第五章、深度学习

赵涵

2022 年 5 月 4 日

我们本章讨论深度学习。在这一章，我们对神经网络一个大致的轮廓，讨论神经网络的向量化与反向传播算法。

最开始，我们回顾一下，神经网络的发展历史：

- 1958，PLA
- 1969，PLA不能解决XOR等非线性数据
- 1981，MLP，多层感知机的出现解决了上面的问题
- 1986，BP算法应用在MLP上，RNN
- 1989，CNN，Univeral Approximation Theorem，但是于此同时，由于深度和宽度的相对效率不知道，并且无法解决BP算法的梯度消失问题
- 1993，1995，SVM+kernel，AdaBoost，RandomForest，这些算法的发展，DL逐渐没落
- 1997，LSTM
- 2006，基于RBM的深度信念信念络和深度自编码
- 2009，GPU的发展
- 2011，在语音识别上的应用
- 2012，ImageNet
- 2013，VAE
- 2014，GAN
- 2016，AlphaGo
- 2018，GNN

DL不是一个新的东西，其近年来的大发展主要原因如下：

1. 数据量变大
2. 分布式计算的发展
3. 硬件算力的发展

对非线性问题，一般有三种方法进行处理：

1. 非线性转换，将低维空间转换到高维空间（Cover 定理），从而变为一个线性问题。
2. 核方法，由于非线性转换是变换为高维空间，因此可能导致维度灾难，并且可能很难得到这个变换函数，核方法不直接寻找这个转换，而是寻找一个内积。
3. 神经网络方法，将复合运算变为基本的线性运算的组合。

1 非线性模型下的监督学习

在之前学习的监督学习部分（从输入x去预测标签y），我们的模型假设是$h_\theta(x)$。在以往的章节中，我们考虑的是$h_\theta(x)=\theta^T x$（线性回归或者线性分类），或者是$h_\theta=\theta^T\phi(x)$（$\phi(x)$是特征图谱）。接下来我们学习一类通用的模型，在输入x与参数θ之间是**非线性**（non-linear）关系。最常见的非线性模型是**神经网络**（neural network）。本章，我们着重讨论神经网络的结构与性质。

假定样本集$\{x^{(i)},y^{(i)}\}_{i=1}^n$，简单起见，我们先处理回归任务，即$y^{(i)}\in\mathbb{R}$和$h_\theta(x)\in\mathbb{R}$。对于回归任务，我们与线性回归类似，对任意一个样本$(x^{(i)},y^{(i)})$定义如下的损失函数：

$$J^{(i)}(\theta)=\frac{1}{2}(h_\theta(x^{(i)})-y^{(i)})^2 \tag{1}$$

那么整个数据集的损失函数为：

$$J(\theta)=\frac{1}{n}\sum_{i=1}^{n}J^{(i)}(\theta) \tag{2}$$

这里与线性回归的不同是，第一，前面的系数不再是1/2，但是对于一个函数求极值来说，整体放大或者缩小，并不影响损失函数局部最优或者全局最优对应的极值点；第二，假设$h_\theta(x)$不再是线性形式了。这里提一下，对于损失函数，文献里有两个名字：cost function或者loss function，都是指代同一个目标函数。

求解目标函数的算法，我们采用批量梯度下降或者随机梯度下降。批量梯度下降的更新规则为：

$$\theta:=\theta-\alpha\nabla_\theta J(\theta) \tag{3}$$

α是学习率，需要人工设定，是一个超参数。对于随机梯度下降，有如下的更新步骤：

Algorithm 1 随机梯度下降

Input: 超参数：学习率α，迭代次数n_{iter}。

Output: 最优目标函数对应的参数θ。

随机初始化θ（一般采用高斯分布进行采样赋值）。

for $i=1$ to n_{iter} **do**

　　从$\{1,...,n\}$里面均匀采样，抽取一个样本j，更新参数θ:

$$\theta:=\theta-\alpha\nabla_\theta J^{(i)}(\theta) \tag{4}$$

现在由于计算机算力的提升，分布式计算与并行计算广泛使用，稍微好一点的个人笔记本电脑一般都带有独立显卡，就是所谓GPU。在GPU上要比在CPU上运算效率高很多倍，原因是CPU的计算的单元是标量，GPU的计算单元是向量。所以同时计算B个样本的梯度，要播分开计算B个样本的梯度要快。因此，小批量梯度下降在深度学习当中，广泛使用。这里我们给出小批量梯度下降的算法步骤：这里我们首先介绍了策略和算法，即损失函数与更新规则。下面我们主要介绍模型的构成。

Algorithm 2 小批量随机梯度下降

Input: 超参数：学习率α，小批量的个数B，迭代次数n_{iter}。

Output: 最优目标函数对应的参数θ。

随机初始化θ（一般采用高斯分布进行采样赋值）。

for $i = 1$ to n_{iter} **do**

　　从$\{1, ..., n\}$里面均匀采样，抽取B个样本集合$j_1, ..., j_B$（不能重复采样），更新参数θ:

$$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta) \tag{5}$$

2 神经网络

神经网络对应着一大类模型，它的假设$h_\theta(x)$设计到矩阵乘法与非线性操作等。我们在这一节，一步一步地建立起神经网络。

我们这一节通过房价预测问题作为例子，即已知房屋的面积，我们预测价格。在之前我们建立的线性回归模型是存在问题的，即很有可能建立起的直线，在面积相对有点小的情况下，预测的房价是负的。所以我们为了解决这个问题，我们要求房子的价格最小是0，不可能是负的。根据这一直觉，我们修改线性模型的假设，定义如下的函数：

$$h_\theta(x) = \max(wx + b, 0), \text{ where } \theta = (w, b) \in \mathbb{R} \tag{6}$$

这里$h_\theta(x)$返回一个实数，要么是线性回归的值，要么是0，取0是因为线性回归预测的值是负数。我们把$\max\{t, 0\}$这种形式的函数，称为ReLU函数（rectified linear unit），所以通常写为：

$$\text{ReLU}(t) \equiv \max\{t, 0\} \tag{7}$$

这个函数的函数图像是（1），图像很清楚的可以看到，这是一个非线性函数。在之前介绍Logistic回归时，我们提到过激活函数，在这我们可以认为，只要把线性关系，映射到非线性，都可以称为**激活函数**（activation function）。显然，ReLU函数就是激活函数的一种（之后我们会把非线性激活称为神经元）。 当输入的特征是一个向量$x \in \mathbb{R}^d$时，那么一个神经元可以写成：

$$h_\theta(x) = ReLU(w^T x + b), \text{ where } w \in \mathbb{R}^d, b \in \mathbb{R}, \text{ and } \theta = (w, b) \tag{8}$$

