# 实验七 神经网络

**实验目的**

掌握神经网络模型的搭建流程以及训练过程

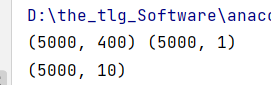
**二、实验内容**

本次实验将利用前馈神经网络来解决手写数字识别问题。实验将使用随机权重初始化方法，设计反向传播算法实现损失函数的非正则化和正则化计算以及梯度计算。最后使用神经网络进行手写数字预测。

**1. 加载数据**

将数据从数据集加载到变量：

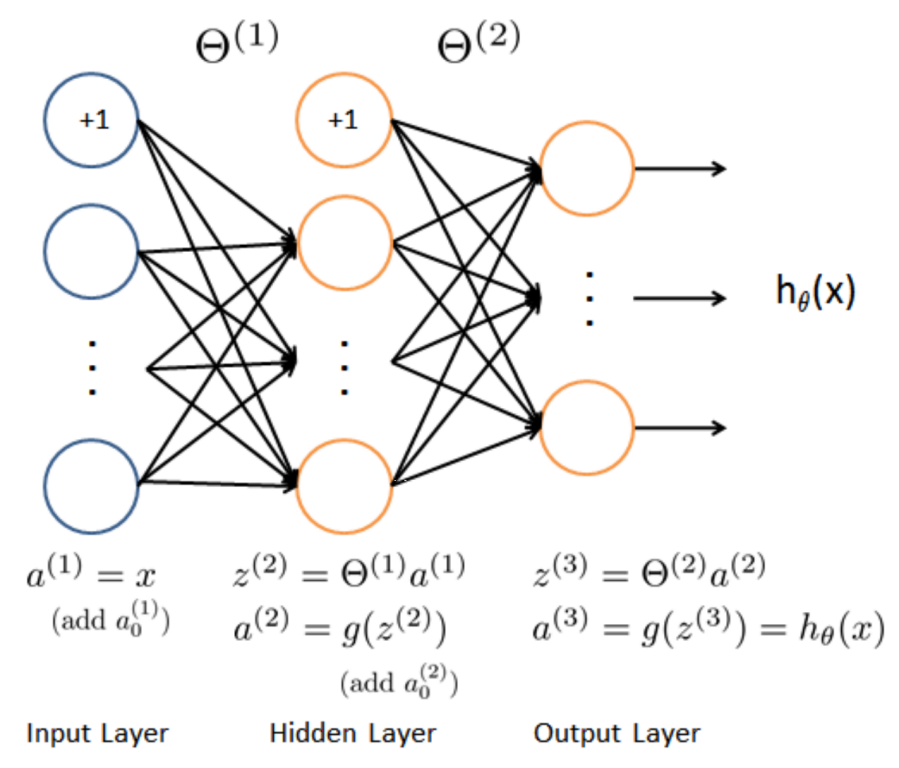
1. data = loadmat('ex3data1.mat')
2. X = data['X']
3. y = data['y']



1. **数据处理**

对标签值y进行独热编码。独热编码可将类别标签n变成长度为k(类别个数)的向量，其中类别标签n的位置标记为1，其余位置标记为0。

1. **模型结构**

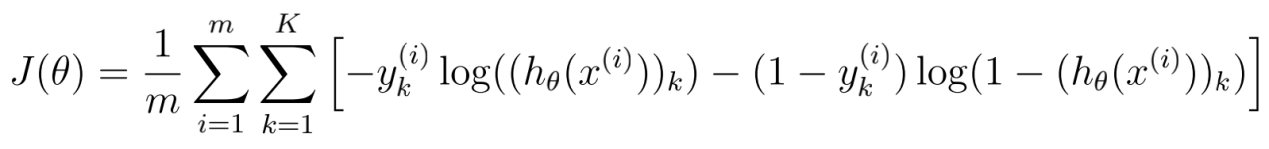


神经网络具有三层，分别是输入层、隐藏层、输出层。输入的图像尺寸为2020，输入层具有400个节点，隐藏层具有25个节点，输出层具有10个节点（有类别个数对应）。

1. **前向传播与损失函数**
2. **def** sigmoid(z):
3. **return** 1 / (1 + np.exp(-z))

6. **def** forward\_propagate(X, theta1, theta2):
7. m = X.shape[0]
8. a1 = np.insert(X, 0, values=np.ones(m), axis=1)
9. z2 = a1 \* theta1.T
10. a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1)
11. z3 = a2 \* theta2.T
12. h = sigmoid(z3)
13. **return** a1, z2, a2, z3, h

