机器学习练习3 - 神经网络Neural Networks

在这个练习中，我们将处理手写数字数据集，这次使用带有反向传播的前馈神经网络。我们将通过反向传播算法实现神经网络代价函数和梯度计算的未正则化和正则化版本。我们还将实现随机权重初始化，以及使用网络进行预测的方法。

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat
%matplotlib inline

data = loadmat('ex3data1.mat')
data
```

Out[ ]:
```
{'__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun
Oct 16 13:09:09 2011',
 '__version__': '1.0',
 '__globals__': [],
 'X': array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]),
 'y': array([[10],
        [10],
        [10],
        ...,
        [ 9],
        [ 9],
        [ 9]], dtype=uint8)}
```

因为我们以后会需要这些，并且会经常使用，我们提前创建一些有用的变量。

In [ ]:
```
X = data['X']
y = data['y']

X.shape, y.shape
```

Out[ ]:
```
((5000, 400), (5000, 1))
```

我们还需要对y标签进行独热编码。one-hot编码将一个类标签n（在k个类中）转换为长度为k的向量，其中索引n为“热”（1），而其余为零。Scikit-learn有一个内置的实用程序，我们可以直接使用。

In [ ]:
```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False,categories='auto')
y_onehot = encoder.fit_transform(y)
y_onehot.shape
```

Out[ ]:
```
(5000, 10)
```

In [ ]:
```
y[0], y_onehot[0,:]
```

Out[ ]:
```
(array([10], dtype=uint8), array([0., 0., 0., 0., 0., 0., 0., 0., 0.,
1.]))
```

本次练习我们将构建一个神经网络，该网络具有与实例数据大小相匹配的输入层（400+偏差单位），25个神经元（包括偏差单位26个）的隐藏层，以及一个具有10个神经元的输出层，这10个单位与我们对类别标签进行one-hot编码相对应。

我们需要实现的第一部分是一个代价函数，用于评估给定网络参数集合的损失。以下是计算代价所需的函数。 请先补充完整sigmoid函数的定义。

In [ ]:
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

请补充完整前向传播计算的函数

In [ ]:
```
def forward_propagate(X, theta1, theta2):
    m = X.shape[0]

    a1 = np.insert(X, 0, values=np.ones(m), axis=1)
    z2 = a1 * theta1.T
    a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1)
    z3 = a2 * theta2.T
    h = sigmoid(z3)

    return a1, z2, a2, z3, h
```

In [ ]:
```
def cost(params, input_size, hidden_size, num_labels, X, y, learning_rate
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    # reshape the parameter array into parameter matrices for each layer
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)]
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):]

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # compute the cost
    J = 0
    for i in range(m):
        first_term = np.multiply(-y[i,:], np.log(h[i,:]))
        second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
        J += np.sum(first_term - second_term)

    J = J / m

    return J
```

我们之前已经使用过sigmoid函数，所以这不是什么新鲜事。前向传播函数根据当前参数计算每个训练实例的估计值。它的输出形状应该与我们的y的one-hot编码的形状相匹配。

In [ ]:
```
# initial setup
input_size = 400
```

```
hidden_size = 25
num_labels = 10
learning_rate = 1

# randomly initialize a parameter array of the size of the full network's
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labe

m = X.shape[0]
X = np.matrix(X)
y = np.matrix(y)

# unravel the parameter array into parameter matrices for each layer
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (h
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (n

theta1.shape, theta2.shape
```

Out[ ]: ((25, 401), (10, 26))

In [ ]:
```
a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
a1.shape, z2.shape, a2.shape, z3.shape, h.shape
```

Out[ ]: ((5000, 401), (5000, 25), (5000, 26), (5000, 10), (5000, 10))

计算预测值h后，我们可以使用代价函数来计算y和h之间的总误差。

如果sigmoid函数和前向传播函数定义正确，那我们将看到输出6.93354165586454

In [ ]:
```
cost(params, input_size, hidden_size, num_labels, X, y_onehot, learning_r
```

Out[ ]: 6.602849414606072

我们的下一步是在成本函数中添加正则化。它实际上并不像看起来那么复杂，事实上，正则化项只是对我们已经计算的成本的简单添加。这是修改后的成本函数。

In [ ]:
```
def cost(params, input_size, hidden_size, num_labels, X, y, learning_rate
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    # reshape the parameter array into parameter matrices for each layer
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)]
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):]

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # compute the cost
    J = 0
    for i in range(m):
        first_term = np.multiply(-y[i,:], np.log(h[i,:]))
        second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
        J += np.sum(first_term - second_term)

    J = J / m

    # add the cost regularization term
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:]
```

```
    return J
```

In [ ]:
```
cost(params, input_size, hidden_size, num_labels, X, y_onehot, learning_r
```

Out[ ]:
```
6.608279504656432
```

接下来是反向传播算法。反向传播计算参数更新，以减少网络在训练数据上的误差。我们需要做的第一件事是计算我们之前创建的 sigmoid 函数的梯度。

In [ ]:
```
def sigmoid_gradient(z):
    return np.multiply(sigmoid(z), (1 - sigmoid(z)))
```

现在我们已经准备好实现反向传播来计算梯度。

In [ ]:
```
def backprop(params, input_size, hidden_size, num_labels, X, y, learning_
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    # reshape the parameter array into parameter matrices for each layer
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)]
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):]