其中b我们把它称作**偏置**（bias），向量w称为**权重向量**（weight vector）。这样的神经元是构成神经网络的基本单元。下面我们进行神经元的堆叠。

如果我们把单个神经元的输出，当成另一个神经元的输入，还可以把单个神经元的输出，输入到多个神经元，就可以构成一个更复杂的结构。我们把这种方式，称为**堆叠**（stack）。那么搭建一个神经网络就像积木一样，把每个神经元当成搭砖块一样，一个一个拼接在一起。让我们继续深入考虑房价预测的问题。除了房子的面积，假设我们还知道房子的卧室数量，邮编，还有地段。在给出这些特征后，我们会认为房子的价格依赖于所能容纳的最大人数。假定容纳的家庭人数是房子面积和卧室数目的函数。邮编可能会提供额外的信息，比如哪个区（以北京市为例，海淀区的房价要比昌平区的房价高很多，因为学校众多，教育资源，医疗资源都很丰富）。结合邮编和地段，那就很容易预测出房子就近具有哪些学校。给出三个特征（容纳人数，地区，学校质量），我们综合起来，就能最终对房子给出一个合理的价格，参看图（2）。 从形式来说，神经网络的输入为x_1, x_2, x_3, x_4。我们把容纳人数，地区，学校质量，用a_1, a_2, a_3（这里的a_i经常称为**隐层神经元**hidden neurons）。我们把每个a_i当成$x_1, ..., x_4$的

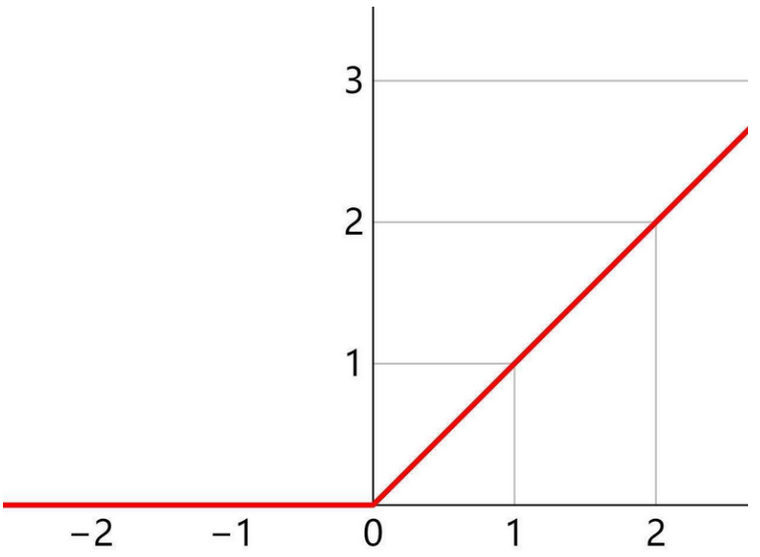

Figure 1: ReLU函数的非线性特征。

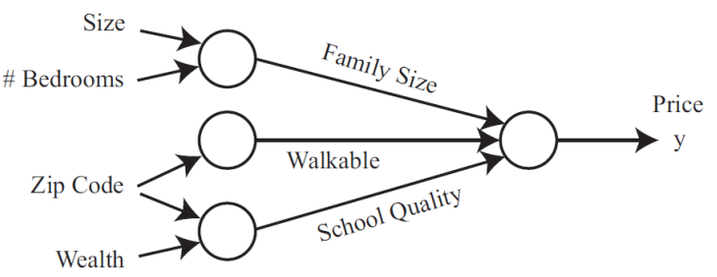

Figure 2: 对房价预测问题，组件的一个小型神经网络。

输出。用公式表达：

$$a_1 = ReLU(\theta_1 x_1 + \theta_2 x_2 + \theta_3) \tag{9}$$

$$a_2 = ReLU(\theta_4 x_3 + \theta_5) \tag{10}$$

$$a_3 = ReLU(\theta_6 x_3 + \theta_7 x_4 + \theta_8) \tag{11}$$

这里$(\theta_1, ..., \theta_8)$是模型参数。最后房价的输出$h_\theta(x)$，是前面$a_1, a_2, a_3$的线性组合，我们有：

$$h_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12} \tag{12}$$

模型参数，再加上上面的4个，即$\theta_1, ..., \theta_{12}$，一共12个。现在我们就用一个相当复杂的函数作为模型的输出。可以通过梯度下降去学习参数θ。

通名字来看，可以看出，搭建出来的人工神经网络的灵感来自于生物学上的神经网络，即大脑内部的结构。隐层神经元$a_1, ..., a_n$对应着生物系统的神经元，参数θ_i对应着神经元的突触。然而，我们搭建的人工神经网络与大脑真正的神经网络相比有多少相似性，我们无从得知。举一个例子，大脑的学习是梯度下降吗？大脑里的神经元是不同的，有视觉神经元，运动神经元，多巴胺神经元等等，但是我们做的人工神经网络，全都是相同的。大脑与智能目前是21世纪最热门的科研问题，并且也是相当困难的问题。

话题回到人工神经网络，在之前的案例当中，我们使用了一些先验知识，即房子的地段，容纳人数等。但是对于其他的问题，我们可能并不熟悉或者了解内部的细节，但是由之前问题搭建出的神经网络，依旧可以作为一个模型来进行使用，即脱离之前的例子，抽象出一个通用模型，即：

$$a_1 = ReLU(w_1^T x + b_1), \text{ where } w_1 \in \mathbb{R}^4 \text{ and } b_1 \in \mathbb{R} \tag{13}$$

$$a_2 = ReLU(w_2^T x + b_2), \text{ where } w_2 \in \mathbb{R}^4 \text{ and } b_2 \in \mathbb{R} \tag{14}$$

$$a_3 = ReLU(w_3^T x + b_3), \text{ where } w_3 \in \mathbb{R}^4 \text{ and } b_3 \in \mathbb{R} \tag{15}$$

我们仍然利用公式（12）作为最后的输出。我们把这种模型称为**全连接神经网络**（fully-connected neural network），因为通过可视化结构（3），我们可以看出，上一层与下一层之间，全都紧密的连接在一起。 不失一般性，我们把两层全连接神经网络，假定输入有d 维，那么两层神经网络的通式为：

$$\forall j \in [1, ..., m],\ z_j = {w_j^{[1]}}^T x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \tag{16}$$

$$a_j = ReLU(z_j) \tag{17}$$

$$a = [a_1, ..., a_m]^T \in \mathbb{R}^m \tag{18}$$

$$h_\theta(x) = {w^{[2]}}^T a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R} \tag{19}$$

