神经网络（前馈型）的基本概念与实现

2021年2月7日　13:40

参阅的博客链接

https://blog.csdn.net/qq_31192383/article/details/77145993
https://blog.csdn.net/qq_31192383/article/details/77198870
https://blog.csdn.net/qq_31192383/article/details/77429409
https://blog.csdn.net/weixin_38347387/article/details/82936585

神经网络

一种可以通过观测数据使计算机学习的仿生语言案例。

深度学习

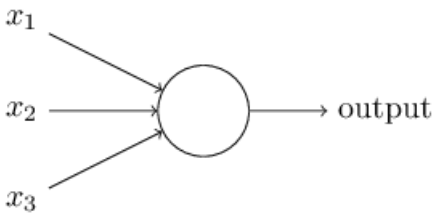
一组强大的神经网络学习技术

神经元

感知机

一类人造神经元（模拟神经元）

结构



输入

一组二进制的参数（x1,x2...）

输出

二进制输出（output）

参数

权重

针对每项输入利用权重来表示该项输入针对输出的重要性。

阈值

感知机的输出或为1或为0，由各项输入与权重和是否超过阈值决定。

输出判断公式

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

W 与 X 为权重值与输入值组成的向量。用向量积 W·X 来代替 ∑jwjxj。并将阈值移到等式左边得到。

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

其中 b = -threshold，其意义为感知机偏差 ， 代表感知机输出1的难易程度。从生物学的角度来讲其衡量了该神经元容易被激活的程度。

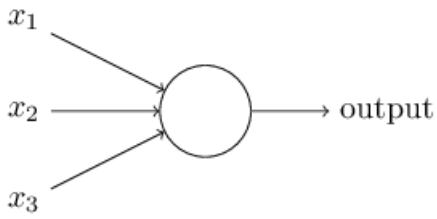
修改权重的学习过程

为了识别特定的内容，可以通过微调权重与偏差的参数让感知机的输出来接近预期想要的结果。但是对于感知机而言，任何感知机上发生的一小点改变都可能导致结果反转（即识别结果要么不变要么置反）。那么细微调节一个参数后感知机剧烈输出变化可能会引起连锁反应影响到一整个识别框架的识别结果，比如调好了识别数字 9 的参数以后可能识别其他数字的功能变得面目全非。那么通过细微地反馈调参以此让机器获得学习的这一过程变得难以实现。

为解决这个问题，提出了sigmoid神经元。

sigmoid神经元

结构



输入

其结构与感知机类似，不过其输入的参数不再仅限于 0 或 1。而是 0 到 1 之间的连续值。

输出

其输出也不再仅仅是 0 或 1。而是 σ(w·x+b) 。其中 σ 被成为 sigmoid 函数，定义为：

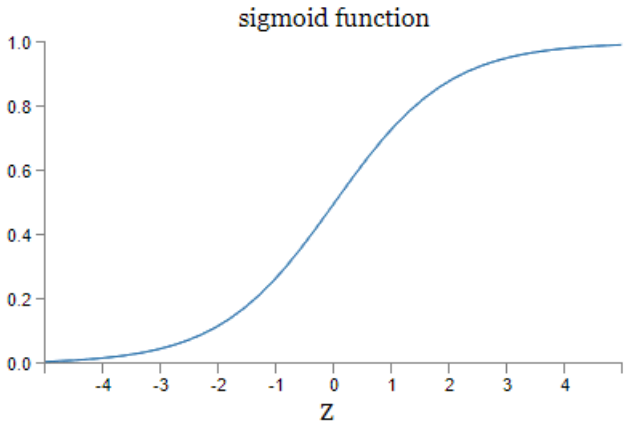
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

