

人工智能 - 神经网络

杜小勤

武汉纺织大学数学与计算机学院

2016/03/28

神经网络

- 典型任务 - 手写体数字的识别;
- 反传算法;

概述

神经网络为许多任务提供了最好的解决方案，例如图像识别、语音识别、自然语言处理。

任务：手写体识别

人类可以毫不费力地识别出它们！


A sample of handwritten text, specifically the number '504192', written in a cursive, informal style on a white background.

图 1-1: 手写体识别任务

人类大脑有 V1、V2、V3、V4、V5 等视觉皮层，它们由低到高逐级形成一个层级系统，能够逐级地处理更复杂的视觉信息。例如，V1 包含 1.4 亿个神经元，各层之间的连接达几十亿个。人类的视觉系统已经演化了几百万年！

DCNN——模拟视觉皮层结构

深度卷积神经网络（Deep Convolutional Neural Network, DCNN）是前向神经网络的一种类型，受生物视觉皮层结构的启发，能够异常出色地处理图像识别任务。

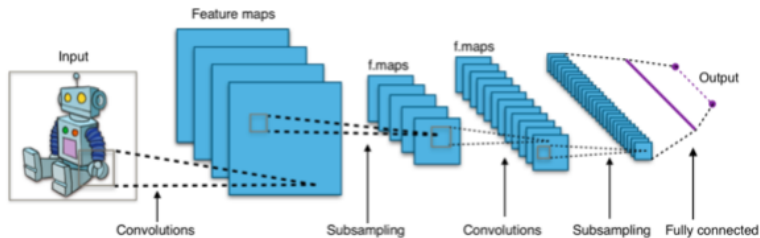


图 1-2: 一个典型的 DCNN 结构

手写体识别任务难吗？

采用传统的解决方案来处理，如何处理？

如何识别 0、1、2、...、9？

如何确定性地、精确地表达“识别算法”？

“复杂的算法，简单的数据” V.S. “简单的算法，复杂的数据”

“大数据”技术，强调从大量的数据中挖掘出有用的信息。

手写体识别任务的神经网络方法

神经网络从大量的手写体数字数据（被称为是“训练数据”）中学习，以权值的形式保存学习结果，从而可以“较容易地”识别出手写体数字——不仅仅是训练数据中的手写体数字，还包括其它的手写体数字（具有一定的泛化作用）。