这里的$\mathbb{R}^d$为列向量。中括号[1]和[2]代表着层数。权重w_j的角标，代表该层第几个神经元的权重。。以上我们讨论了下面我们讨论如何对一层的神经元进行向量化。

在我们介绍更多层和更复杂的神经网络结构时，我们这里先把神经网络的数学表达向量化。之所以要向量化，是为了加速计算（之前我们提到过，GPU能更好地处理向量运算）。以往地训练规则，我们知道都是循环地嵌套，因为循环会让计算变慢，向量化地目的就是减少循环。所谓地**向量化**（vectorization），就是把每个神经元的计算堆叠在一起，从向量运算，转变成矩阵运算。在这里，我们应该强调，目前的深度学习库，其中Python下的pytorch和Tensorflow都可以非常简单快捷的搭建神经网络，但是对于一个初学者，还是需要详细了解神经网络的运算机制。

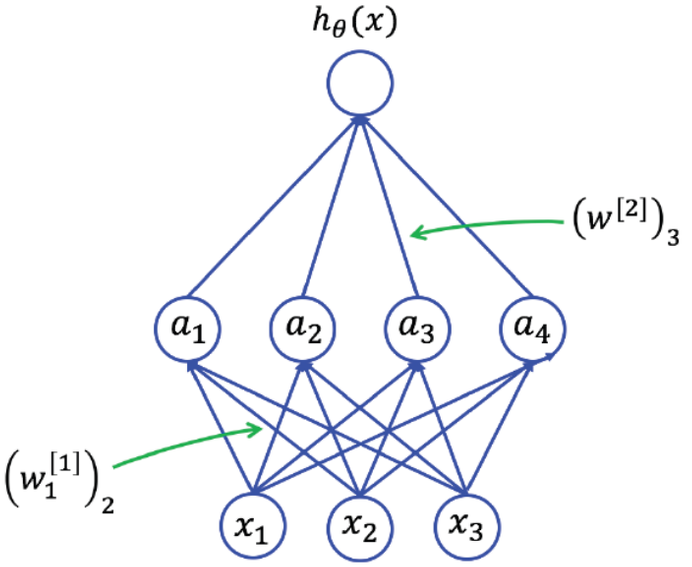

Figure 3: 两层全连接神经网络。每条边从x_i指向a_j，表示a_j的值依赖于x_i。每条边都对应着一个权重$(w_j^{[1]})_i$，记号i-th表示上一层的神经元编号，j-th表示下一层的神经元编号，中括号[1] 表示是第几层对应的权重值。a_j的值，由上一层所有的神经元输出的值线性叠加并激活得到，即：$a_j = ReLU(\sum_{i=1}^{d}(w_j^{[1]})_i x_i$。

首先我们定义一个矩阵$W^{[1]} \in \mathbb{R}^{m\times d}$把所有第一层的神经元的权重进行拼接，即：

$$W^{[1]} = \begin{bmatrix} - & w_1^{[1]^T} & - \\ - & w_2^{[1]^T} & - \\ & \vdots & \\ - & w_m^{[1]^T} & - \end{bmatrix} \in \mathbb{R}^{m\times d} \tag{20}$$

这个矩阵称为参数矩阵，我们可以继续写出第一层神经元的输出$z = [z_1, \ldots, z_m]^T \in \mathbb{R}$的计算公式：

$$\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} - & w_1^{[1]^T} & - \\ - & w_2^{[1]^T} & - \\ & \vdots & \\ - & w_m^{[1]^T} & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix} \tag{21}$$

对上式进行维度检查，$z \in \mathbb{R}^{m\times 1}, W^{[1]} \in \mathbb{R}^{m\times d}, x \in \mathbb{R}^{d\times 1}, b^{[1]} \in \mathbb{R}^{m\times 1}$。写成矩阵的形式：

$$z = W^{[1]}x + b^{[1]} \tag{22}$$

上式就是把单个神经元整合在一起的矩阵表达式。同样，对于激活这一操作，不仅可以在标量上进行运算，也可以以推广到向量或者矩阵，即ReLU函数，对z进行操作，得到a，即：

$$a = ReLU(z) \tag{23}$$

第二层类似，同样定义$W^{[2]} = [w^{[2]^T}] \in \mathbb{R}^{1\times m}$。然后参考第一层的写法，第二层和输出层有如下的表达

式：

$$a = ReLU(W^{[1]}x + b^{[1]}) \tag{24}$$
$$h_\theta(x) = W^{[2]}a + b^{[2]} \tag{25}$$

模型的参数是$W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}$。对于两层的神经网络，有时也成为一隐藏层的神经网络（one-hidden-layer neural network）。

在讨论完两层的神经网络后，那么我们就可以推广到多层神经网络，同样采用向量化的表达式，假设一共有r层，那么我们有：

$$a^{[1]} = ReLU(W^{[1]}x + b^{[1]}) \tag{26}$$
$$a^{[2]} = ReLU(W^2a^{[1]} + b^{[2]}) \tag{27}$$
$$\cdots \tag{28}$$
$$a^{[r-1]} = ReLU(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \tag{29}$$
$$h_\theta(x) = W^{[r]}a^{[r-1]} + b^{[r]} \tag{30}$$

这里我们需要知道，每一层的搭建，都需要进行维度检查，即假设$a^{[k]}$的维度是m_k，那么它对应的权重矩阵$W^{[k]} \in \mathbb{R}^{m_k \times m_{k-1}}$，对应的偏置项$b^{[k]} \in \mathbb{R}^{m_k}$，对于第一层来说，$W^{[1]} \in \mathbb{R}^{m_1 \times d}$，对于最后一层，我们有$W^{[r]} \in \mathbb{R}^{1 \times m_{r-1}}$。那么对于一个搭建好的神经网络，一共有$m_1 + m_2 + \cdots + m_r$个神经元，一共具有$(d+1)m_1 + (m_1+1)m_2 + \cdots + (m_{r-1}+1)m_r$个参数。对于输入层，我们一般记为第零层，即$a^{[0]} = x$，然后最后一层是$a^{[r]} = h_\theta(x)$。最后，写出通式：

$$a^{[k]} = ReLU(W^{[k]}a^{[k-1]} + b^{[k]}), \forall k = 1, \ldots, r-1 \tag{31}$$

最后一层，我们根据任务的不同，一般会选择不同的激活函数，二分类问题选择Sigmoid函数激活，多分类函数选择softmax激活，回归任务一般不使用激活函数，直接输出。

