**第5章**

# 反向传播

|  |  |
| --- | --- |
|  | 本章内容 |
|  | ◎ 反向传播原理  ◎ 反向传播代码实现 |

## 5.1 反向传播原理

在第三章的内容中，我们介绍过神经网络的发展历史，由于感知机无法解决复杂的任务，之后提出了多层网络的结构，但是多层结构却并没有有效的办法低复杂度，直到1974年Paul Werbos提出了一种学习神经网络的算法back propagation反向传播，该算法能在每一轮迭代的过程中高效的计算梯度，最后采用梯度下降法即可解决降低复杂度的问题。本章将会详细介绍反向传播算法， 并讲解其中的原理，最后以代码的形式呈现给读者。

首先来看一个简单的神经网络如图5.1-1，该网络包含三层，输入层，隐藏层与输出层，输入层包含两个节点与一个偏执项，隐藏层也是两个节点与一个偏执项，最后输出层输出两个值。每一条线上的w指的是对应节点的权重，激活函数默认采用sigmoid函数。

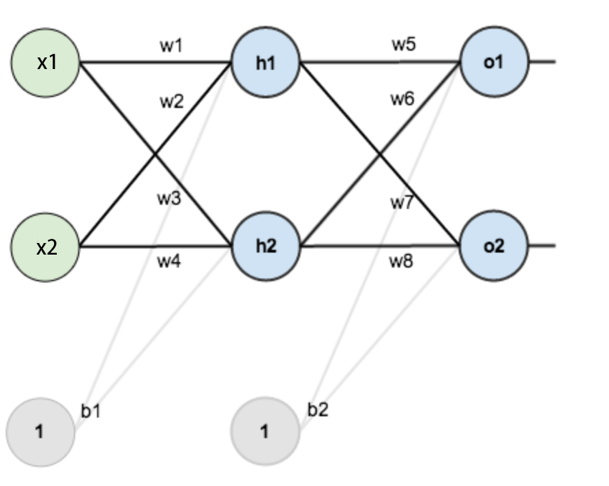


图5.1-1 神经网络

在讲解反向传播之前，先给读者推导出前向传播结果，这里定义几个标识符，其中表示神经元的加权输出，表示sigmoid激活函数，表示真实结果，由图可知：

接下来计算误差，因为这里有两个输出节点，所以总误差是两个节点的误差的和。

误差计算完成后，即可针对每一个节点计算梯度，这里以为例，来更新隐藏层的权重，在计算偏导之前，我们需要知道一个公式，sigmoid函数的导数为

那么根据求导的链式法则有

根据梯度下降法即可更新

同理，的更新方法也是如上所示，但是对于输入层的权重，计算方法有一定的区别，这里以为例，因为的偏导数是由决定的，而的值又是由与共同决定的，所以根据链式求导法则有

根据梯度下降法更新

对于来说，其更新方法与相同，这里不再赘述。

到这里为止，就给读者介绍完了反向传播算法，其实反向传播的关键点即在于通过链式法则求偏导，对于偏导受多个值影响的情况，需要分别求出不同值的偏导并求和，最后根据梯度下降法来更新权重。下一节，我们会从代码的角度来为读者演示反向传播。

## 5.2 反向传播代码实现

上一节介绍了反向传播的原理，并推导了一个简单的神经网络的反向传播过程，本节将会带领读者从代码层面来实现反向传播。

本节将会采用keras自带的一份数据，该数据样本包含 1970 年代的在波士顿郊区不同位置的房屋信息，总共有 13 种房屋属性，目标值是对应房屋的价格，因此也是一个回归问题，目标是让损失函数最小化，这里我们设定一个三层的神经网络，包括输入层，隐藏层，输出层，因为数据有13种属性，因此输入层设定13个节点，隐藏层设定64个节点，这个参数是一个超参数，读者可自行调整，输出层一个节点，即输出值房价，接下来就一起来看其中的代码。