一个拥有输入 x1,x2...x1,x2... 权重 w1,w2...w1,w2... 偏差 b 的 sigmoid 神经元的输出为：

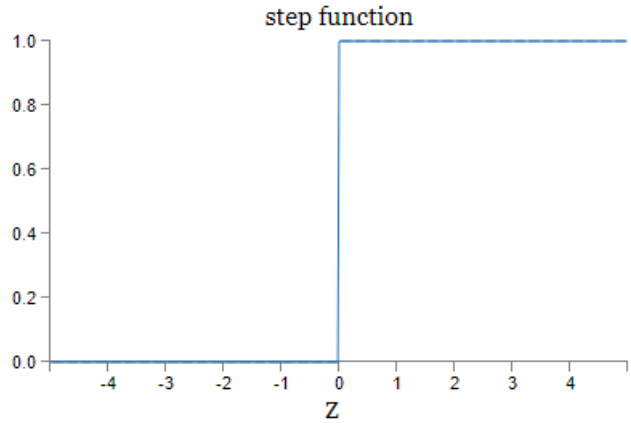
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}. \tag{4}$$

事实上 sigmoid 神经元的输出行为和感知机很类似，在 z 为一个很大的正数时 σ 趋近于 1，反之为一个很小的负数时趋近于 0，只有在中间部分其输出与感知机有不同之处。

对于 σ，我们不需要对其精确表达有明确的理解，只需要熟悉其函数图像。相比于感知机的要么 0 要么 1 的跳跃式变化，其曲线变得平缓。其输入的微小改变也只会导致输出的微小改变，避免了结果反转带来的预期外的错误。



感知机的激活函数为阶跃函数。



Sigmoid 函数就是对阶跃函数做了平滑处理之后的样子。

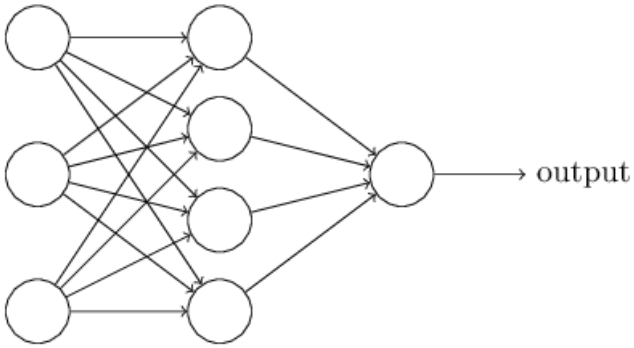
正如前面所说，σ 其本身作为平滑函数，其函数图像（平滑功能）比起其本身的函数表达式重要（比如 sigmoid 函数其仅仅只是作为激活函数的一种），因为有了这平滑处理，才得以避免结果反转，其输出量的微小变化可以近似地用表达式表达为：

$$\Delta \text{output} \approx \sum_j \frac{\partial \,\text{output}}{\partial w_j} \Delta w_j + \frac{\partial \,\text{output}}{\partial b} \Delta b, \qquad (5)$$

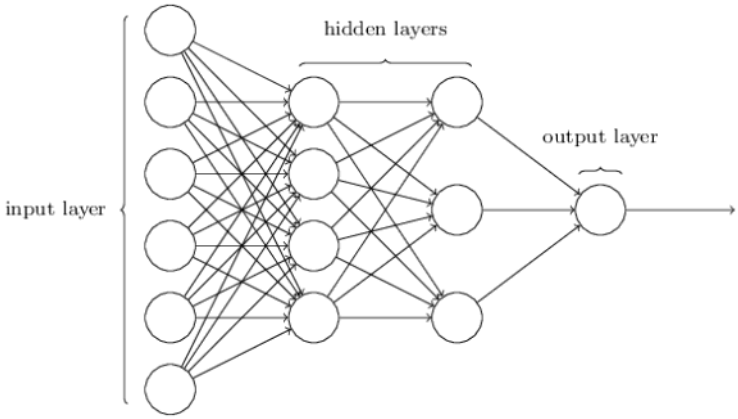
该偏导数的意义就是，权重 w 与阈值 b 对输出结果的影响大致是线性的，在激活函数发生改变时，唯一需要改变的就是偏导数的形式。