对于各层之间的激活，我们上面介绍了ReLU函数，还有一些比较常用的函数，我们这里简单介绍一下：

$$\sigma(z) = \frac{1}{1+e^{-z}}(\text{sigmoid}) \tag{32}$$
$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}(\tanh) \tag{33}$$
$$\sigma(z) = \max(\alpha z, z)(\text{Leaky ReLU}, \alpha \in \mathbb{R}) \tag{34}$$

我们解释一下，为什么要使用激活函数，而不是直接采取$\sigma(z) = z$，可以先假定偏置都是0。我们进行如下的计算：

$$h_\theta(x) = W^{[2]}a^{[1]} \tag{35}$$
$$= W^{[2]}\sigma(z^{[1]}) \text{ by definition} \tag{36}$$
$$= W^{[2]}z^{[1]} \text{ since } \sigma(z) = z \tag{37}$$
$$= W^{[2]}W^{[1]}x \text{ from Equation(21)} \tag{38}$$
$$= \tilde{W}x \text{ where } \tilde{W} = W^{[2]}W^{[1]} \tag{39}$$

最后一个等式把两个系数吸收为一个因子$\tilde{W}$。我们可以看到，如果没有激活函数，无论构造多少层，最终都等效为1层。换句话说，激活函数承担着非线性变换的责任，一旦没有了非线性，神经网络立刻就退化为线性回归模型。

我们最后在讨论一下神经网络与核方法之间的联系。在之前的章节中，我们介绍了特征图谱的概念。主要的思想也是要把属性x通过核函数，做非线性变换，然后在进行回归或者分类，即$\theta^T\phi(x)$，这里$\phi(x)$就是核函数映射后的特征图谱，对于一个学习模型来说，是非常依赖特征图谱$\phi(x)$的选择的，一般情况下，人们会利用一些先验知识进行构造合适的特征图谱。这种人为地构造特征图谱的过程，称为**特征工程**（feature engineering）。

在了解了深度学习，我们就可以通过深度学习自动地得出正确的特征图谱。在全连接神经网络的倒数第二层，当作特征图谱，从第一层到倒数第二层，所有的参数，可以认为是β，换句话说，我们可以认为倒数第二层的输出$a^{[r-1]}$，对应着特征图谱，即$a^{[r-1]}=\phi_\beta(x)$。现在我们可以写出最后一层的输出：

$$h_\theta(x)=W^{[r]}\phi_\beta(x)+b^{[r]} \tag{40}$$

当训练过程结束，即β固定，那么$\phi_\beta(\cdot)$就可以认为是完整的特征图谱，那么最后一层就可以当作线性模型了。但是训练神经网络的过程，一定涉及到$W^{[r]}$与$b^{[r]}$，所以在学习的过程，不仅在学习特征图谱的参数，而且还在学习线性模型的参数。因此，深度学习不在依赖人类的先验知识，也不再需要手动的去建立特征图谱，倒数第二层的输出尝尝被当作特征图谱，在各类任务当中，大面积使用（迁移学习的核心，就是对神经网络最后几层进行更换，使用前面得到的特征图谱）。

再次回到之前讲的案例，房价预测，一个全连接神经网络，完全可以脱离我们具体出来的一些特征，比如容纳人数，地段等等，它会自动地发现一些有用的特征，还有一个特性，就是从一个数据集学到的神经网络，换一个类似的数据集，往往得到的特征图谱还是管用的。只不过我们并不知道，这些特征的具体含义。所以为什么很多人把神经网络称为**黑箱**（black box），因为我们并不知道它的输出代表着什么含义。

3 反向传播算法

这一节，我们主要讨论反向传播算法，或者叫自动微分，目的就是在做梯度下降的时候，计算损失函数的梯度$\nabla_\theta J^{(i)}(\theta)$。关于计算梯度这个事，在CS229课程讲义上，给出了一个定理：

定理：假定一个可导函数是由N次计算得到的一个映射，即$f:\mathbb{R}^l\to\mathbb{R}$，那么计算这个函数的梯度，与计算该函数的函数值，时间复杂度是一样的，都是$O(N)$。

这个定理传达出一个信息，就是只要是可导的函数，梯度的计算，并不会因为函数的复杂，使得梯度计算的时间复杂度成指数，或者多项式增加，而仅仅是线性关系。这就为训练一个特别大的神经网络提供了理论基础。在Tensorflow和Pytorch深度学习库中，已经不再需要知道到底事如何进行训练模型了，但是作为初学者，还是需要知道其中的训练规则与细节。

3.1 链式求导法则

回顾微积分里面的链式求导法则。假定目标函数J依赖参数$\theta_1,\ldots,\theta_p$，其中中间变量为$g_1,\ldots,g_k$，表达式如下：

$$g_j=g_j(\theta_1,\ldots,\theta_p),\forall j\in\{1,\ldots,k\} \tag{41}$$

目标函数为：

$$J=J(g_1,\ldots,g_k) \tag{42}$$

中间变量g_i承担着复合函数的嵌套作用。那么链式求导法则为：

$$\frac{\partial J}{\partial\theta_i}=\sum_{j=1}^{k}\frac{\partial J}{\partial g_j}\frac{\partial g_j}{\partial\theta_i} \tag{43}$$

这种单层嵌套的复合函数，在神经网络里，我们把J称为输出变量，$g_1,\ldots,g_k$称为中间变量，$\theta_1,\ldots,\theta_p$称为输入变量。

3.2 两层神经网络的反向传播

我们先看两层的神经网络如何求导。为了简洁，我们先从标量求导出发，然后推广到矩阵形式。

我们先看一下两层神经网络对应的损失函数方程组：

$$
\begin{aligned}
z &= W^{[1]}x + b^{[1]} && (44)\\
a &= ReLU(z) && (45)\\
h_\theta(x) \equiv o &= W^{[2]}a + b^{[2]} && (46)\\
J &= \frac{1}{2}(y-o)^2 && (47)
\end{aligned}
$$

当输入的向量$x \in \mathbb{R}^d$时，参数矩阵的维度分别为：$W^{[1]} \in \mathbb{R}^{m\times d}, W^{[2]} \in \mathbb{R}^{1\times m}$和$b^{[1]}, z, a \in \mathbb{R}^m$和$o, y, b^{[2]} \in \mathbb{R}$。对于向量的形式，依旧按照之前的约定，默认为列向量处理。[1]