手写体识别任务的训练数据



图 1-3: 手写体识别程序的训练数据样例

手写体识别任务的神经网络方法

神经网络方法能够从训练数据中自动地“推理”出识别手写体数字的特征，通过增加训练数据，可以提高识别精度。

一个简单的神经网络手写体数字识别程序可以达到 96% 的精度，再结合其它技术，就可以达到 99% 的精度。

商业化的识别程序已经用于银行的支票处理、邮局的邮编处理等。

神经网络

神经网络是由神经元按照一定的方式组合在一起的，神经元之间具有某种连接。神经网络种类较多。

下面将以“前向神经网络”（Feedforward Neural Networks, FNN）为例进行介绍。

一个典型的神经网络结构

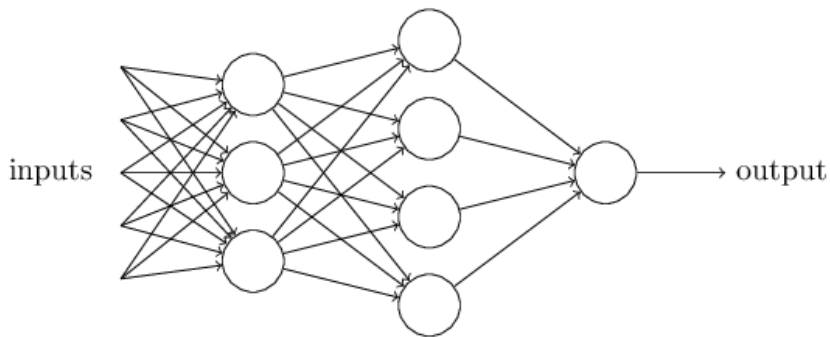


图 1-4: 一个典型的神经网络结构

简化结构

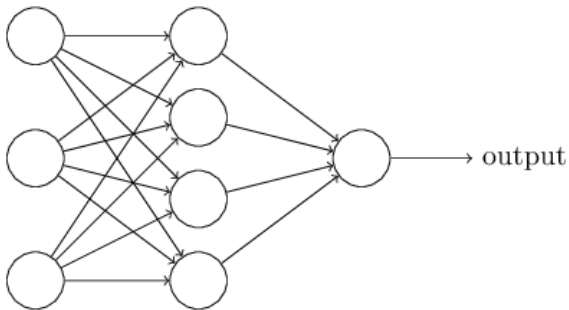


图 1-5: 简化的神经网络结构

一个复杂的神经网络结构

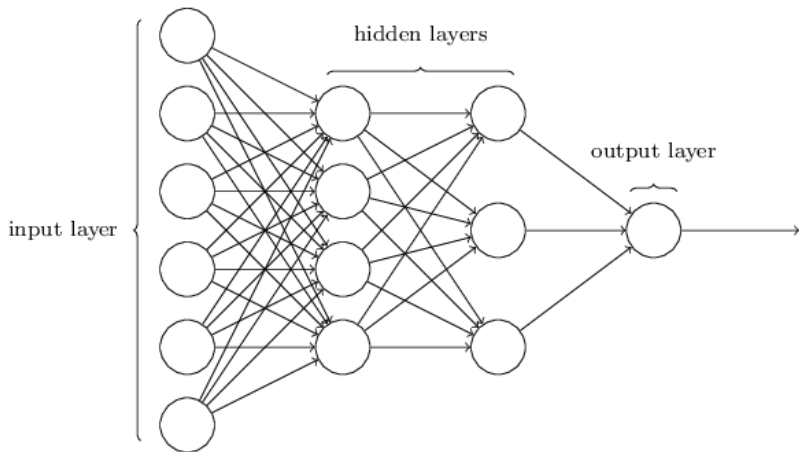


图 1-6: 一个复杂的神经网络结构

2 种神经元

- Perceptrons - 感知器
- Sigmoid - S 型

Perceptrons

Frank Rosenblatt 于 1950-60 年代提出。

现在，其它类型的神经元用得更加广泛，例如 Sigmoid 神经元。

可以通过理解 Perceptrons，来更好地理解 Sigmoid 神经元。

结构示例

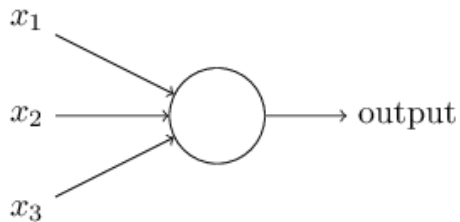


图 1-7: Perceptron 的结构示例

Perceptrons 示例

三个输入： x_1 、 x_2 、 x_2 ——表示输入数据，取值 0 或 1。

三个权值： w_1 、 w_2 、 w_3 ——表示输入数据的重要性。

输出：

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq threshold \\ 1 & \text{if } \sum_j w_j x_j > threshold \end{cases} \quad (1)$$

Perceptrons 就是一个决策模型！

决策：是否参加某个节日聚会，考虑的因素如下：

- 天气是否好： x_1 , w_1
- 是否有人陪伴你： x_2 , w_2
- 交通是否便利： x_3 , w_3

决策模型

假设模型参数是： $w_1 = 6$, $w_2 = 2$, $w_3 = 2$,
 $threshold = 5$ 。

当输入数据是： $x_1 = 1$, $x_2 = 0$, $x_3 = 0$, 有：

$$output = 1(\sum_j w_j x_j = 6 > threshold)。$$

意味着，在这个模型下，只有天气好，才会参加
节日聚会。

决策模型

改变模型参数，就可以获得不同的决策模型。

例如，设 $threshold = 3$ ，意味着，在这个新模型下，不管天气如何，只要后 2 个条件满足，就会参加节日聚会。 $threshold$ 表示对参加节日聚会的渴望程度，值越小，越渴望，或者神经元越易被激活。

比 Perceptrons 复杂的决策模型

神经网络是一个更加复杂的决策模型，如图1-4所示。

这是一个三层神经网络，图中的每一个神经元都可以执行简单的决策，后层神经元的输入是前层神经元的输出，它们在更抽象的层次上进行决策。

这个神经网络可以执行更加复杂的决策！

符号简化

为简化讨论，对公式（1）进行简化：

$$output = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (2)$$

$w \cdot x$ 是向量点积， w 是权值向量， x 是输入向量。

$b = -threshold$ ， b 是偏置（bias）。

w 的每个分量：表示对应的输入数据分量的重要程度，值越大，表示越重要。 b ：表示神经元被激活的难易程度，值越大，越易激活，值越小，越不易被激活（与 $threshold$ 相反）。

基本的逻辑函数

Perceptrons 的另一个作用是，可以计算基本的逻辑函数。例如，NAND 逻辑功能实现：

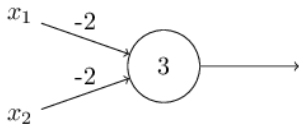


图 1-8: Perceptrons 的 NAND 实现

$$00 \rightarrow 1: (-2) \cdot 0 + (-2) \cdot 0 + 3 = 3 > 0$$

$$01 \rightarrow 1: (-2) \cdot 0 + (-2) \cdot 1 + 3 = 1 > 0$$

$$10 \rightarrow 1: (-2) \cdot 1 + (-2) \cdot 0 + 3 = 1 > 0$$

$$11 \rightarrow 0: (-2) \cdot 1 + (-2) \cdot 1 + 3 = -1 < 0$$

Perceptrons

实际上，上例表明，我们可以使用 Perceptrons 构成复杂网络来计算任意的逻辑函数。

超越 Perceptrons

由 Perceptrons 构成的神经网络似乎只能模仿 NAND 的功能。

但是假如，我们能够设计一种学习算法，让神经网络自动地依据输入数据来调整（训练）神经元的权值与偏置，那么它就可以学习任何的输入-输出映射（函数）了！

特别有意义的是，它不需要程序员的干预，也不需要程序员手工设计函数映射程序。并且在很多场合下，使用传统的方法来设计一个正确的、好的映射程序是极其困难的。

超越 Perceptrons

Perceptrons 神经元的主要问题在于，它的输出是非连续非平滑的（在 0 和 1 之间跳变），这对于神经网络的学习来讲是致命的，因为神经网络的学习是一个权值与偏置进行逐步调整并逐步反映到输出的过程。