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # initializations
    J = 0
    delta1 = np.zeros(theta1.shape)  # (25, 401)
    delta2 = np.zeros(theta2.shape)  # (10, 26)

    # compute the cost
    for i in range(m):
        first_term = np.multiply(-y[i,:], np.log(h[i,:]))
        second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
        J += np.sum(first_term - second_term)

    J = J / m

    # add the cost regularization term
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:]

    # perform backpropagation
    for t in range(m):
        a1t = a1[t,:]  # (1, 401)
        z2t = z2[t,:]  # (1, 25)
        a2t = a2[t,:]  # (1, 26)
        ht = h[t,:]  # (1, 10)
        yt = y[t,:]  # (1, 10)

        d3t = ht - yt  # (1, 10)

        z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)
        d2t = np.multiply((theta2.T * d3t.T).T, sigmoid_gradient(z2t))  #

        delta1 = delta1 + (d2t[:,1:]).T * a1t
        delta2 = delta2 + d3t.T * a2t
```

```
    delta1 = delta1 / m
    delta2 = delta2 / m

    # unravel the gradient matrices into a single array
    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

    return J, grad
```

反向传播计算中最困难的部分（除了理解为什么我们要进行所有这些计算之外）就是正确地获取矩阵的尺寸。顺便说一下，如果你发现使用A * B和np.multiply(A, B)时令人困惑，不用担心，其实很多人一开始也不容易理解。基本上前者是矩阵乘法，后者是逐元素乘法（除非A或B是一个标量值，在这种情况下，它并不重要）。。

不管怎么说，让我们测试一下，以确保该函数返回我们期望的结果。

In [ ]:
```
J, grad = backprop(params, input_size, hidden_size, num_labels, X, y_oneh
J, grad.shape
```

Out[ ]:
```
(6.608279504656432, (10285,))
```

我们还要对反向传播函数进行一个修改-在梯度计算中添加正则化。最终的正则化版本如下。

In [ ]:
```
def backprop(params, input_size, hidden_size, num_labels, X, y, learning_
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)

    # reshape the parameter array into parameter matrices for each layer
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)]
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):]

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # initializations
    J = 0
    delta1 = np.zeros(theta1.shape)  # (25, 401)
    delta2 = np.zeros(theta2.shape)  # (10, 26)

    # compute the cost
    for i in range(m):
        first_term = np.multiply(-y[i,:], np.log(h[i,:]))
        second_term = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
        J += np.sum(first_term - second_term)

    J = J / m

    # add the cost regularization term
    J += (float(learning_rate) / (2 * m)) * (np.sum(np.power(theta1[:,1:]

    # perform backpropagation
    for t in range(m):
        a1t = a1[t,:]  # (1, 401)
        z2t = z2[t,:]  # (1, 25)
        a2t = a2[t,:]  # (1, 26)
```

```
        ht = h[t,:]  # (1, 10)
        yt = y[t,:]  # (1, 10)

        d3t = ht - yt  # (1, 10)

        z2t = np.insert(z2t, 0, values=np.ones(1))  # (1, 26)
        d2t = np.multiply((theta2.T * d3t.T).T, sigmoid_gradient(z2t))  #

        delta1 = delta1 + (d2t[:,1:]).T * a1t
        delta2 = delta2 + d3t.T * a2t

    delta1 = delta1 / m
    delta2 = delta2 / m

    # add the gradient regularization term
    delta1[:,1:] = delta1[:,1:] + (theta1[:,1:] * learning_rate) / m
    delta2[:,1:] = delta2[:,1:] + (theta2[:,1:] * learning_rate) / m

    # unravel the gradient matrices into a single array
    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

    return J, grad
```

In [ ]:
```
J, grad = backprop(params, input_size, hidden_size, num_labels, X, y_oneh
J, grad.shape
```

Out[ ]:
```
(6.608279504656432, (10285,))
```

我们终于准备好训练我们的网络，并使用它进行预测。这与前面的逻辑回归练习是类似的。

In [ ]:
```
from scipy.optimize import minimize

# minimize the objective function
fmin = minimize(fun=backprop, x0=params, args=(input_size, hidden_size, n
                method='TNC', jac=True, options={'maxiter': 250})
fmin
```

由于目标函数不太可能完全收敛，我们对迭代次数进行了限制。我们的总成本已经降到0.5以下，这表明算法正在发挥作用。让我们使用优化后的参数，并通过网络进行前向传播，以获得一些预测。

In [ ]:
```
X = np.matrix(X)
theta1 = np.matrix(np.reshape(fmin.x[:hidden_size * (input_size + 1)], (h
theta2 = np.matrix(np.reshape(fmin.x[hidden_size * (input_size + 1):], (n

a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
y_pred = np.array(np.argmax(h, axis=1) + 1)
y_pred
```

Out[ ]:
```
array([[10],
       [10],
       [10],
       ...,
       [ 9],
       [ 9],
       [ 9]], dtype=int64)
```

最后，我们可以计算准确度，以了解我们训练的网络表现如何。

In [ ]:
```
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))
```

```
accuracy = 99.38%
```

我们成功地实现了带有反向传播的基本前馈神经网络，并将其用于对手写数字图像进行分类。 本实验到此结束。