神经网络

架构



最左侧为输入层，其神经元被成为输入神经元。
最右侧为输出层，其神经元被称为输出神经元。
中间的部分为隐藏层，其可以为一层亦可以为多层。如下图为多隐藏层的情况。



// 多层的网络结构有时被成为 MLPs 即多层感知机，虽然该网络由 sigmoid 神经元组成。

这些模型中并不存在回路，上一层的输出将作为下一层的输入，被称为前馈型神经网络。意味着信息只会往

前走，不会出现信息流的环状结构。这种结构将会出现 σ 输出依赖于其输入的情况。

本次实践大多情况仅考虑前馈网络的结构。

示例

考虑完成手写数字识别功能的神经网络模型。

比如其系统输入为 64*64 的一张灰度图片。

那么体现在神经网络模型中其输入就是 64*64 = 4,096 个输入神经元，其输入参数为在 0 到 1 之间的灰度值。输出神经元只有一个，即若 output < 0.5 则为目标数字，否则反之。下面深入分析下其实现过程。

手写数字识别分类网络

模块

当输入的图像为一串手写的数字时，手写数字识别主要被划分两个问题。

手写数字的分片

即将图片中的一串手写数字划分为一组分开来的单个手写数字。

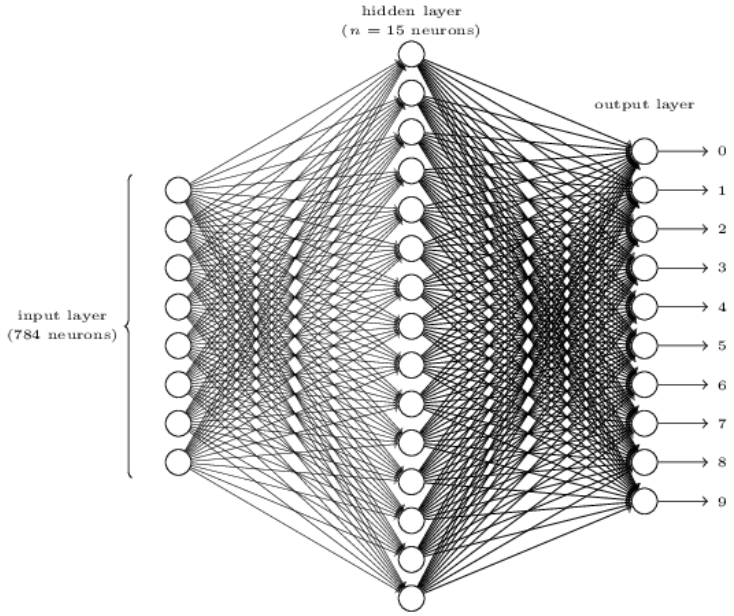
单个手写数字的识别

即对划分出的单个手写数字图片进行构建网络识别。

显然，本次博客内容主要在于第二个方面，也就是识别单个手写数字的部分。

结构

为了识别单个数字，将用到三层神经网络。



输入层

包含对输入像素进行编码的神经元。如同上文末尾提到的输入方式。

训练数据有许多28X28的手写数字的像素图组成，也就是说我们的输入层包含了28X28=784个神经元，一个神经元对应一个像素位的值。输入的像素值为灰度值，0.0表示白色，1.0表示黑色，0到1之间的值表示不同程度的灰色。

隐藏层

记隐藏层的神经元数量为n，将对n的取值进行实验。

输出层

网络的输出层包含了10个神经元。如果第一个神经元被激活，也就是输出值无限接近1，那么可以认为这个网络识别出来的数字是0.如果是第二个神经元激活，那么识别出来的数字为1。更准确的说，对输出神经元从0到9编号，然后找出激活值最高的神经元。如果编号为6的神经元激活值最高，那么认为输入的数字为6。

其他思路比较

问题

在输出层这一块，这里采用了10个神经元来做输出，但是通过简单的信息存储的知识可以知道，4

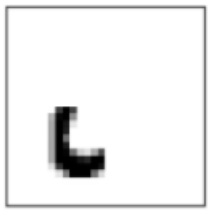
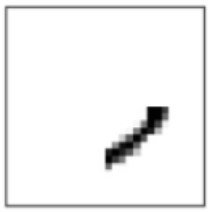
位的二进制编码就足以表示 0 - 9 这10个数字。
那么是否采取 4 位的结果输出更具效率。