它所期待的是，输入上微小的变化将引起输出上微小的变化，而不是跳变——后者让神经网络的学习变得非常不可控。

学习或训练：获取实际输出与理想输出之间的误差，反传调整所有权值与偏置。我们希望，它们的变化是连续平滑的，而不是突变的。

超越 Perceptrons

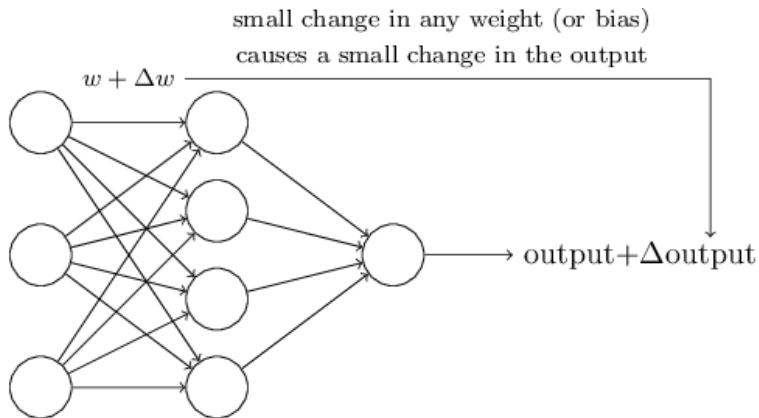


图 1-9: 神经网络权值调整的作用

超越 Perceptrons

如果采用 Perceptrons 做手写体数字识别系统的神经元，那么神经元的输出在学习或训练期间，随着权值与偏置的调整，有可能会在 0 和 1 之间跳变。导致的结果是，系统的识别调整不是微调的，也不是稳定的。

例如，假定在某一时刻，系统误把“8”当作“9”，并假定经过权值与偏置的调整，这种错误已经得到解决。但是，Perceptrons 神经元输出的跳变性，将会导致系统的不稳定性，并最终可能会导致其它已得到正确识别的数字产生识别问题。