首先，导入数据，并设定一个随机种子，方便后续生成的随机数都是一样的。

|  |  |
| --- | --- |
| 1 | import numpy as np |
| 2 | from keras.datasets import boston\_housing |
| 3 | (x\_train,y\_train), (x\_test, y\_test) = boston\_housing.load\_data() |
| 4 | np.random.seed(2) |

然后定义神经网络中的参数，这里采用随机初始化的方式，并存储在一个字典里，方便后续使用。

|  |  |
| --- | --- |
| 1 | def initialize\_parameters(n\_x, n\_h, n\_y): |
| 2 | W1 = np.random.randn(n\_h, n\_x) |
| 3 | b1 = np.zeros((n\_h, 1)) |
| 4 | W2 = np.random.randn(n\_y, n\_h) |
| 5 | b2 = np.zeros((n\_y, 1)) |
| 6 |  |
| 7 | parameters = {"W1": W1,  "b1": b1,  "W2": W2,  "b2": b2} |
| 8 | return parameters |

接下来计算前向传播的值，注意此处采用的激活函数是tanh，为了让输出层的结构和隐藏层相同，多定义了一个A2， A2与Z2的值是相同的。

|  |  |
| --- | --- |
| 1 | def forward\_propagation(X, parameters): |
| 2 | W1 = parameters["W1"] |
| 3 | b1 = parameters["b1"] |
| 4 | W2 = parameters["W2"] |
| 5 | b2 = parameters["b2"] |
| 6 |  |
| 7 | Z1 = np.dot(W1, X.T) + b1 |
| 8 | A1 = np.tanh(Z1) |
| 9 | Z2 = np.dot(W2, A1) + b2 |
| 10 | A2 = Z2 |
| 11 | cache = {"Z1": Z1,  "A1": A1,  "Z2": Z2,  "A2": A2} |
| 12 | return A2, cache |

前向传播计算完成后即可计算损失函数的值，这里采用的是均方差损失函数。

|  |  |
| --- | --- |
| 1 | def compute\_cost(A2, Y): |
| 2 | cost = np.sum(np.square(Y - A2)) / 2 / Y.shape[0] |
| 3 | return cost |

接下来就是最为关键的算法反向传播，反向传播的计算方法上一节我们已经讲述过，这里不再赘述，下文的代码以di的形式表示i的偏导数。

|  |  |
| --- | --- |
| 1 | def backward\_propagation(parameters, cache, X, Y): |
| 2 | m = X.shape[0] |
| 3 | W1 = parameters["W1"] |
| 4 | W2 = parameters["W2"] |
| 5 | A1 = cache['A1'] |
| 6 | A2 = cache['A2'] |
| 7 |  |
| 8 | dZ2 = A2 - Y |
| 9 | dW2 = np.dot(dZ2, A1.T) / m |
| 10 | db2 = np.mean(dZ2) |
| 11 | dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2)) |
| 12 | dW1 = np.dot(dZ1, X) / m |
| 13 | db1 = np.mean(dZ1) |
| 14 | grads = {"dW1": dW1,  "db1": db1,  "dW2": dW2,  "db2": db2} |
| 15 | return grads |

偏导数计算完成后，即可采用梯度下降法来更新梯度。

|  |  |
| --- | --- |
| 1 | def update\_parameters(parameters, grads, learning\_rate=0.01): |
| 2 | W1 = parameters['W1'] |
| 3 | b1 = parameters["b1"] |
| 4 | W2 = parameters["W2"] |
| 5 | b2 = parameters["b2"] |
| 6 | dW1 = grads['dW1'] |
| 7 | db1 = grads['db1'] |
| 8 | dW2 = grads['dW2'] |
| 9 | db2 = grads['db2'] |
| 10 |  |
| 11 | W1 = W1 - learning\_rate \* dW1 |
| 12 | b1 = b1 - learning\_rate \* db1 |
| 13 | W2 = W2 - learning\_rate \* dW2 |
| 14 | b2 = b2 - learning\_rate \* db2 |
| 15 | parameters = {"W1": W1,  "b1": b1,  "W2": W2,  "b2": b2} |
| 16 | return parameters |

