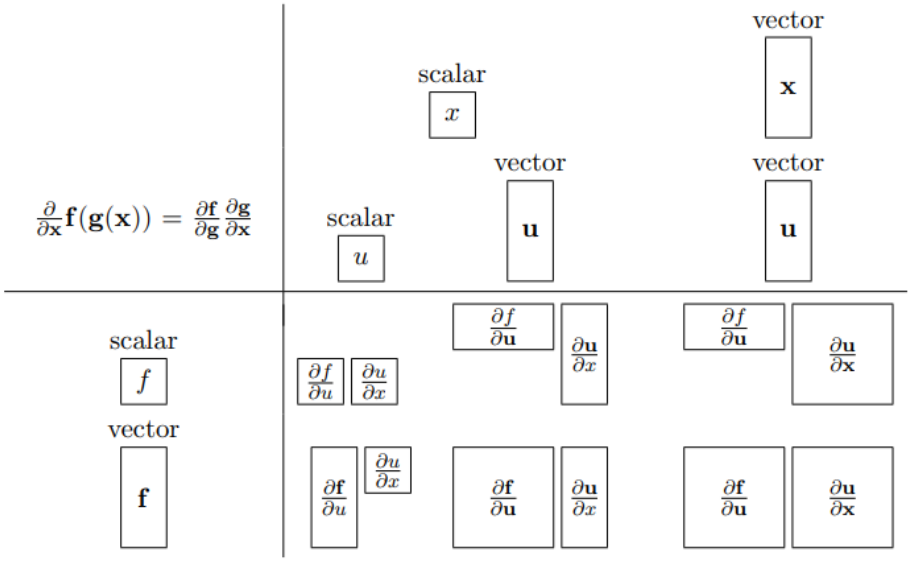
$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x})) = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ & & \cdots & \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ & & \cdots & \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_n} \end{bmatrix}$$

其中 m=|**f**|，n=|**x**|，k=|**g**|。由此产生的雅各布系数为 m×n（m×k 矩阵乘以 k×n 矩阵）。乘以一个 k×n 的矩阵）。在这个公式中，我们还可以进一步简化，因为对于许多应用来说，雅各布矩阵是对角矩阵。与神经网络相关的数学处理的是向量的函数而不是函数的向量。例如，神经元的 **z=w*x+b** 有项 sum(w⊗x)（来自于 **w*x**），其激活函数是 max(0, x)。我们将在下一节中考虑这些函数的导数。

当 f_i 纯粹是 g_i 的函数，而 g_i 纯粹是 x_i 的函数时：在这种情况下，矢量链规则简化为。

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x})) = diag(\frac{\partial f_i}{\partial g_i})diag(\frac{\partial g_i}{\partial x_i}) = diag(\frac{\partial f_i}{\partial g_i}\frac{\partial g_i}{\partial x_i})$$

因此，Jacobian 简化为一个对角矩阵，其元素可由单变量链式法则求出。在经历了所有的数学运算之后，这就是回报。以下是表中总结的为了得到雅各布系数而需要乘以的适当的成分



四、 神经元激活的梯度

我们现在有了计算一个典型的神经元激活导数所需的所有部分，即单个神经网络计算单元相对于模型参数 w 和 b 的导数。activation(**x**) = max(0, **w*x**+b)

让我们稍后再担心激活函数 max 的问题，重点是计算 $\frac{\partial w*x+b}{\partial w}$ $\frac{\partial w*x+b}{\partial b}$。(回

顾一下神经网络通过优化其权重和偏置来学习）。到现在我们还没有讨论点积的导数，**y=f(w)*g(x)**，但我们可以使用链式法则来避免该规则。 点积 **w*x** 只是元素间的相乘之和（译者注：这里并没有讨论矩阵微积分，只有向量微积分），可以写做：

$$\sum_i^n (w_i x_i) = sum(\mathbf{w} \otimes \mathbf{x}).$$

我们知道如何计算 **sum(x)**和 **w⊗x** 的偏导数，但还没有研究 **sum(w⊗x)**的偏导数。我们需要用连锁法则来解决这个问题，因此我们可以引入一个中间向量变量 u，就像我们使用单变量链规则一样。

$$\begin{aligned} \mathbf{u} &= \mathbf{w} \otimes \mathbf{x} \\ y &= sum(\mathbf{u}) \end{aligned}$$