结果表示，10 位的输出能够更准确地识别出手写数字。

解答

因为在隐层层处理图形的过程中，这里将图像进行分割处理，比如一号神经元处理的是

同样的二号三号四号神经元需要处理的是，

这些图像，那么如果一号神经元识别了第一张图片认为其是数字 0 的左上角组成，但是四位输出结果中 0000 0001 0010 ... 等等其第一位都是 0，那么其处理输出结果（数据）无法有效地与现实情况（图片）对应。
当然这只是在分割图像进行处理这一思路上会出现的问题。如果通过别的思路处理也许四位输出会表现地比十位更好。

理解

对于深度神经网络算法，其中间隐藏层的处理过程针对不同问题可以有很多不同的处理办法，其输出层也会受隐藏层的不同处理方式而采用不同的方式输出。

算法分析

训练输入

x是一个28X28=784的向量，向量中每一个元素表示图像中的一个灰度值。

期望输出

y=y(x)，其中 y 是一个10维的向量。比如，有一个特定的显示为6的图像输入为 x，那么它期望的输出值应该为 $y(x)=(0,0,0,0,0,0,1,0,0,0)^T$

找到权重与偏差

让输出 y(x) 能够拟合所有的 x 输入。定义一个代价函数来量化需要实现的目标。

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|y(x)-a\|^2. \qquad (6)$$

// 符号 ||v|| 表示的是向量 v 的长度（模）
C(w, b) 被称为二次代价函数。其中 a 代表当输入 x 时网络的输出向量，其中 a 的值取决于 w, b, x。
在式（6）中

$$\sum_x \|y(x)-a\|^2.$$

这一部分代表了在所有情况（所有的 x 取值）下 y(x) 与 a 偏离程度的累积，其数学意义就是 y(x) 与 a 的拟合程度的高低，实际意义就是现实情况与理想情况的差距程度。
当 a 趋近于 y(x) 时，C(w, b) 趋近于 0。这种情况就是目标情况。
训练的目的就是为了找到一组合适的权重与偏差（w，b），让 C(w, b) 最小。即最小化代价函数。

问题

为什么不直接尝试最大化正确识别图像的数量，反而去尝试让代价函数最小化？

解答

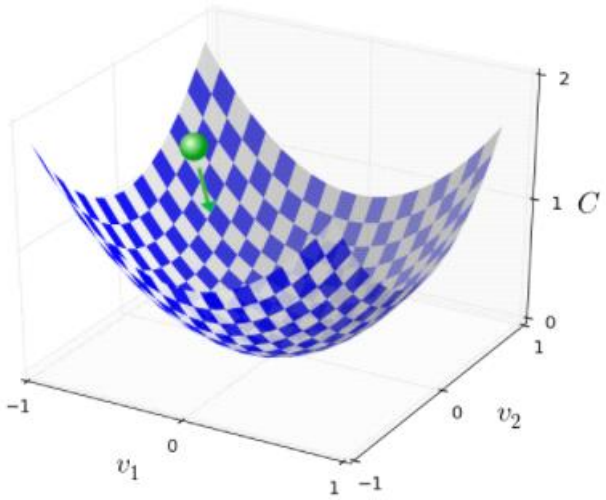
因为相比于代价函数，正确识别的数量是离散的，在面对权重与偏差的微调，其不一定能及时提供微小的改变来帮助提高性能，而代价函数是平滑的，它能够更灵敏地反映出改变权重与偏差对系统带来的变化。

梯度下降法

这里我们专注于将目标函数最小化，不去考虑其他额外的问题，单纯通过数学的推导试图找到让目标函数最小化的方法。

化简得到关系函数

模拟小球滑落山谷的过程。山谷（二维输入的函数图像），小球高度（目标函数值）。



当球在 v1 方向上移动 Δv1，在 v2 方向上移动 Δv2。微积分计算得到 C（高度）的改变量为：

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{7}$$

需要找到一种方法，使得能够找到 Δv2 和 Δv2 使得 ΔC 为负。即找到每一小步小球的运动方向，使得小球的高度是在下降的。

定义 Δv 为 v 的变化向量，

定义C的梯度为偏导数的向量：$(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$。

用 ∇C 表示梯度向量（个人理解：表示当前位置的地形情况，即影响球运动后高度改变的唯一因素）：

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T. \tag{8}$$

有了上面的定义，高度改变量 ΔC 可以写为：

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{9}$$

根据关系函数确定 Δv 的取值策略

在这个式子中，∇C 是已知量，其取决于当前小球所在山谷的位置（即当前的参数状态）。那么 ΔC 与 Δv 的关系就变得十分明了，我们也能够比较清晰地看出 Δv 的选择策略。