我们先计算$\frac{\partial J}{\partial W^{[2]}}$。对把参数矩阵写成行向量的形式，$W^{[2]} = [W_1^{[2]},\ldots,W_m^{[2]}]$。对其中一个元素$W_i^{[2]}$使用链式求导法则，有：

$$
\begin{aligned}
\frac{\partial J}{\partial W_i^{[2]}} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial W_i^{[2]}} && (48)\\
&= (o-y)\frac{\partial o}{\partial W_i^{[2]}} && (49)\\
&= (o-y)a_i \text{ because } o = \sum_{i=1}^{m} W_i^{[2]}a_i + b^{[2]} && (50)
\end{aligned}
$$

上面是对一个元素求导，使用向量的记号，写成更为紧凑的形式，有：

$$
\frac{\partial J}{\partial W^{[2]}} = (o-y)a^T \in \mathbb{R}^{1\times m} \tag{51}
$$

相似地，对偏置进行求导，有：

$$
\frac{\partial J}{\partial b^{[2]}} = (o-y) \in \mathbb{R} \tag{52}
$$

在第零章中，我们讨论过标量函数对向量求导，基于此，我们用$\frac{\partial J}{\partial A}$用来表示，损失函数$J$对参数矩阵$A$地求导，结果是一个与$A$维度相同的矩阵，矩阵中地每个元素为$\frac{\partial J}{\partial A_{ij}}$，角标$ij$表示第$i$行第$j$列。

接下来计算$\frac{\partial J}{\partial W^{[1]}}$，同样我们还是从一个元素出发，用$W_{ij}^{[1]}$表示参数矩阵$W$中的一个元素，其中$i \in [m], j \in [d]$，我们使用链式法则有：

$$
\begin{aligned}
\frac{\partial J}{\partial W_{ij}^{[1]}} &= \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial W_{ij}^{[1]}} && (53)\\
&= \frac{\partial J}{\partial z_i}x_j(\text{ because } z_i = \sum_{k=1}^{d} W_{ik}^{[1]}x_k + b_i^{[1]}) && (54)
\end{aligned}
$$

上式是标量形式，向量化后的紧凑形式为：

$$
\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z}x^T \tag{55}
$$

[1]在numpy里，数组与向量有一定的区别，之间相互转化，需要进行类型转换，这一点需要格外注意。

这里每个量的维度分别为：$\frac{\partial J}{\partial W^{[1]}} \in \mathbb{R}^{m\times d}, \frac{\partial J}{\partial z} \in \mathbb{R}^{m\times 1}, x^T \in \mathbb{R}^{1\times d}$。对参数矩阵的求导过程中，需要计算中间变量$z$的导数，所以接下来，我们继续计算$\frac{\partial J}{\partial z}$，同样依照链式求导法则，我们有：

$$\begin{aligned}\frac{\partial J}{\partial z_i} &= \frac{\partial J}{\partial a_i}\frac{a_i}{z_i} &(56)\\ &= \frac{\partial J}{\partial a_i}\mathbb{I}\{z_i \geq 0\} &(57)\end{aligned}$$

这样，我们就把中间涉及到的导数，计算得到了。我们简单起见，把对激活函数的求导，采用更为简洁的记号。先假定激活函数为$a = \sigma(z)$，目标函数$J = J(a)$，那么我们采用向量形式表达链式求导：

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z) \tag{58}$$

这里$\sigma'(\cdot)$代表对激活函数求导，$\odot$代表元素积，所谓的元素积表示对应元素相乘，结果不改变维度，这是与矩阵相乘不同的，是属于两种不同的运算方式。

最后我们需要计算$\frac{\partial J}{\partial a}$。这是上面求导时，出现的中间变量。我们对其中一个元素使用链式求导法则，有：

$$\begin{aligned}\frac{\partial J}{\partial a_i} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_i} &(59)\\ &= (o-y)W_i^{[2]}(\text{ where } o = \sum_{i=1}^{m} W_i^{[2]}a_i + b^{[2]}) &(60)\end{aligned}$$

通过向量化，写成如下紧凑的形式：

$$\frac{\partial J}{\partial a} = {W^{[2]}}^T(o-y) \tag{61}$$

以上，我们就把对参数矩阵求导中，涉及到的所有中间导数，都计算清楚了，根据梯度下降算法，从最后一层，逐步向后传播梯度，我们把这种学习参数的形式，称为**反向传播算法**（Back-propagation algorithm）。我们就可以写出如下的算法：

Algorithm 3 两层神经网络的梯度计算顺序

1: 计算中间变量$z \in \mathbb{R}^m, a \in \mathbb{R}^m, o \in \mathbb{R}$。

2:

3: 计算：

$$\begin{aligned}\delta^{[2]} &\equiv \frac{\partial J}{\partial o} = (o-y) \in \mathbb{R} &(62)\\ \delta^{[1]} &\equiv \frac{\partial J}{\partial z} = ({W^{[2]}}^T(o-y)) \odot \mathbb{I}\{z \geq 0\} \in \mathbb{R}^{m\times 1} &(63)\end{aligned}$$

4: 计算：

$$\begin{aligned}\frac{\partial J}{\partial W^{[2]}} &= \delta^{[2]}a^T \in \mathbb{R}^{1\times m} &(64)\\ \frac{\partial J}{\partial b^{[2]}} &= \delta^{[2]} \in \mathbb{R} &(65)\\ \frac{\partial J}{\partial W^{[1]}} &= \delta^{[1]}x^T \in \mathbb{R}^{m\times d} &(66)\\ \frac{\partial J}{\partial b^{[1]}} &= \delta^{[1]} \in \mathbb{R}^m &(67)\end{aligned}$$

3.3 多层神经网络

这一节，我们从两层神经网络的反向传播出发，推广到多层神经网络。我们首先回顾一下多层神经网络对应的前向传播，参看方程组（26）（27）（28）（29）（30），在添加上最后一层的输出和损失函数：

$$a^{[r]} = z^{[r]} = W^{[r]}a^{r-1} + b^{[r]} \tag{68}$$

$$J = \frac{1}{2}(a^{[r]} - y)^2 \tag{69}$$

根据链式求导法则，中间层的参数，求导公式为：

$$\frac{\partial J}{\partial W^{[k]}} = \frac{\partial J}{\partial z^{[k]}} a^{[k-1]^T} \tag{70}$$

$$\frac{\partial J}{\partial b^{[k]}} = \frac{\partial J}{\partial z^{[k]}} \tag{71}$$

因此，下面需要计算$\frac{\partial J}{\partial z^{[k]}}$，方便起见，定义$\delta^{[k]} \equiv \frac{\partial J}{\partial z^{[k]}}$。我们分别地计算$k = r, \ldots, 1$，最后一层因为不存在激活函数，所以直接有：

$$\delta^{[k]} \equiv \frac{\partial J}{\partial z^{[k]}} = (z^{[r]} - y) \tag{72}$$