损失函数用于评估一组网络参数与预测值之间的差距。

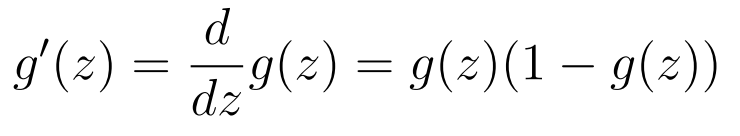


1. **def** cost(params, input\_size, hidden\_size, num\_labels, X, y,learning\_rate):
2. m = X.shape[0]
3. X = np.matrix(X)
4. y = np.matrix(y)
5. # 将参数转换为矩阵形式
6. theta1 = np.matrix(np.reshape(params[:hidden\_size \* (input\_size + 1)], (hidden\_size, (input\_size + 1))))
7. theta2 = np.matrix(np.reshape(params[hidden\_size \* (input\_size + 1):], (num\_labels, (hidden\_size + 1))))
9. # 前向传播
10. a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)
11. # 计算损失
12. J = 0
13. **for** i **in** range(m):
14. first\_term = np.multiply(-y[i, :], np.log(h[i, :]))
15. second\_term = np.multiply((1 - y[i, :]), np.log(1 - h[i, :]))
16. J += np.sum(first\_term - second\_term)
17. J = J / m
19. **return** J

计算神经网络总误差。

1. **反向传播**

反向传播算法更新模型参数，以减少神经网络在训练数据上的误差。首先计算sigmoid函数的梯度。

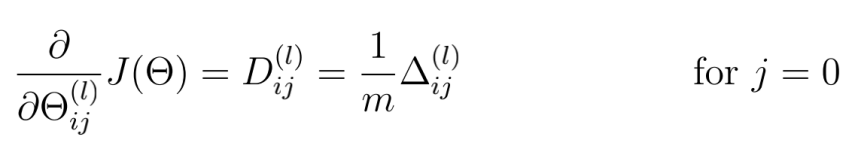


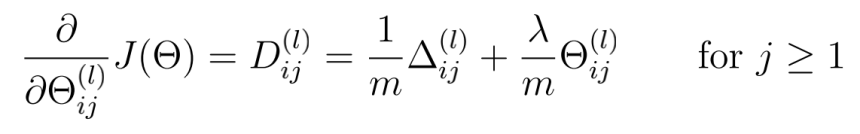
反向传播算法基于损失函数的结果进行梯度计算，接下来拓展损失函数。

1. **def** backprop(params, input\_size, hidden\_size, num\_labels, X, y, learning\_rate):
2. m = X.shape[0]
3. X = np.matrix(X)
4. y = np.matrix(y)
5. # 将参数转换为矩阵形式
6. theta1 = np.matrix(np.reshape(params[:hidden\_size \* (input\_size + 1)], (hidden\_size, (input\_size + 1))))
7. theta2 = np.matrix(np.reshape(params[hidden\_size \* (input\_size + 1):], (num\_labels, (hidden\_size + 1))))
8. # 前向传播
10. # 初始化
11. delta1 = np.zeros(theta1.shape)  # (25, 401)
12. delta2 = np.zeros(theta2.shape)  # (10, 26)
13. # 计算损失
14. a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)
15. # 损失函数正则化项
16. j=cost(params, input\_size, hidden\_size, num\_labels, X, y\_onehot,learning\_rate)
17. # 反向传播
18. **for** t **in** range(m):
19. a1t = a1[t,:]  # (1, 401)
20. z2t = z2[t,:]  # (1, 25)
21. a2t = a2[t,:]  # (1, 26)
22. ht = h[t,:]  # (1, 10)
23. yt = y[t,:]  # (1, 10)
24. d3t = ht - yt  # (1, 10)
25. z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)
26. d2t = np.multiply((theta2.T \* d3t.T).T, sigmoid\_gradient(z2t))  # (1, 26)
27. delta1 = delta1 + (d2t[:,1:]).T \* a1t
28. delta2 = delta2 + d3t.T \* a2t
29. delta1 = delta1 / m
30. delta2 = delta2 / m
31. # 梯度矩阵分解为数组
32. grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))
33. **return** j, grad