$$\Delta v = -\eta \nabla C, \tag{10}$$

// η 为一个很小的正参数。

如此一来就有：

ΔC ≈ −η ΔC ·ΔC = −η ‖ ∇C ‖^2

从这条式子不难看出，按照这种 Δv 的取值策略，ΔC 恒小于 0。那么小球的位置计算公式可以得到为：

$$v \to v' = v - \eta \nabla C. \tag{11}$$

算法本质

让小球永远沿着当前地形的下降最快的地方滑落。

如果找到滑落最快的移动方向就解决了这个问题，而根据两个维度的偏导方向的反方向移动一定是下降地最快的。比如说，往 x 正方向的偏导为一个正数，从几何上来讲它是上升的趋势，所以要向低处滑落要沿其反方向滑落；反之为负数，几何上是下降趋势，应当沿其同方向滑落，但是因为其偏导为负数，所以为了同方向滑落也需要将偏导取反。

这仅仅是在一维层面的理解，如果只是像上面那样解释的话，那么为什么偏偏要沿二维偏导的反方向滑落呢，因为偏导的大小体现了地形的陡峭程度，经过二者偏导数合成的反方向可以被证明是下落速度最快的方向。

应用梯度下降法

梯度下降法应用到神经网络的代价函数最小化，就是将代价函数作为目标函数带入前面的算法当中。可以得到权重 w 与偏差 b 的更新函数。

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{16}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{17}$$

问题

这里的代价函数可以以这样的形式表达：C = (∑x Cx) / n。要得到目标代价函数的最小值，需要对所有的 Cx 进行梯度下降算法操作，这会使得训练时间过长。

有一种算法叫做随机梯度下降法可以用来加速学习。

随机梯度下降法

思想

通过随机选取小的训练样本来计算 ∇Cx 来近似估计 ∇C。通过平均这一个小的样本就可以快速的估计出梯度 ∇C，这有助于帮助加速梯度下降，进而更快的学习。更精确地说，随机梯度下降是通过随机选取一个小数量地训练样本 m 作为输入。 将这些随机的输入样本记为 X1,X2,...,Xm，并把它们称为一个小批量数据。假设提供的样本容量 m 是足够使得 ∇Cxj 与计算所有得样本 ∇Cx 的值约等，即

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{18}$$

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}, \tag{19}$$

可以通过计算样本大小为 m（< 全部样本） 的小批量数据的梯度，来估计整体的梯度。为了明确地将其和神经网络联系起来，假设 wk 和 bl 分别为神经网络的权值和偏差。然后随机梯度下降通过随机选取一个训练样本作为输入，然后通过下式训练：

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{20}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{21}$$

其中两个求和符号是在随机选取的小批量的训练样本上进行的。然后使用另一组随机的训练样本去训练，以此类推，直到我们用完了所有的训练输入，这样被称为一个训练迭代期（完成一次下降）。然后

我们会开始一个新的训练迭代期（根据新的位置判断新的下降路径）。

// PS：上面介绍的方法针对代价函数的策略是对所有误差求平均数，但是在面对样本数量不定的情况可以采用求和（例如实时情况），即省去 1/n 或 1/m 。不过这会导致学习速率的改变，面对不同工作应当考虑不同的解决方案。