而对于其他层，还需要对激活函数求导，我们有：

$$\delta^{[k]} \equiv \frac{\partial J}{\partial z^{[k]}} = \frac{\partial J}{\partial a^{[k]}} \odot ReLU'(z^{[k]}) \tag{73}$$

这里的k取值为$k \leq r-1$。$a^{[k]}$与$z^{[k+1]}$的关系，可以写作：

$$z^{[k+1]} = W^{[k+1]}a^{[k]} + b^{[k+1]} \tag{74}$$

$$J = J(z^{[k+1]}) \tag{75}$$

由求导关系得：

$$\frac{\partial J}{\partial a^{[k]}} = W^{[k+1]^T} \frac{\partial J}{\partial z^{[k+1]}} \tag{76}$$

把上式得到得结果，代入到$\delta^{[k]}$中：

$$\delta^{[k]} = (W^{[k+1]^T} \frac{\partial J}{\partial z^{[k+1]}}) \odot ReLU'(z^{[k]}) \tag{77}$$

$$= W^{[k+1]^T} \delta^{[k+1]} \odot ReLU'(z^{[k]}) \tag{78}$$

至此我们把多层神经网络的梯度都计算得到，下面给出梯度反向传播算法：

4 多样本训练模型的向量化

以上我们都是在一个训练样本上进行讨论的，但是我们说过，我们通过小批量训练加速训练，这里我们假设一次想同时训练3个样本：$x^{(1)}, x^{(2)}, x^{(3)}$。那么第一层神经网络，分别计算三个样本的输出，有：

$$z^{[1](1)} = W^{[1]}x^{(1)} + b^{[1]} \tag{83}$$

$$z^{[1](2)} = W^{[1]}x^{(2)} + b^{[1]} \tag{84}$$

$$z^{[1](3)} = W^{[1]}x^{(3)} + b^{[1]} \tag{85}$$

Algorithm 4 多层神经网络的梯度计算顺序

1: 计算并存储中间变量：从$k=1,\ldots,r$，计算$a^{[k]},z^{[k]}$。把这种计算方式，称为前向传播。

2: 计算梯度：

3: **for** $k=r$ to 1 **do**

4: **if** $k=r$ **then**

$$\delta^{[r]} \equiv \frac{\partial J}{\partial z^{[r]}} \tag{79}$$

5: **else**

6: 计算：

$$\delta^{[k]} \equiv \frac{\partial J}{\partial z^{[k]}} = \left(W^{[k+1]^T}\delta^{[k+1]}\right) \odot ReLU'(z^{[k]}) \tag{80}$$

7: 计算：

$$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]}a^{[k-1]^T} \tag{81}$$

$$\frac{\partial J}{\partial b^{[k]}} = \delta^{[k]} \tag{82}$$

我们注意到记号的不同，方括号代表的是层数，小括号代表的是样本的编号。如果是一个一个训练，那么很显然需要计算三次，现在我们把样本堆叠在一起：

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{d\times 3} \tag{86}$$

这里我们需要提醒一下，堆叠的方式还有另一种，还可以横向堆叠，无论怎么堆叠，都可以得到多样本训练的公式。同样输出的向量z也可以按照列向量堆叠，即：

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]} \tag{87}$$

上面这个公式有一个问题，就是这里$b^{[1]} \in \mathbb{R}^{4\times 1}$和一个维度不匹配的$W^{[1]}X \in \mathbb{R}^{4\times 3}$相加，从矩阵的角度出发，是无法相加的。但在Python中，存在一种维度不匹配的相加方式，称为**广播**（broadcasting）。它的方法是创造一个中间变量$\tilde{b}^{[1]} \in \mathbb{R}^{4\times 3}$：

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{88}$$

这样就可以正常相加了：$Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$，这就是广播的过程。即复制出来多列，等价于每列都加相同的向量。

不同的深度学习包，对应着不同的堆叠方式，如果是横向堆叠的话，对应的参数矩阵就是纵向堆叠。换句话说，要实现线性叠加的结果，样本和参数的堆叠方式，要相反，对于横向堆叠方式，我们就不在详细的推导，直接给出结果：

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in \mathbb{R}^{3\times m} \tag{89}$$

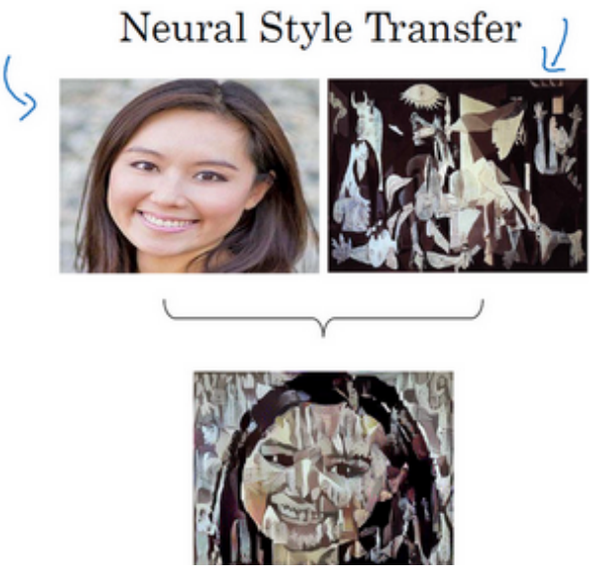

Figure 4: 神经网络实现不同风格的图片叠加。

这里不同的堆叠形式，很容易混乱，过多地讨论这些形式化地东西，意义不大。不管采用何种方式，最后都得落实到程序的实现中，所以这一部分，只要求在Python当中，了解和简单使用Pytorch，搭建简单的神经网络，并能训练出一个全连接神经网络模型，深度学习计算包，会自动地帮你避开这些问题，省去不少的麻烦。

5 卷积神经网络简介

计算机视觉是一个飞速发展的一个领域，这多亏了深度学习。这一部分，我们摘自吴恩达的深度学习课程的第四章的其中一部分，如果想了解更多有关计算机视觉的内容，可以去系统地学习对应的课程。

深度学习与计算机视觉可以帮助汽车，查明周围的行人和汽车，并帮助汽车避开它们。还使得人脸识别技术变得更加效率和精准，你们即将能够体验到或早已体验过仅仅通过刷脸就能解锁手机或者门锁。当你解锁了手机，我猜手机上一定有很多分享图片的应用。在上面，你能看到美食，酒店或美丽风景的图片。有些公司在这些应用上使用了深度学习技术来向你展示最为生动美丽以及与你最为相关的图片。机器学习甚至还催生了新的艺术类型。深度学习之所以让我兴奋有下面两个原因，我想你们也是这么想的。