Sigmoid 神经元

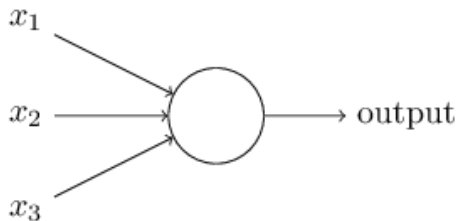


图 1-10: Sigmoid 的结构示例

Sigmoid 神经元

结构上，Sigmoid 神经元与 Perceptrons 一样，但是，它的输入数据与输出数据的范围不一样。

输入数据 x_1 、 x_2 、 x_2 的取值不再只能是 0 或 1，而是 $[0,1]$ 。

输出 *output* 也不再只能是 0 或 1，而是：

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (3)$$

其中， $z = \sum_j w_j \cdot x_j + b$ 。

Sigmoid 的激活函数

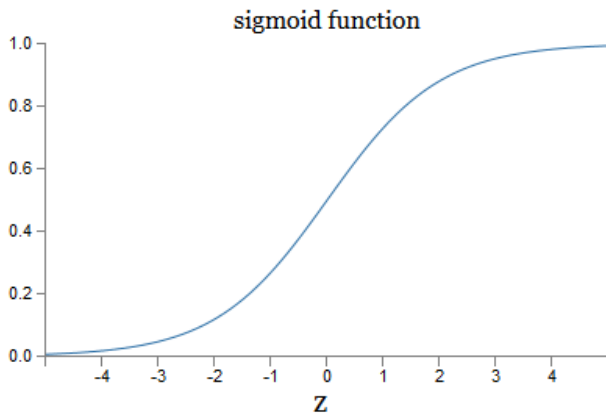


图 1-11: Sigmoid 的曲线图

Sigmoid V.S. Perceptrons

从 Sigmoid 函数的曲线图可以看出，当 z “很大”和“很小”时，两类神经元的行为是类似的。

当 z 适中时，Sigmoid 的行为与 Perceptrons 的行为非常不同。区别来自于 Sigmoid 函数与阶梯函数的不同。

Perceptrons 的激活函数——阶梯函数

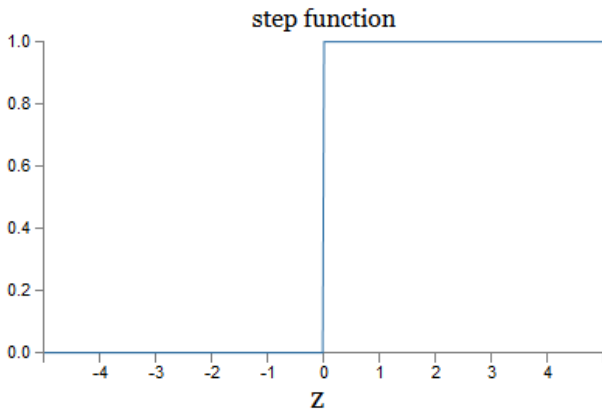


图 1-12: 阶梯函数的曲线图

Sigmoid 神经元

Sigmoid 的激活函数是连续平滑的。

权值与偏置上微小的变化将引起输出上微小的变化，即：

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b \quad (4)$$

上式表明， $\Delta output$ 是 Δw_j 和 Δb 的线性函数。

用于手写体数字识别的神经网络

输入层：神经元个数为 $28 \times 28 = 784$ （手写体数字图像的点阵：1 表示黑色像素点，0 表示白色像素点）。

隐藏层：神经元个数为 15 或 30。

输出层：神经元个数为 10，输出值 0000000000 表示数字 0，输出值 0100000000 表示数字 1，依次类推。

用于手写体数字识别的神经网络

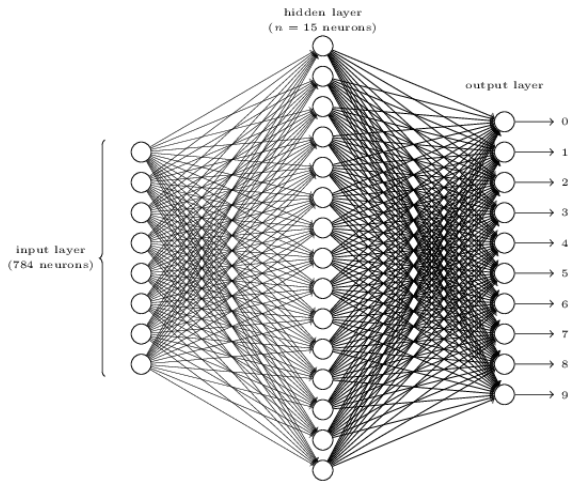


图 1-13: 用于手写体数字识别的神经网络

数据集

使用 MNIST 数据集。

60,000 幅图像，作为训练数据 (Training data)，
用于训练。

10,000 幅图像，作为测试数据 (Testing data)，
用于测试。

训练方法

采用基于梯度下降的反传算法
(Gradient-descent based backpropagation)

代价函数 / 目标函数

定义该问题的代价函数如下：

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2 \quad (5)$$

目标是找到一套权值 w 与偏置 b ，以最小化该代价函数或目标函数。

$C(w, b)$ 被称为是二次代价函数 (Quadratic cost function)，也被称为均方误差 (Mean squared error, MSE) 函数。

梯度下降原理

假设 $C(v_1, v_2)$ 是一个只有 2 个变量 v_1 和 v_2 的二次代价函数。

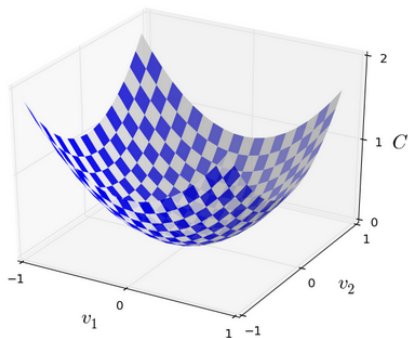


图 1-14: 2 变量的二次代价函数

梯度下降原理

我们的目标是最小化这个二次代价函数，即要找到代价函数的全局极小值。

对于只有几个变量的代价函数，我们可以简单地使用微积分的方法来计算全局极小值：求解“导数值 = 0”。

梯度下降原理

但是，对于像神经网络这样的系统，上述方法并不能奏效，因为神经网络涉及的变量——权值与偏置数量非常巨大。

我们要引入另一种方式来解决它。我们仍然要使用这个简单的、只有 2 个变量的代价函数作为实例来进行分析。

梯度下降原理

从代价函数的曲面上，随机地选取一个点，然后把一个球放在该点处，可以想象，该球将沿着曲面“径直”地滚落下去，最终将会停留在曲面的最低点处。

你能从上述描述中，得到一个解决方案吗？

新方法：我们可以采用某种方式来“模拟”这个过程！

梯度下降原理

对于代价函数 C 而言：

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \quad (6)$$

我们的新方法：不断地调整 Δv_1 和 Δv_2 ，使得 ΔC 是负的，这样就可以确保代价函数 C 不断地变小。

梯度下降原理

设：

$$\Delta v = (\Delta v_1, \Delta v_2)^T \quad (7)$$

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T \quad (8)$$

这样，公式（6）可以重新写成：

$$\Delta C \approx \nabla C \cdot \Delta v \quad (9)$$

梯度下降原理

观察公式 (9)，我们可以这样设置 Δv （它是一个向量，对于本例而言，分量分别是 Δv_1 和 Δv_2 ）：

$$\Delta v = -\eta \nabla C \quad (10)$$

其中， η 是一个小的正数，被称为学习率。

梯度下降原理

这样，综合公式 (9) 和公式 (10)，有：

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \quad (11)$$

上式确保了 $\Delta C \leq 0$ 。

梯度下降原理

综上所述，我们得到了一个全新的方法：

$$\Delta v = -\eta \nabla C \quad (12)$$

$$v \leftarrow v + \Delta v \quad (13)$$

上述调整过程一直迭代进行下去：计算梯度 ∇C ，计算 Δv ，计算 v ，计算代价 C ，直到代价函数满足某个要求为止。

上面的公式被称为是变量更新规则。

梯度下降算法

这个方法被称为是梯度下降算法，可以很自然地扩展到多变量的情况。

梯度下降算法

对于一般情形，假设变量为 v_1 和 v_2 ，那么公式 (12) 的分量形式为：

$$v_1 \leftarrow v_1 + \Delta v_1 = v_1 - \eta \frac{\partial C}{\partial v_1} \quad (14)$$

