反向传播算法

Zebediah

2024 年 5 月 9 日

本文参考 Michael Nielsen 的 Neural Network and Deep Learning 第 2 章.

目录

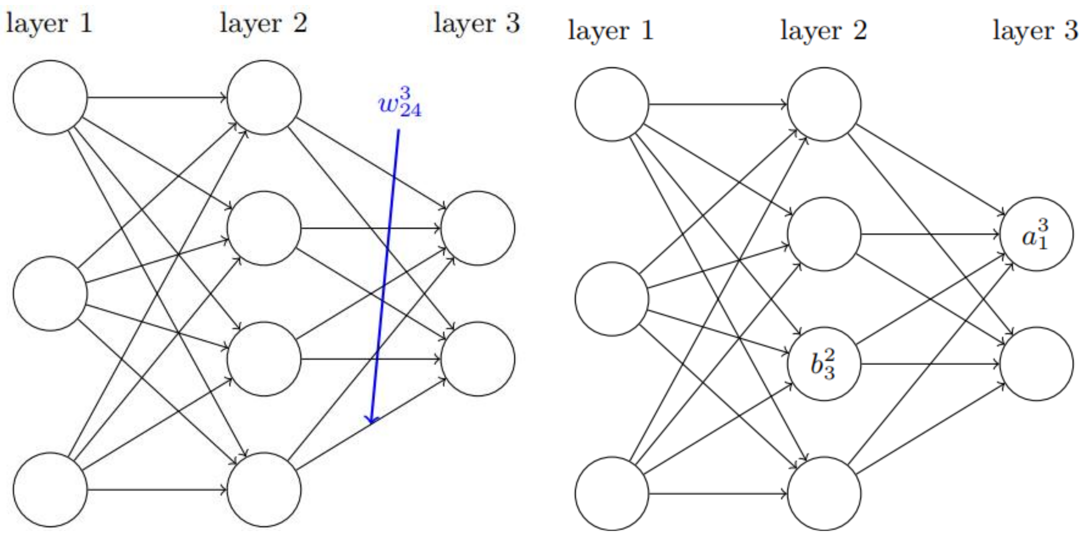
1 引入

我们之前已经学习了使用梯度下降算法学习的权重和偏置, 但并没有讨论如何计算代价函数的梯度, 反向传播算法是一种计算梯度的快速算法.

1.1 一些记号

首先引入一些常用记号, 图 1 给出了这些记号在神经网络中的示例.

- w^l_{jk}: 从 $(l-1)^{\text{th}}$ 层的第 k^{th} 个神经元到 l^{th} 层的第 j^{th} 个神经元的连接上的权重;
- b^l_j: 在 l^{th} 层第 j^{th} 个神经元的偏置;
- a^l_j: 在 l^{th} 层第 j^{th} 个神经元的激活值.



(a) w^l_{jk} (b) b^l_j, a^l_j

图 1: 神经网络中的一些常用记号

在引入上述记号后, 我们可以得到如下关系:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{1.1}$$

其中 $\sigma(\cdot)$ 为 sigmoid 函数. 事实上, 还能将 (1.1) 写成更为简便的向量形式, 即

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \overset{\text{def}}{=} \sigma\left(z^l\right), \tag{1.2}$$

这里 $z^l = w^l a^{l-1} + b^l$ 为一个中间量, 称其为 l 层神经元的带权输入.

1.2 关于代价函数的两个假设

我们使用的代价函数是二次代价函数

$$C = \frac{1}{2n}\sum_x \left\|y(x) - a^L(x)\right\|^2, \tag{1.3}$$

其中 n 是训练样本的总数, $y = y(x)$ 是对应的目标输出, L 表示网络的层数, $a^L = a^L(x)$ 是当输入是 x 时的网络输出的激活值向量. 为了应用反向传播, 我们需要给出以下两个假设:

(a) 代价函数可以写成每个训练样本 x 上代价函数的均值, 即 $C = \frac{1}{n}\sum_x C_x$, 其中 $C_x = \frac{1}{2}\left\|y(x) - a^L(x)\right\|^2$;

(b) 代价可以写成关于神经网络输出的函数, 图 2 给出了一个示例.

基于假设 (a), 在反向传播中, 可以首先对一个独立的训练样本计算 $\partial C_x/\partial w$ 和 $\partial C_x/\partial b$, 然后通过在所有训练样本上进行平均化得到 $\partial C/\partial w$ 和 $\partial C/\partial b$. 对于假设 (b), 代价函数 C 依赖于神经网络输出是显然的, 而同时 C 也是与目标输出 y 有关的函数, 但由于输入样本 x 是固定的, 因此 y 也是一个固定的参数, 故可将 C 直接写成关于神经网络输出的函数.