第一，计算机视觉的高速发展标志着新型应用产生的可能，这是几年前，人们所不敢想象的。通过学习使用这些工具，你也许能够创造出新的产品和应用。

其次，即使到头来你未能在计算机视觉上有所建树，但我发现，人们对于计算机视觉的研究是如此富有想象力和创造力，由此衍生出新的神经网络结构与算法，这实际上启发人们去创造出计算机视觉与其他领域的交叉成果。即使你在计算机视觉方面没有做出成果，我也希望你也可以将所学的知识应用到其他算法和结构。

与计算机视觉相关地任务：图片分类，目标检测，风格迁移等等。所谓的风格迁移，具有如下的形式：

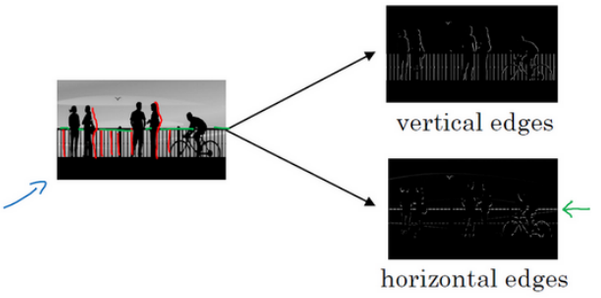

Figure 5: 边缘检测。

5.1 卷积操作

作为卷积神经网络的简介，我们着重介绍卷积操作，目的是帮助各位能在遇到相关的词汇不感到陌生。主要内容包括，图片的构成，边缘检测，过滤器等一些内容。

图片是非结构化数据，一张图片，在计算机的存储方式为一个张量，它的维度为$m \times n \times 3$，m, n代表着图片的长与宽，3代表着信道，所谓信道就是通常所说的三基色：红绿蓝（RGB）。每一个信道，对应着一个$m \times n$的矩阵，矩阵元是一个$0 \sim 255$之间的整数，表示在该信道下，具备色彩的程度。一种特殊的图片，是灰度图，即我们在医院常见的X光片，只有黑白两色，他在计算机当中，是一个矩阵，后面的3应改为1，取值的大小代表着白黑之间的程度，数字越大，对应的点越黑，一般情况下，把这个点称为像素。

我们首先从边缘检测出发，所谓的边缘检测如下图所示： 看一个例子，这是一个6×6的灰度图像。因为是灰度图像，所以它是6×6×1的矩阵，而不是6×6×3的，因为没有RGB三通道。为了检测图像中的垂直边缘，你可以构造一个3×3矩阵。在共用习惯中，在卷积神经网络的术语中，它被称为**过滤器**（filter）。构造一个3×3的过滤器，具有如下的形式：

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{90}$$

。在论文它有时候会被称为**核**（kernel），而不是过滤器，但在这里，我将使用过滤器这个术语。对这个6×6的图像进行卷积运算，卷积运算用“*”来表示，用3×3 的过滤器对其进行卷积。在数学中“*”就是卷积的标准标志，但是在Python中，这个标识常常被用来表示乘法或者元素乘法。所以这个“*”有多层含义，它是一个重载符号。作为例子，图6展示了一个竖直边缘检测操作； 这个卷积运算的输出将会是一个4×4的矩阵，你可以将它看成一个4×4的图像。下面来说明是如何计算得到这个4×4矩阵的。为了计算第一个元素，在4×4左上角的那个元素，使用3×3的过滤器，将其覆盖在输入图像，如图7所示。然后进行元素乘法得到最左上角的元素，即3+1+2+0+0+0+(-1)+(-8)+(-2)=-5。 把这9个数加起来得到-5，当然，你可以把这9个数按任何顺序相加。接下来，为了弄明白第二个元素是什么，你要把蓝色的方块，向右移动一步，像这样，把这些绿色的标记去掉,如图8： 重复进行元素乘法，然后加起来。通过这样做得到-10。再将其右移得到-2，接着是2，3。以此类推，这样计算完矩阵中的其他元素，如图9。 为什么这个可以做垂直边缘检测呢？让我们来看另外一个例子。为了讲清楚，用一个简单的例子。这是一个简单的6×6图像，左边的一半是10，右边一般是0。如果你把它当成一个图片，左边那部分看起来是白色的，像素值10是比较亮的像素值，右边像素值比较暗，我使用灰色来表示0，尽管它也可以被画成黑的。图片里，有一个特别明显的垂直边缘在图像中间，这条垂直线是从黑到白的过渡线，或者从白色到深色。所以，当你用一个3×3过滤器进行卷积运算的时候，这个3×3的过滤器可视化为下面这个样子，在左边有明亮的像素，然后有一个过渡，0在中