$$v_2 \leftarrow v_2 + \Delta v_2 = v_2 - \eta \frac{\partial C}{\partial v_2} \quad (15)$$

推广到神经网络

对于神经网络，对于任一权值变量 w_{jk}^l 和偏置变量 b_j^l ，它的基本形式是一样的：

$$w_{jk}^l \leftarrow w_{jk}^l + \Delta w_{jk}^l = w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l} \quad (16)$$

$$b_j^l \leftarrow b_j^l + \Delta b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l} \quad (17)$$

其中， w_{jk}^l 表示第 $(l-1)$ 层的第 k 个神经元到第 l 层的第 j 个神经元之间的权值， b_j^l 表示第 l 层的第 j 个神经元的偏置。

推广到神经网络

我们需要针对神经网络中每一层的每一个权值变量和每一个偏置变量，推导出它们的更新规则。

注意到，公式（5）表示的代价函数有如下形式：

$$C = \frac{1}{n} \sum_x C_x \quad (18)$$

它表示 n 个训练样例的代价函数。其中，每一个训练样例的代价函数是：

$$C_x = \frac{\|y(x) - a\|^2}{2} \quad (19)$$

推广到神经网络

因此，对于 n 个训练样例：

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \quad (20)$$

当训练样例的个数非常大时，神经网络的训练将会非常慢。

批梯度下降 V.S. 随机梯度下降

Batch Gradient Descent (BGD): 每次使用所有的训练样例;

Stochastic Gradient Descent (SGD): 每次使用一个训练样例或一小部分训练样例 (例如 10 个或 100 个);

随机梯度下降

代价函数的梯度可以近似为：

$$\nabla C \approx \frac{1}{m} \sum_{o=1}^m \nabla C_{x_o} \quad (21)$$

其中， m 是 mini-batch 的样例数目。

随机梯度下降

这样，使用 mini-batch 下的权值与偏置更新公式如下：

$$w_{jk}^l \leftarrow w_{jk}^l + \Delta w_{jk}^l = w_{jk}^l - \frac{\eta}{m} \sum_o \frac{\partial C_{x_o}}{\partial w_{jk}^l} \quad (22)$$

$$b_j^l \leftarrow b_j^l + \Delta b_j^l = b_j^l - \frac{\eta}{m} \sum_o \frac{\partial C_{x_o}}{\partial b_j^l} \quad (23)$$

基于随机梯度下降的反传算法

前面介绍的是梯度下降方法的整体：针对某个代价函数，如何使用梯度下降方法来更新权值与偏置，以达到最小化代价函数的目的。

问题是，如何有效地计算出每个梯度 $\frac{\partial C}{\partial w_{jk}^l}$ 和 $\frac{\partial C}{\partial b_j^l}$ 。

基于随机梯度下降的反传算法

反传 (Backpropagation) 算法是一种计算每个权值与偏置变量的梯度的方法，基于反传算法，我们可以不断地改变权值与偏置，来逐渐地降低整体代价，直到代价满足我们的要求。

为方便起见，我们需要引入一个中间量 δ_j^l ，它表示第 l 层的第 j 个神经元的误差，最后，我们将通过中间量 δ_j^l 来导出梯度 $\frac{\partial C}{\partial w_{jk}^l}$ 和 $\frac{\partial C}{\partial b_j^l}$ 。

注意：以下的公式推导是针对一个样例的情况，多个样例的梯度就是梯度的平均值。

基于随机梯度下降的反传算法

我们定义中间量 δ_j^l :

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \quad (24)$$

其中, z_j^l 是第 l 层的第 j 个神经元的权值输入:

$$z_j^l = \sum_o w_{jo}^l a_o^{l-1} + b_j^l \quad (25)$$

其中 a_o^{l-1} 表示第 $(l-1)$ 层第 o 个神经元的输出
 $a_o^{l-1} = \sigma(z_o^{l-1})$ 。

基于随机梯度下降的反传算法

首先，推导出神经网络的最后一层 L 的误差：

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \quad (26)$$

$$= \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) \quad (27)$$

基于随机梯度下降的反传算法

如果代价函数使用公式 (5)，那么

$\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$ ，其中， y_j 是神经网络输出层的第 j 个神经元的理想输出。