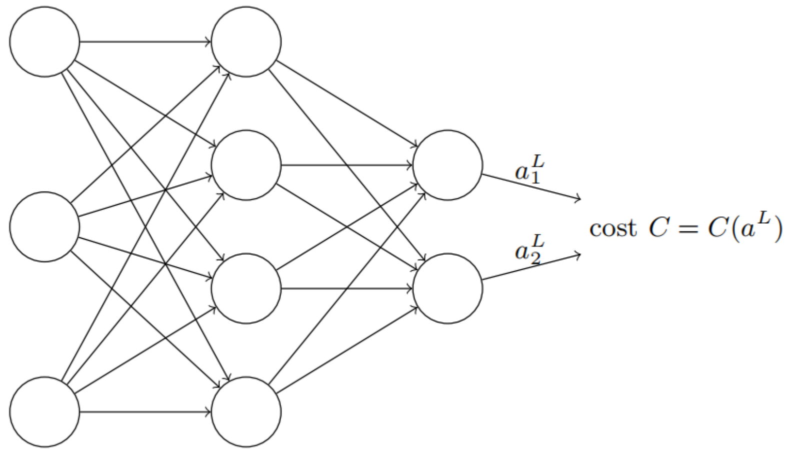


图 2: 代价可以写成神经网络输出的函数

2 反向传播的四个基本方程及证明

如图 3 所示, 一个调皮鬼在 l 层的第 j^{th} 个神经元上, 若给予一个很小的扰动 Δz_j^l 在神经元的带权输入上, 则神经元输出由 $\sigma\left(z_j^l\right)$ 变成 $\sigma\left(z_j^l+\Delta z_j^l\right)$, 这个变化会向网络后面的层进行传播, 最终导致整个代价产生 $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$ 的改变.

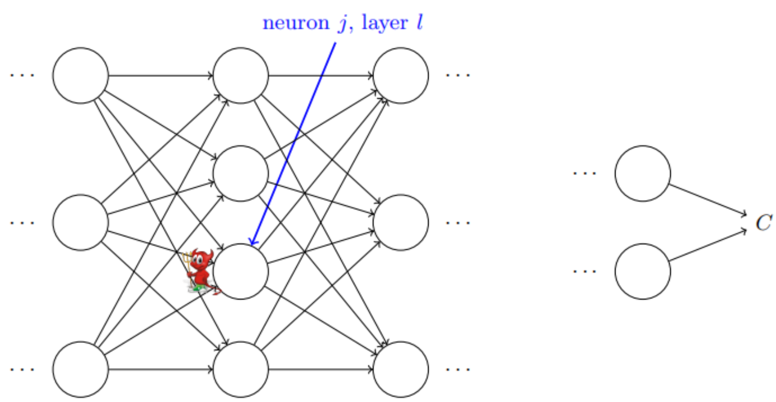


图 3: 在神经元的带权输入上给予扰动 Δz_j^l

我们的目标是极小化代价函数 C. 假设 $\frac{\partial C}{\partial z_j^l}$ 有一个很大的值 (或正或负), 那么我们可以通过选择与 $\frac{\partial C}{\partial z_j^l}$ 相反符号的 Δz_j^l 来降低代价. 如果 $\frac{\partial C}{\partial z_j^l}$ 接近 0 , 则不能通过扰动带权输入 z_j^l 来降低太多代价, 但这已经非常接近最优了. 因此, 我们发现 $\frac{\partial C}{\partial z_j^l}$ 是神经元的误差的一种度量, 故可定义 l 层的第 j^{th} 个神经元上的误差为:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{2.1}$$

2.1 四个基本方程

有了误差的定义后, 我们可以推导出如下四个基本方程:

$$\delta^L = \nabla_a C \odot \sigma'\left(z^L\right); \tag{BP1}$$

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right); \tag{BP2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l; \tag{BP3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{BP4}$$

四个基本方程将在 2.2 节中进行证明, 在此我们不妨先解释一下四个基本方程的直观理解:

BP1: 该方程为输出层误差的方程, 它实际上是等式 $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'\left(z_j^L\right)$ 的矩阵形式. 对于 δ_j^L, 假如 C 不太依赖一个特定的输出神经元 j, 那么 δ_j^L 就会很小, 这也是我们想要的效果. $\sigma'\left(z_j^L\right)$ 刻画了在 z_j^L 处激活函数 σ 变化的速度.