一旦我们重新表述了 y，我们就能认识到两个我们已经知道它们偏导数的子表达式，。

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}}(\mathbf{w} \otimes \mathbf{x}) &= diag(\mathbf{x}) \\ \frac{\partial y}{\partial \mathbf{u}} &= \frac{\partial}{\partial \mathbf{u}} sum(\mathbf{u}) &= \vec{1}^T \end{aligned}$$

让我们来解决神经元激活函数：max（0，**w * x** + b）（Relu）。函数的导数是一个分段函数。当 z≤0 时，导数为 0，因为 z 是一个常数。当 z > 0 时，最大函数的导数只是 z 的导数，也就是 1。

若 **x** 为向量，我们得到其标量导数为：

$$\frac{\partial}{\partial x_i} max(0, x_i) = \begin{cases} 0 & x_i \leq 0 \\ \frac{dx_i}{dx_i} = 1 & x_i > 0 \end{cases}$$

$$\frac{\partial}{\partial \mathbf{x}} max(0, \mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} max(0, x_1) \\ \frac{\partial}{\partial x_2} max(0, x_2) \\ \cdots \\ \frac{\partial}{\partial x_n} max(0, x_n) \end{bmatrix}$$

为了得到激活函数的导数，我们需要使用链式规则，因为嵌套的子表达式，**w * x** + b。让我们引入中间变量 z。

$$z(\mathbf{w}, b, \mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

$$activation(z) = max(0, z)$$

我们可以重写如下。

$$\frac{\partial activation}{\partial \mathbf{w}} = \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ \mathbf{x}^T & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

现在让我们用这些偏导数来处理整个损失函数。

五、 神经网络损失函数的梯度

训练一个神经元需要我们最小化"成本 "函数，因为我们用多个矢量输入和多个标量目标进行训练，所以我们需要对模型的参数 **w** 和 b 进行梯度下降。让：

$$X = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^T$$

其中 N=|X|，然后让

$$\mathbf{y} = [target(\mathbf{x}_1), target(\mathbf{x}_2), \ldots, target(\mathbf{x}_N)]^T$$

其中 y_i 是一个标量。那么成本方程就变成了:

$$C(\mathbf{w}, b, X, \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - activation(\mathbf{x}_i))^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - max(0, \mathbf{w} \cdot \mathbf{x}_i + b))^2$$

按照我们的连锁规则过程，引入了这些中间变量。

$$\begin{aligned} u(\mathbf{w}, b, \mathbf{x}) &= max(0, \mathbf{w} \cdot \mathbf{x} + b) \\ v(y, u) &= y - u \\ C(v) &= \frac{1}{N}\sum_{i=1}^{N} v^2 \end{aligned}$$

让我们先计算一下关于 **w** 的梯度。

5.1 W 的梯度

$$
\begin{aligned}
\frac{\partial C(v)}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} v^2 \\
&= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial \mathbf{w}} v^2 \\
&= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial v^2}{\partial v} \frac{\partial v}{\partial \mathbf{w}} \\
&= \frac{1}{N} \sum_{i=1}^{N} 2v \frac{\partial v}{\partial \mathbf{w}} \\
&= \frac{1}{N} \sum_{i=1}^{N} \begin{cases} 2v\vec{0}^T = \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2v\mathbf{x}^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}
\end{aligned}
$$

$$
\frac{\partial C}{\partial \mathbf{w}} = \frac{2}{N} \sum_{i=1}^{N} e_i \mathbf{x}_i^T \quad \text{(for the nonzero activation case)}
$$

从这里可以看出，这种计算是对 **X** 中所有 x_i 的加权平均。误差项 e_i，即每个 x_i 的目标输出和实际神经元输出之间的差异。如果误差为 0，那么梯度为零，我们就达到了最小损失。如果 e_i 是某个小的正差，那么梯度就是朝 x 方向的一小步。如果 e1 很大，那么梯度就是在这个方向上的一大步。如果 e1 是负的，梯度是负的，意味着最高的成本是在负的方向。当然，我们希望减少而不是增加损失，这就是为什么梯度下降递归关系采取梯度的负值。关系采用梯度的负数来更新当前位置（对于标量学习率 η）。

$$
\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial C}{\partial \mathbf{w}}
$$