如果 $\sigma(z_j^L)$ 函数是 Sigmoid 函数，那么

$$\sigma'(z_j^L) = \sigma(z_j^L)(1 - \sigma(z_j^L))。$$

基于随机梯度下降的反传算法

它的向量形式是：

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (28)$$

注意， \odot 表示 Hadamard 积， $\nabla_a C$ 是一个向量，它的分量是 $\frac{\partial C}{\partial a_j^L}$ 。

如果代价函数使用公式 (5)，那么 $\nabla_a C = (a^L - y)$ 。其中， a^L 是输出层的实际输出向量， y 是输出层的理想输出向量。

基于随机梯度下降的反传算法

层 l 与层 $(l+1)$ 的 δ 之间的关系:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \quad (29)$$

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \quad (30)$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \quad (31)$$

基于随机梯度下降的反传算法

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \cdot \sigma'(z_j^l) \quad (32)$$

向量形式：

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \cdot \sigma'(z^l) \quad (33)$$

基于随机梯度下降的反传算法

偏置的梯度：

$$\frac{\partial C}{\partial b_j^l} = \sum_k \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} \quad (34)$$

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad (35)$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \quad (36)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (37)$$

基于随机梯度下降的反传算法

权值的梯度：

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_i \frac{\partial C}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l} \quad (38)$$

$$z_j^l = \sum_o w_{jo}^l a_o^{l-1} + b_j^l \quad (39)$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \quad (40)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (41)$$

基于随机梯度下降的反传算法

1. Input a set of training examples

2. For each training example x : Set the corresponding input activation $a^{x,1}$, and perform the following steps:

- **Feedforward:** For each $l = 2, 3, \dots, L$ compute

$$z^{x,l} = w^l a^{x,l-1} + b^l \text{ and } a^{x,l} = \sigma(z^{x,l}).$$

- **Output error $\delta^{x,L}$:** Compute the vector

$$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L}).$$

- **Backpropagate the error:** For each

$$l = L - 1, L - 2, \dots, 2 \text{ compute}$$

$$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l}).$$

3. Gradient descent: For each $l = L, L - 1, \dots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

图 1-15: Backpropagation 算法

学习缓慢问题——神经元饱和

如果使用公式 (5) 作为代价函数，并且使用 Sigmoid 作为神经元的激活函数，那么在输出层就存在学习缓慢的问题（实际上，由于 Sigmoid 神经元的使用，其它层也存在这个问题）。

主要原因是：Sigmoid 在两端的梯度很小，导致权值与偏置调整量也很小。例如，输出层的误差公式 (26)：
$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L)。$$

这个问题被称为神经元饱和问题，是神经网络系统中存在的一个重要问题。

解决办法之一

定义新的代价函数：cross-entropy 代价函数

$$C(w, b) = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] \quad (42)$$

其中， n 是训练数据的数目， y_j 是输出层第 j 个神经元的理想输出， a_j^L 是输出层第 j 个神经元的实际输出。

注意：神经元仍然采用 Sigmoid 函数。

cross-entropy 代价函数

它满足两个条件：

- $C(w, b)$ 是非负的： $y_j \in [0, 1]$,
 $a_j^L \in (0, 1) \Rightarrow \ln a_j^L < 0$;
- 如果 y_j 和 a_j^L 很接近，那么 $C(w, b)$ 就接近于 0；

cross-entropy 代价函数

还是以单个样例为例。关键量 δ_j^L 中因子 $\frac{\partial C}{\partial a_j^L}$ 的计算：

$$\frac{\partial C}{\partial a_j^L} = -\left(\frac{y_j}{a_j^L} - \frac{1 - y_j}{1 - a_j^L}\right) = \frac{1 - y_j}{1 - a_j^L} - \frac{y_j}{a_j^L} \quad (43)$$

$$= \frac{a_j^L - y_j}{a_j^L(1 - a_j^L)} = \frac{a_j^L - y_j}{\sigma(z_j^L)(1 - \sigma(z_j^L))} = \frac{a_j^L - y_j}{\sigma'(z_j^L)} \quad (44)$$

cross-entropy 代价函数

将 $\frac{\partial C}{\partial a_j^L} = \frac{a_j^L - y_j}{\sigma'(z_j^L)}$ 代入公式 (26), 得到:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) = a_j^L - y_j \quad (45)$$

上式表明, cross-entropy 代价函数的引入, 消去了输出层 δ_j^L 中的 Sigmoid 梯度 $\sigma'(z_j^L)$ 。

何时使用二次代价函数？

一般而言，如果输出层的神经元使用 Sigmoid 激活函数（特别是分类 classification 问题），那么使用 cross-entropy 代价函数总是合适的。

但是，如果输出层的神经元使用线性激活函数（对于回归 regression 问题），即 $a_j^L = z_j^L$ ，那么使用二次代价函数就是合适的，因为 $\delta_j^L = (a_j^L - y_j)$ 本身就成立。