BP2: 该方程使用了下一层的误差 δ^{l+1} 来表示当前层的误差 δ^l. 假设我们知道 $l+1^{\text{th}}$ 层的误差 δ^{l+1}, 当我们应用转置的权重矩阵 $\left(w^{l+1}\right)^T$, 可以凭直觉地把它看作是在沿着网络反向移动误差, 然后与 $\sigma'\left(z^l\right)$ 做 Hadamard 积, 可以理解为让误差通过 l 层的激活函数反向传递回来并给出在第 l 层的带权输入的误差 δ.

BP3: 该方程给出了代价函数关于偏置的变化率, 且变化率就等于误差 δ_j^l.

BP4: 该方程给出了代价函数关于权重的变化率, 较 (BP3) 多了一项在 $l-1^{\text{th}}$ 层的激活值 a_k^{l-1}.

2.2 四个基本方程的证明

(BP1): $\delta^L = \nabla_a C \odot \sigma'\left(z^L\right)$.

证明. 注意到第 k^{th} 个神经元的输出激活值 a_k^L 只依赖于当 $k=j$ 时第 j^{th} 个神经元的输入权重 z_j^L, 即当 $k \neq j$ 时, $\partial a_k^L / \partial z_j^L = 0$. 利用链式法则有

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'\left(z_j^L\right),$$

将其写成向量形式即得 (BP1). □

(BP2): $\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right)$.

证明. 注意到

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

于是 $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'\left(z_j^l\right)$. 利用链式法则有

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1} = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'\left(z_j^l\right),$$

将其写成向量形式即得 (BP2). □

(BP3) 和 (BP4): $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$ 和 $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$.

证明. 利用链式法则有

$$\frac{\partial C}{\partial b_j^l}=\frac{\partial C}{\partial z_j^l}\cdot\frac{\partial z_j^l}{\partial b_j^l}=\delta_j^l\cdot\frac{\partial\left(w_{jk}^l a_k^{l-1}+b_j^l\right)}{\partial b_j^l}=\delta_j^l,$$
$$\frac{\partial C}{\partial w_{jk}^l}=\frac{\partial C}{\partial z_j^l}\cdot\frac{\partial z_j^l}{\partial w_{jk}^l}=\delta_j^l\cdot\frac{\partial\left(w_{jk}^l a_k^{l-1}+b_j^l\right)}{\partial w_{jk}^l}=a_k^{l-1}\delta_j^l,$$

(BP3) 和 (BP4) 得证. □

3 反向传播算法

3.1 算法步骤

反向传播算法给出了一种计算代价函数梯度的方法, 算法描述如下所示:

算法 1 BP

1: **输入训练样本的集合**
2: **对每个训练样本 x:** 设置对应的输入激活 $a^{x,1}$, 并执行下面的步骤:
- **前向传播:** 对每个 $l=2,3,\cdots,L$ 计算 $z^{x,l}=w^l a^{x,l-1}+b^l$ 和 $a^{x,l}=\sigma\left(z^{x,l}\right)$;
- **输出误差 $\delta^{x,L}$:** 计算向量 $\delta^{x,L}=\nabla_a C_x\cdot\sigma'\left(z^{x,L}\right)$;
- **反向传播误差:** 对每个 $l=L-1,L-2,\ldots,2$ 计算 $\delta^{x,l}=\left(\left(w^{l+1}\right)^T\delta^{x,l+1}\right)\odot\sigma'\left(z^{x,l}\right)$;

3: **梯度下降:** 对每个 $l=L-1,L-2,\cdots,2$ 根据 $w^l\to w^l-\frac{\eta}{m}\sum_x\delta^{x,l}\left(a^{x,l-1}\right)^T$ 和 $b^l\to b^l-\frac{\eta}{m}\sum_x\delta^{x,l}$ 更新权重和偏置.

3.2 算法的优点

对于梯度的计算我们有一种更简单的表示方式. 事实上, 利用偏导数的定义有

$$\frac{\partial C}{\partial w_j}\approx\frac{C\left(w+\epsilon e_j\right)-C(w)}{\epsilon},\tag{3.1}$$

其中 $\epsilon>0$ 是一个很小的正数, e_j 是在第 j 个方向上的单位向量. 同样地也可以计算 $\partial C/\partial b$. 这个方法看似简单易懂, 但算法的运行速度非常慢. 事实上, 在 (3.1) 中, 对于不同的权重 w_j, 我们均需要计算 $C\left(w+\epsilon e_j\right)$ 来得到 $\partial C/\partial w_j$, 算法的复杂度非常高. 然而在反向传播算法中, 利用链式法则我们可以同时计算所有的偏导数, 总的计算代价大约为两倍前向传播, 这极大地加快了算法速度.