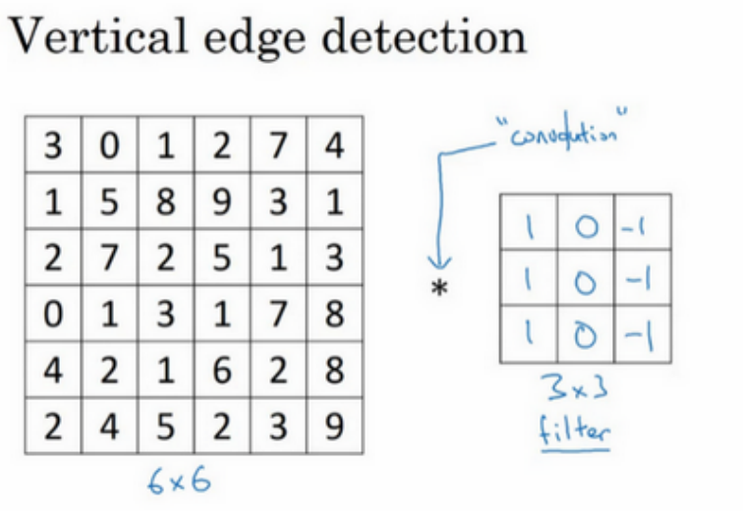

Figure 6: 竖直边缘检测。

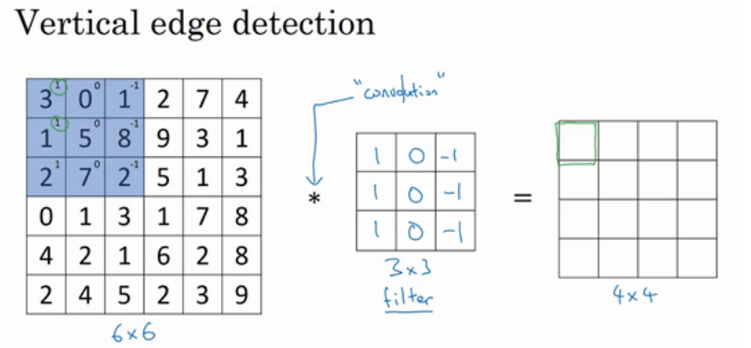

Figure 7: 卷积计算规则。

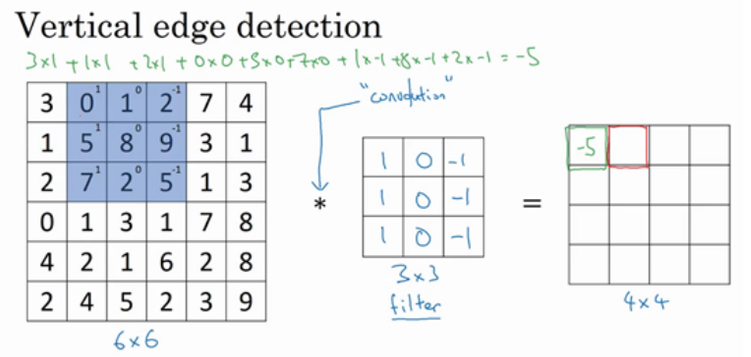

Figure 8: 卷积计算规则。

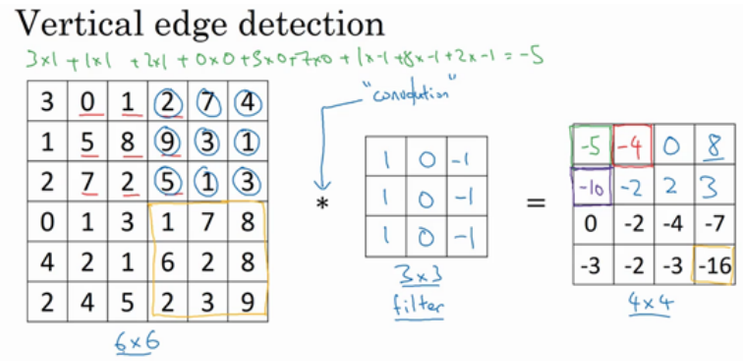

Figure 9: 卷积计算规则。

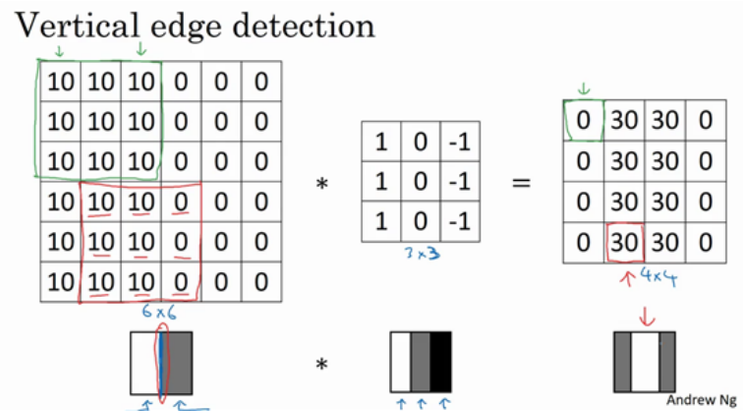

Figure 10: 竖直边缘检测示例。

间，然后右边是深色的。卷积运算后，你得到的是右边的矩阵。如果你愿意，可以通过数学运算去验证。举例来说，最左上角的元素0，就是由这个3×3块（绿色方框标记）经过元素乘积运算再求和得到的，10×1+10×1+10×1+10×0+10×0+10×0+10×(-1)+10×(-1)+10×(-1)=0。相反这个30是由这个（红色方框标记）得到的，10×1+10×1+10×1+10×0+10×0+10×0+0×(-1)+0×(-1)+0×(-1)=30。如果把最右边的矩阵当成图像，它是这个样子。在中间有段亮一点的区域，对应检查到这个6×6图像中间的垂直边缘。这里的维数似乎有点不正确，检测到的边缘太粗了。因为在这个例子中，图片太小了。如果你用一个1000×1000的图像，而不是6×6的图片，你会发现其会很好地检测出图像中的垂直边缘。在这个例子中，在输出图像中间的亮处，表示在图像中间有一个特别明显的垂直边缘。从垂直边缘检测中可以得到的启发是，因为我们使用3×3的矩阵（过滤器），所以垂直边缘是一个3×3的区域，左边是明亮的像素，中间的并不需要考虑，右边是深色像素。在这个6×6图像的中间部分，明亮的像素在左边，深色的像素在右边，就被视为一个垂直边缘，卷积运算提供了一个方便的方法来发现图像中的垂直边缘。如图10。 我们通过上面的例子介绍了卷积操作，对于一个彩色图片，只不过把过滤器也取成张量的形式，相乘后，最后相加，得到卷积后的值，如图11。

References

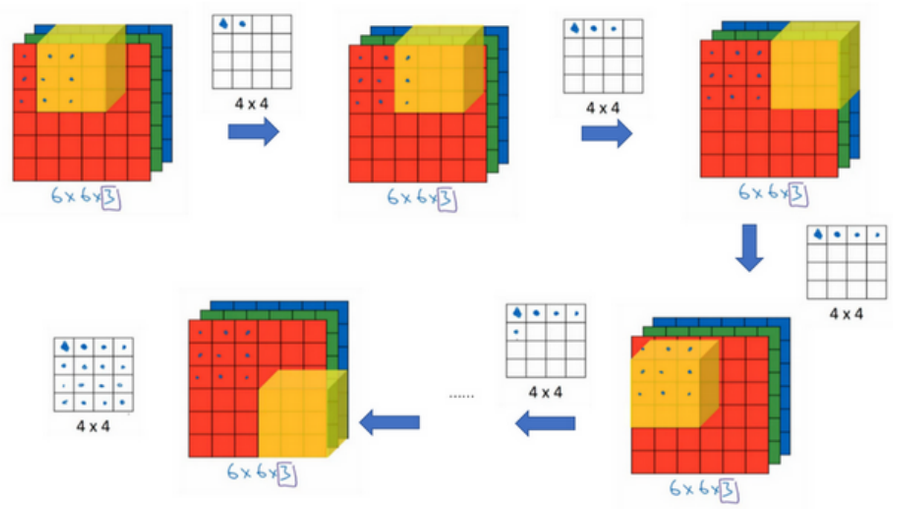

Figure 11: 彩色图片卷积操作。