Softmax 激活函数与 Softmax 层

输出层使用 Softmax 激活函数，形成 Softmax 层：

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L \quad (46)$$

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \quad (47)$$

Softmax 层的输出可以被看作是一个概率分布。

许多问题可以被解释成概率分布。例如，MNIST 分类问题，它的 a_j^L 就可以被看作是正确数字分类 j 的一个概率估算。

解决办法之二：Softmax 层 +log-likelihood 代价函数

Softmax 输出层使用 log-likelihood 代价函数：

$$C(w, b) = -\frac{1}{n} \sum_x \ln a_y^L \quad (48)$$

其中， n 是训练数据的数目， y 是 Softmax 输出层的理想分类编号：第 y 个输出神经元的理想输出是 1，其余神经元的输出是 0)， a_y^L 是第 y 个输出神经元的实际输出。例如在 MNIST 中，如果 $y = 7$ ，表明正确分类是数字 7，那么输出层的第 7 个神经元的理想输出是 1，其余全部是 0。

log-likelihood 代价函数

还是以单个样例为例：

$$C_x(w, b) = -\ln a_y^L = -\ln \frac{e^{z_y^L}}{\sum_o e^{z_o^L}} \quad (49)$$

$$= \ln \sum_o e^{z_o^L} - z_y^L \quad (50)$$

log-likelihood 代价函数

如果 $y = j$:

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} (\ln \sum_o e^{z_o^L} - z_y^L) = \frac{e^{z_j^L}}{\sum_o e^{z_o^L}} - 1 \quad (51)$$

$$= a_j^L - 1 \quad (52)$$

如果 $y \neq j$:

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} (\ln \sum_o e^{z_o^L} - z_y^L) = \frac{e^{z_j^L}}{\sum_o e^{z_o^L}} - 0 \quad (53)$$

$$= a_j^L - 0 \quad (54)$$

log-likelihood 代价函数

综合 2 种情况：

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^L} = a_j^L - \delta_{yj} \quad (55)$$

$$\delta_{yj} = \begin{cases} 1 & j = y \\ 0 & j \neq y \end{cases} \quad (56)$$

log-likelihood 代价函数

偏置的梯度：

$$\frac{\partial C}{\partial b_j^L} = \sum_k \frac{\partial C}{\partial z_k^L} \frac{\partial z_k^L}{\partial b_j^L} = \frac{\partial C}{\partial z_j^L} \frac{\partial z_j^L}{\partial b_j^L} = \delta_j^L \frac{\partial z_j^L}{\partial b_j^L} \quad (57)$$

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L \quad (58)$$

$$\frac{\partial z_j^L}{\partial b_j^L} = 1 \quad (59)$$

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L = a_j^L - \delta_{yj} \quad (60)$$

log-likelihood 代价函数

权值的梯度：

$$\frac{\partial C}{\partial w_{jk}^L} = \sum_i \frac{\partial C}{\partial z_i^L} \frac{\partial z_i^L}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = \delta_j^L \frac{\partial z_j^L}{\partial w_{jk}^L} \quad (61)$$

$$z_j^L = \sum_o w_{jo}^L a_o^{L-1} + b_j^L \quad (62)$$

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1} \quad (63)$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1} \delta_j^L = a_k^{L-1} (a_j^L - \delta_{yj}) \quad (64)$$

log-likelihood 代价函数

比较公式 (55) 与公式 (45)，可以发现，它们实际上是一致的。它们都用来解决学习缓慢的问题。

使用 log-likelihood 代价函数的 Softmax 输出层与使用 cross-entropy 代价函数的 Sigmoid 输出层，两者非常类似。

log-likelihood 代价函数

下面将使用另一种方法来推导上述公式。由于推导过程中要用到 Softmax 激活函数的偏导数，所以先给出它的偏导数推导过程。

仍然以 a_i^L 来表示 Softmax 输出层 L 的第 i 个神经元，它的输出值为：

$$a_i^L = \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} \quad (65)$$

log-likelihood 代价函数

如果 $j = i$:

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} = \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} - e^{z_i^L} \cdot \frac{e^{z_i^L}}{(\sum_k e^{z_k^L})^2} \quad (66)$$

$$= a_i^L - (a_i^L)^2 = a_i^L(1 - a_i^L) = a_j^L(1 - a_j^L) \quad (67)$$

如果 $j \neq i$:

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} = -e^{z_i^L} \cdot \frac{e^{z_j^L}}{(\sum_k e^{z_k^L})^2} \quad (68)$$

$$= -a_i^L a_j^L \quad (69)$$

log-likelihood 代价函数

Softmax 输出层使用 log-likelihood 代价函数：

$$C(w, b) = -\frac{1}{n} \sum_x \sum_k y_k^x \ln a_k^L \quad (70)$$

其中， n 表示训练数据的数目， k 表示 Softmax 输出层中第 k 个神经元， y_k^x 表示（与训练数据 x 对应的）Softmax 输出层中第 k 个神经元的理想输出，取值 0 或 1。例如，对于 MNIST 分类问题，如果某个 x 训练数据的理想输出是向量 $y^x = 0100000000$ ，则表示对应的是数字 1，有 $y_2^x = 1$ ， $y_k^x = 0$ ，对于 $k \neq 2$ 。 a_k^L 表示 Softmax 输出层中第 k 个神经元的实际输出。