3.3 二次代价函数的不足

这里举一个简单的例子. 设二次代价函数为

$$C=\frac{(y-a)^2}{2},$$

其中 a 是神经元的输出. 在此不妨使用训练输入和目标输出为 $x=1, y=0$ 的样本, 利用链式法则不难得到:

$$\frac{\partial C}{\partial w}=(a-y)\sigma'(z)x=a\sigma'(z), \quad \frac{\partial C}{\partial b}=(a-y)\sigma'(z)=a\sigma'(z). \tag{3.2}$$

sigmoid 函数图像如图 4 所示. 我们发现, 当输出接近 1 时, 曲线非常平缓, 即 $\sigma'(x)$ 非常小, 这就会使得 (3.2) 计算出来的结果也很小, 从而导致学习变得缓慢. 为了消除 $\sigma'(x)$ 带来的影响, 一个很好的方法就是使用交叉熵代价函数, 这将在书中第三章深入讨论.

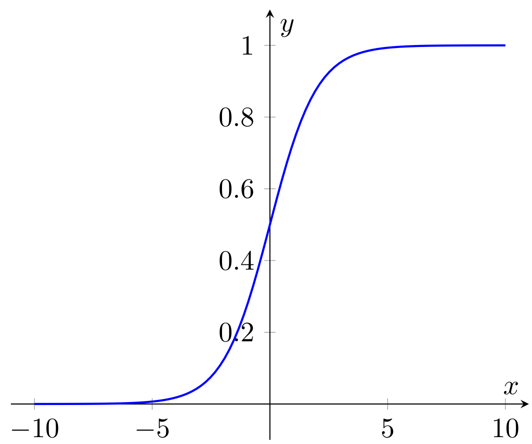


图 4: sigmoid 函数

4 代码

最后给出反向传播算法的代码:

```python
import random
import numpy as np

class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes) # sizes 为每层的神经元数量
        self.sizes = sizes
        # 初始化权重和偏置, sizes[1:] 是一个列表切片(从第二个元素开始到列表末尾).
        # 对于列表中的每个元素 y, 调用 np.random.randn(y, 1) 生成一个随机的 y 行 1 列的数组
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a): # 正向传播, 返回的是被激活函数作用后的激活值
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

```

```python
    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None): # 梯度下降算法
        training_data = list(training_data) # 将输入的训练数据转换成列表, n 表示列表长度
        n = len(training_data)
        if test_data: # 如果测试集非空, 则执行下面代码
            test_data = list(test_data)
            n_test = len(test_data)
        for j in range(epochs): # 迭代
            random.shuffle(training_data) # random.shuffle 作用是随机打乱列表中元素的顺序
            mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)] # 每 mini_batch_size 个样本为一批数据
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta) # 使用这样一批数据更新参数(权重和偏置)
            if test_data:
                print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test))
            else:
                print("Epoch {} complete".format(j))

    def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases] # 初始化权重和偏置
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y) # 反向传播算法
            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)]
        self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)] # 更新权重和偏置
        self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases] # 初始化权重和偏置
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # 前向传播
        activation = x # x 表示输入值
        activations = [x] # 用列表储存 x 的值
        zs = [] # z 表示未被激活的加权值, 并用列表 zs 储存
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b # 计算隐藏层每个神经元处的输入值
            zs.append(z) # 用列表 zs 储存
            activation = sigmoid(z) # 激活 z, 并将其储存在列表 activations 中
            activations.append(activation)
        # 反向传播
        # 下面三个方程是反向传播的基本方程, 用于计算误差, 权重和偏置
        delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1]) # 这里计算的是输出层处反向传播的各个参数
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

```python
        for l in range(2, self.num_layers): # 反向传播，遍历前面的 layers
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp # 反向传播的基本方程，建立了相邻层误差的联系
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, test_data):
        # 这里使用了 np.argmax() 函数来找到具有最高概率的预测结果的类别标签
        # 然后将预测结果与实际标签 y 进行比较，形成由 (预测结果，实际标签) 组成的元组列表 test_results
        # 最后统计预测结果与实际标签相同的数量，并将其作为函数的返回值
        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): # 定义代价函数的导数
        return (output_activations-y)


def sigmoid(z): # 定义激活函数
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z): # 定义激活函数的导数
    return sigmoid(z)*(1-sigmoid(z))

## 识别手写数字示例
import mnist_loader # 这里使用了 mnist_loader 用于输入数据，在此把它视作黑盒即可
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
training_data = list(training_data)
net = Network([784, 30, 10]) # 这里使用一个浅层神经网络进行训练，三个数分别对应输入层，隐藏层，输出层的神经元数量
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```