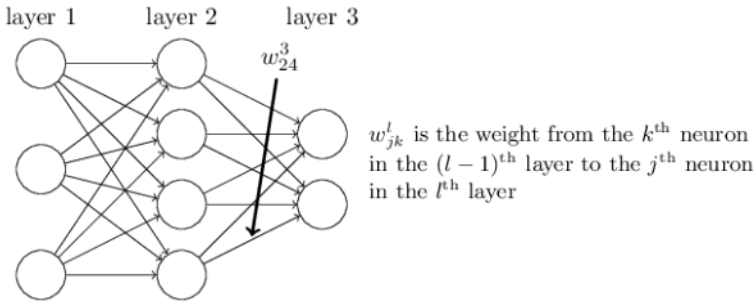
反向传播算法

相关定义

$$w^l_{jk}$$

定义 $(l-1)$ 层第 k 个神经元到第 l 层第 j 个神经元之间的权重。

如图展示了第2层第4个神经元到第3层第2个神经元之间的权重：

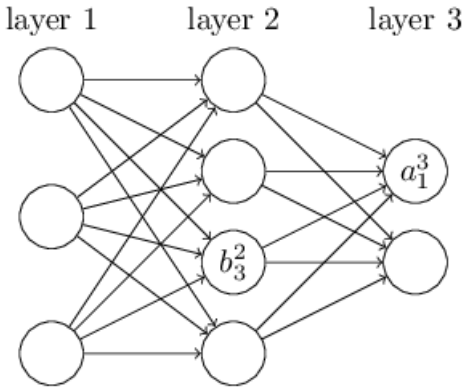


$$b^l_j$$

表示第 l 层第 j 个神经元的偏差

$$a^l_j$$

表示第 l 层第 j 个神经元的激活值



有了这些定义，就可以表示出激活值 a 的计算公式。

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{23}$$

引入向量化函数。（即将向量内单个元素的运算宏观到向量之间的运算）

$$a^l = \sigma(w^l a^{l-1} + b^l). \tag{25}$$

中间量（带权输入）

$$z^l = w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

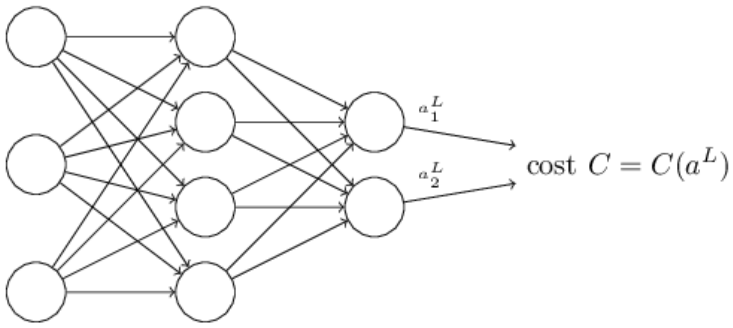
两个假设

假设代价函数可以写成对于每一个训练样本 x 的代价值 C_x 的均值。即

$$C = \frac{1}{n}\sum_x C_x$$

因为，随机梯度中将训练集划分为小份，需要有这个等式才能保证反向传播计算得到单个训练样本偏导数与最终偏导数的正确联系。

假设代价函数可以写成神经网络的输出函数。即



二次代价函数对于单个样本：

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2, \qquad (27)$$

因为 y 作为理想值是一个固定值，那么在等式中的变量就仅剩 a。因此将代价函数 C 看成输出激活值 a 的函数的假设也是合理的。

Hadamard 乘积

假设有两个同纬度的向量 s 和 t，那么 $s \odot t$ 就可以用来表示这两个向量之间按元素乘积，即 $(s \odot t)_j = s_j t_j$,举个例子：

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \qquad (28)$$

算法

中间量定义

$$\delta_j^l$$

表示 l 层第 j 个神经元的误差。

定义误差

对于每一个神经元，从输出结果上来看，每次由 w 权值与 d 偏差经计算产生的结果式为：

σ(w x+b) 但是因为如果考虑进 σ 函数的话反向传播的表达会变得复杂，因此通常仅考虑 w x+b 即前文定义的带权输入 z。

因为当前的待优化输入（ w，d ）必然带有误差，其对 z 的影响就是 Δz。（意思就是将 z 中存在的 Δz 视为输入中存在的误差）

又因为之前的假设前提，可以将 C 视为激活值的输出（也就是整个网络的最终输出）。因此该神经元带来的误差 Δz 给整个系统带来的影响即为：

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \qquad (29)$$

这里要理解的一点是，所谓反向传播，就是从结果出发将传播过程中出现的误差一步步整合最终传到向梯度参数（C 对于 w 与 d 的偏导）。所以 Δz 它不是增量，不是我们目标要去将其添加进 z 来使 z 更加正确的内容，而是一个存在于原本的 z 中，我们需要去剔除的量。