log-likelihood 代价函数

还是以单个样例为例：

$$C_x(w, b) = - \sum_k y_k \ln a_k^L \quad (71)$$

log-likelihood 代价函数

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L} \left(- \sum_k y_k \ln a_k^L \right) = - \sum_k y_k \frac{1}{a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad (72)$$

$$= -y_j \frac{1}{a_j^L} \frac{\partial a_j^L}{\partial z_j^L} - \sum_{k \neq j} y_k \frac{1}{a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad (73)$$

$$= -y_j \frac{1}{a_j^L} a_j^L (1 - a_j^L) - \sum_{k \neq j} y_k \frac{1}{a_k^L} (-a_j^L a_k^L) \quad (74)$$

$$= -y_j + y_j a_j^L + \sum_{k \neq j} y_k a_j^L = -y_j + a_j^L \sum_k y_k \quad (75)$$

$$= a_j^L - y_j \quad (76)$$

log-likelihood 代价函数

可以看出，它与公式 (55) 的意义是一样的。

权值与偏置的偏导推导过程与前面一样，此处不再赘述。

Overfitting

训练和设计神经网络使用三类数据：Training data、Validation data、Testing data。

Overfitting: 从统计学的角度来看，如果模型反映的是数据中的随机噪声或误差，而不是以模型误差的形式反映数据中的内在关系时，就发生了Overfitting。

对于神经网络而言，Overfitting 发生时，泛化能力就降低了，此时，神经网络“精确”地拟合了Training data，而不能泛化到其它未“看见”过的数据上。

Overfitting

Training data: 用于训练神经网络，调整权值与偏置；

Validation data: 用于 Overfitting 的判断及早期停止 (Early stop)，如果神经网络在 Training data 上的精度在提高，但是在 Validation data 上的精度维持不变或降低，表明 Overfitting 发生了，应该停止训练；

Testing data: 用于最终测试神经网络的能力；

解决 Overfitting 问题的方法之一

方法之一：增加 Training data 的规模。

例如，采集更多的 Training data，或者通过平移、旋转、镜像等方式从原始的 Training data 中生成更多新的 Training data。

解决 Overfitting 问题的方法之二

解决 Overfitting 的方法之二：使用 Regularization 技术。

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \quad (77)$$

其中， C_0 是原始代价函数，例如二次代价函数、cross-entropy 代价函数、log-likelihood 代价函数等。 $\lambda > 0$ 是调节参数， n 是训练数据的数目， w 是神经网络的权值（项）。该调节技术被称为是 L2 regularization。

Regularization

总体上，Regularization 技术使得神经网络偏爱较小的权值。

公式 (77)，在形式上非常类似于 UCT (Monte Carlo Tree Search+UCB) 中的 UCB 公式，该公式反映了 Exploration-Exploitation 之间的某种平衡，公式的第 1 项反映了 Exploitation，第 2 项反映了 Exploration。

公式 (77)，反映了代价函数最小化与权值最小化之间的某种平衡。 λ 决定了它们之间的相对重要性。

Regularization

带调节项的一般性权值偏导公式（一个训练样例）：

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C_0}{\partial w_{ij}^l} + \frac{\lambda}{n} w_{ij}^l \quad (78)$$

偏置偏导公式不变，还是：

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C_0}{\partial b_j^l} \quad (79)$$

其中， $\frac{\partial C_0}{\partial w_{ij}^l}$ 与 $\frac{\partial C_0}{\partial b_j^l}$ 的公式在前面已经推导出来了。

Regularization

带调节项的一般性权值更新公式（一个训练样例）：

$$w_{ij}^l = w_{ij}^l - \eta \frac{\partial C_0}{\partial w_{ij}^l} - \frac{\eta \lambda}{n} w_{ij}^l = (1 - \frac{\eta \lambda}{n}) w_{ij}^l - \eta \frac{\partial C_0}{\partial w_{ij}^l} \quad (80)$$

偏置更新公式不变，还是：

$$b_j^l = b_j^l - \eta \frac{\partial C_0}{\partial b_j^l} \quad (81)$$

其中， $1 - \frac{\eta \lambda}{n}$ 是权值衰减（weight decay）因子。

Regularization

带调节项的一般性权值更新公式 (mini-batch):

$$w_{ij}^l = w_{ij}^l - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w_{ij}^l} - \frac{\eta \lambda}{n} w_{ij}^l \quad (82)$$

$$= (1 - \frac{\eta \lambda}{n}) w_{ij}^l - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w_{ij}^l} \quad (83)$$

偏置更新公式不变，还是：

$$b_j^l = b_j^l - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b_j^l} \quad (84)$$

Regularization

在 MNIST 上的实验结果表明，给代价函数增加调节项具有如下优点：

- 减少 overfitting;
- 增加分类精度;
- 可以提高神经网络的稳定性。如果不使用调节项，神经网络有时候会陷入局部极小，不稳定;

参考文献

- [1] Neural Networks and Deep Learning.
- [2] Wikipedia: Convolutional neural network.
- [3] MNIST data set.
- [4] Softmax Layer and Log-likelihood loss function
- [5] Softmax vs. Softmax-Loss: Numerical Stability
- [6] What's the difference between 3 data sets
- [7] Overfitting