到此，整个流程即编写完成，接下来，我们把上述的所有过程串起来，并每训练5轮就打印一次损失函数的值。

|  |  |
| --- | --- |
| 1 | def model(X, Y, n\_h, epochs=100, learning\_rate=0.01): |
| 2 | parameters = initialize\_parameters(X.shape[1], n\_h, 1) |
| 3 |  |
| 4 | for epoch in range(0, epochs): |
| 5 | A2, cache = forward\_propagation(X, parameters) |
| 6 | cost = compute\_cost(A2, Y) |
| 7 | grads = backward\_propagation(parameters, cache, X, Y) |
| 8 | parameters=update\_parameters(parameters,grads, learning\_rate) |
| 9 | if epoch % 5 == 0: |
| 10 | print("Cost after iteration %i: %f" % (epoch, cost)) |
| 11 | return parameters |

训练完成后，还需要测试过程，这里编写测试代码，并打印测试集的损失函数的值。

|  |  |
| --- | --- |
| 1 | def predict(parameters, X, Y): |
| 2 | A2, cache = forward\_propagation(X, parameters) |
| 3 | return np.sum(np.square(Y - A2)) / 2 / Y.shape[0] |

一切准备就绪，即可开始调用模型进行训练与预测，此处设定隐藏层的节点数为64，训练轮数为100轮，学习率为0.01。

|  |  |
| --- | --- |
| 1 | parameters = model(x\_train, y\_train, n\_h=64, epochs=100, learning\_rate=0.01) |

输出结果如图5.2-1所示

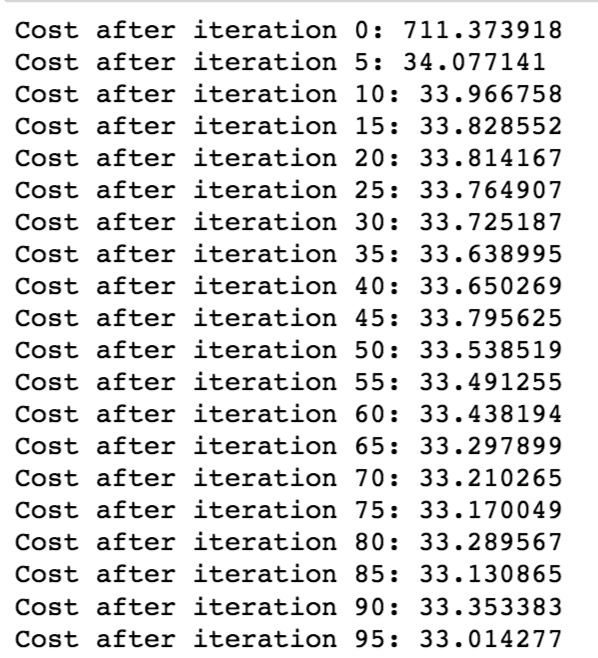


图5.2-1 训练集损失函数结果

最后再执行下预测函数，看下在测试集的效果。

|  |  |
| --- | --- |
| 1 | cost = predict(parameters, x\_test, y\_test) |
| 2 | print(cost) |

最终测试集的损失函数的值如图5.2-2所示为30.70，总体效果还不错，没有出现过拟合的情况，读者也可自行修改网络参数，包括添加神经元的个数，添加网络层数等方法进一步降低损失函数的值。



5.2-2 测试集损失函数结果

反向传播算法在各大神经网络框架中都已封装好，包括keras，但是该算法在神经网络中占据着举足轻重的地位，可以说没有反向传播就没有现如今神经网络带来的巨大价值，建议读者在学习keras框架过程中，详细阅读本章内容，掌握好其中的原理。