明白了误差的定义以后，就可以去寻找其与梯度参数的关联了。毕竟反向传播要解决的问题是要求得：

$$\frac{\partial C}{\partial b_l}\frac{\partial C}{\partial w_k}$$

核心方程

首先要得到输出层的误差。从误差的表达式入手，

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{29}$$

应用链式法则，以激活值偏导数的形式表示 (29)

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}$$

因为一个输出神经元的激活值 a 只取决于其本身输入的带权输入 z，所以这里当 k != j 时的其他量没有意义。因此可以简化。

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

又

$$a_j^L = \sigma(z_j^L)$$

所以有，

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

由此得到第一条方程：

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{BP1}$$

将其改作矩阵形式

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

下面要得到隐藏层前一层误差与后一层误差的关系表达式。
同样是链式法则：

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}\end{aligned}$$

由 z 的定义可以得到式子:

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

作微分:

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

将其带入第一条式子：

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

这就是用分式写成的第二条方程：

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l), \qquad \text{(BP2)}$$

第二条方程其实正是反转传播的体现，其输入量为 l+1 层的误差，在乘上 w 的转置后（w 是前一层输入值 a 的系数）的结果就是上一层到这一层误差传播的速率，再做Hadamard积就可以得到上一层的误差的数值（雾）。

通过上面两条方程我们可以推导出任意一层的误差。

接下来就要将之前所作的努力与梯度系数（目标）相联系起来了。

由定义：z = a·w + b，误差是 z 与 C 的微分。目标是 b，w 与 C 的微分。

与 b 是常数关系。

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \qquad \text{(BP3)}$$

与 w 是线性关系，系数为 a

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \qquad \text{(BP4)}$$

至此四条方程全部得到：

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \qquad \text{(BP4)}$$

// 回过头来看还真是能感受到数学之美。。。

接下来设计程序的话，输入值为

本次运行的到的输出层激活值 a

a_j^L

输出层各神经的权重 w 与 偏差 b

z^L

就能得到梯度系数。算法是可以实现的。

步骤

- 输入x：为输入层设置对应的激活值a^1。
- 向前传播：对所有的$l = 2, 3, 4..., L$，计算$z^l = w^l a^{l-1} + b^l$和$a^l = \sigma(z^l)$
- 输出误差：计算向量$\delta^l = \nabla_a C \odot \sigma'(z^l)$
- ★ 误差逆传播：对所有的$l = L-1, L-2, \ldots, 2$计算$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \text{ and } \frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

- 输出：代价函数的梯度：

实现

```
"""
network.py
~~~~~~~~~~

实现了简单的神经网络。
"""
#### Libraries
# 标准库
import random
# 第三方库
import numpy as np
from xlwings import xrange

class Network(object):

    def __init__(self, sizes):
        """
        list对象sizes包含了各层中神经元的数量，例如想要创建第一二三层分别为2，3，1个神经元的网络，可以这样创建类对象
        net = Network([2,3,1])
        """
        # 层数
        self.num_layers = len(sizes)
        # 各层神经元数量
        self.sizes = sizes
        # 偏差 b
        # 除去输入层，每一层的神经元生成偏差随机数（正态分布）
        # y 代表每一层神经元数量，对应每一个神经元生成偏差
        self.biases = [np.random.randn(y, 1)
                       for y in sizes[1:]]
        # 权重 w
        # 对每一个输入生成权重，上一层的每一个神经元对该层每一个神经元都会产生输入
        # x 代表上一层神经元数量（除去输出层），y 代表本层神经元数量（除去输入层）
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
    def feedforward(self, a):
        """在输入为 a 的情况下，输出神经网络结果"""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """
```