测试函数确保返回正确的结果。

还需要在反向传播函数的梯度计算中加入正则化。





1. **def** backpropzengze(params, input\_size, hidden\_size, num\_labels, X, y, learning\_rate):
2. m = X.shape[0]
3. X = np.matrix(X)
4. y = np.matrix(y)
5. # 将参数转换为矩阵形式
6. theta1 = np.matrix(np.reshape(params[:hidden\_size \* (input\_size + 1)], (hidden\_size, (input\_size + 1))))
7. theta2 = np.matrix(np.reshape(params[hidden\_size \* (input\_size + 1):], (num\_labels, (hidden\_size + 1))))
8. # 前向传播
10. # 初始化
11. delta1 = np.zeros(theta1.shape)  # (25, 401)
12. delta2 = np.zeros(theta2.shape)  # (10, 26)
13. # 计算损失
14. a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)
15. # 损失函数正则化项
16. j=cost(params, input\_size, hidden\_size, num\_labels, X, y\_onehot,learning\_rate)
17. # 反向传播
18. **for** t **in** range(m):
19. a1t = a1[t,:]  # (1, 401)
20. z2t = z2[t,:]  # (1, 25)
21. a2t = a2[t,:]  # (1, 26)
22. ht = h[t,:]  # (1, 10)
23. yt = y[t,:]  # (1, 10)
24. d3t = ht - yt  # (1, 10)
25. z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)
26. d2t = np.multiply((theta2.T \* d3t.T).T, sigmoid\_gradient(z2t))  # (1, 26)
27. delta1 = delta1 + (d2t[:,1:]).T \* a1t
28. delta2 = delta2 + d3t.T \* a2t
29. delta1 = delta1 / m
30. delta2 = delta2 / m
31. delta1[:, 1:] = delta1[:, 1:] + (theta1[:, 1:] \* learning\_rate) / m
32. delta2[:, 1:] = delta2[:, 1:] + (theta2[:, 1:] \* learning\_rate) / m
33. # 梯度矩阵分解为数组
34. grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))
35. **return** j, grad

测试函数确保返回正确的结果

1. **训练神经网络**
2. **from** scipy.optimize **import** minimize
3. # 最小化目标函数
4. fmin = minimize(fun=backprop, x0=params, args=(input\_size, hidden\_size, num\_labels, X, y\_onehot, learning\_rate), method='TNC', jac=True, options={'maxiter': 250})
5. **print**(fmin)
6. **预测手写数字**

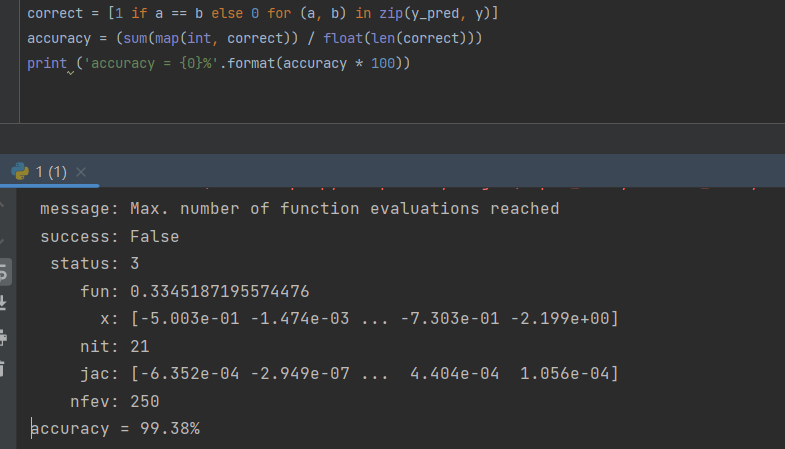
训练过程对迭代次数进行了限制，因为目标函数不可能完全收敛。不过神经网络的总损失值已经降到了0.5以下，表明该算法是有效的。我们使用学习到的参数来预测手写数字。

1. X = np.matrix(X)
2. theta1 = np.matrix(np.reshape(fmin.x[:hidden\_size \* (input\_size + 1)], (hidden\_size, (input\_size + 1))))
3. theta2 = np.matrix(np.reshape(fmin.x[hidden\_size \* (input\_size + 1):], (num\_labels, (hidden\_size + 1))))
4. a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)
5. y\_pred = np.array(np.argmax(h, axis=1) + 1)
6. **print**(y\_pred)
7. **神经网络性能**

correct = [1 if a == b else 0 for (a, b) in zip(y\_pred, y)]

accuracy = (sum(map(int, correct)) / float(len(correct)))

print ('accuracy = {0}%'.format(accuracy \* 100))



1. **实验总结**

实验旨在利用神经网络实现手写数字识别，通过梯度下降算法进行模型训练。通过前向传播计算预测值，并结合损失函数评估预测结果与实际标签之间的误差。在反向传播过程中，根据误差梯度更新神经网络参数，同时考虑到正则化项以避免过拟合问题。最终，通过训练得到的模型对测试集进行预测，并计算准确率评估模型性能。