```
    train_data 一个 (x, y) 的元组的列表，表示训练集输入与其期望的输出。
    epochs 迭代期数量。
    mini_batch_size 采样时小批量数据大小
    eta 学习速率，即 η
    可选参数: test_data 测试集
    完成随机梯度下降法。
    """
    if test_data:
        n_test = len(test_data)
    n = len(training_data)
    # 进入学习迭代
    for j in xrange(epochs):
        # 对训练集进行随机排序
        random.shuffle(training_data)
        # 划分训练集，进行随机迭代，从 0 到 n (训练集长度) ，步长 mini_batch_size
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        # 进入每个训练迭代期，直至用完了所有训练输入
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print ("Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test))
        else:
            print ("Epoch {0} complete".format(j))
def update_mini_batch(self, mini_batch, eta):
    """
    mini_batch 随机梯度步长
    eta 学习速率， η
    """
    # b 对于代价函数 C 的微分，∂C/∂b
    # 目前是用 0 填充的 b 的矩阵
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    # w 对于代价函数 C 的微分，∂C/∂w
    # 目前是用 0 填充的 w 的矩阵
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        # Δ(∂C/∂b) 与 Δ(∂C/∂w)
        # backprop 计算代价函数的梯度
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        # 累加
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    # 迭代权重与偏差，公式(20)(21)
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x.  ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    # b 对于代价函数 C 的微分，∂C/∂b 目标值
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    # w 对于代价函数 C 的微分，∂C/∂w 目标值
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # 向前传播
    activation = x
    # 保存每一层 a 激活值
    activations = [x]
```

```
        # 保存每一层 z 带权输入
        zs = []
        for b, w in zip(self.biases, self.weights):
            # 计算带权输入 z
            z = np.dot(w, activation)+b
            zs.append(z)
            # 计算激活值 a
            activation = sigmoid(z)
            activations.append(activation)
        # 误差逆传播
        # 公式 1
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in xrange(2, self.num_layers):
            # 倒序获取 z
            z = zs[-l]
            # 求 σ'(z)
            sp = sigmoid_prime(z)
            # 公式 2
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            # 公式 3 与 4
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
    def evaluate(self, test_data):
        """
        返回正确识别的数量。
        进行了一次向前传播并保存结果与测试机结果比对
        """
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
    def cost_derivative(self, output_activations, y):
        """代价函数的导函数"""
        return (output_activations-y)
def sigmoid(z):
    """sigmoid 函数"""
    return 1.0/(1.0+np.exp(-z))


def sigmoid_prime(z):
    """sigmoid 函数的导函数"""
    return sigmoid(z)*(1-sigmoid(z))

"""
mnist_loader
~~~~~~~~~~~~

加载 MNIST 图像数据.
"""
#### Libraries
# Standard library
import pickle
import gzip
# Third-party libraries
import numpy as np
def load_data():
    """
    从硬盘中载入数据集
    """
    f = gzip.open('../data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding='bytes')
    f.close()
    return (training_data, validation_data, test_data)
def load_data_wrapper():
```

```
    """
    将装载的数据集打包
        将灰度图片序列化
        重新打包
    """
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)
def vectorized_result(j):
    """返回一个10维的单位向量，第j项是1.0"""
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

'''
Author: vanot313
Date: 2021-02-08 22:14:07
LastEditTime: 2021-02-08 22:14:44
LastEditors: vanot313
Description: 测试文件
FilePath: \neural-networks-and-deep-learning\src\test.py
'''
from src.network import *
from src.mnist_loader import *
tr, va, te = load_data_wrapper()
tr = list(tr)
va = list(va)
te = list(te)
net = Network([784, 30, 10])
net.SGD(tr, 30, 3, 3.0, te)
```

末

这次原本的计划是要完成对 RNN 循环神经网络的学习的。但是翻开 RNN 的博客发现如读无字天书，基本除了看得懂是中文别的都看不懂，因此需要重新补一下神经网络知识。这次因为项目也赶进度，所以博客写的比较潦草，但是也完整地从底层实现了一遍神经网络，收获还不错。