实验 4：神经网络识别手写数字（BP）

介绍

在本实验中，我们将实现神经网络的反向传播算法，并将其应用于手写数字识别任务。

数据集

ex4data1.mat - 手写数字的训练集

1. 神经网络

在上一次实验中，你实现了神经网络的前馈传播，并使用它来预测我们提供权重的手写数字。在本实验中，您将实现反向传播算法来学习神经网络的参数。

我们构建神经网络的整体架构的步骤：

1、对初始数据进行预处理，以满足各函数计算的需求

(1) 对 y 标签进行一次 one-hot 编码。one-hot 编码将类标签 n（k类）转换为长度为 k 的向量，其中索引 n 为“hot”(1)，而其余为 0 。

(2) 确定输入层和输出层的单元数。较为合理的默认选择是只有一层隐藏层，如果有多个隐藏层，那么每个隐藏层的单元数最好相同（虽然更多的单元数会得到更好的结果，但是也要考虑到计算量）。隐藏层的单元数还应该和输入层的单元数相匹配，可以是1倍、2倍、3倍4倍等。

(3) 随机初始化完整网络参数大小的参数数组，也就是随机初始化权重。通常我们会把权重初始化为很小的值，接近于0。

(4) 将 X 和 y 转换为可以用于矩阵计算的格式

(5) 将参数数组解开为每个层的参数矩阵

2、评估一组给定的网络参数的损失的代价函数。反向传播参数更新计算将减少训练数据上的网络误差并返回代价和梯度。

3、在反向传播函数经过一定次数的迭代之后，将总代价下降到0.5以下。

4、使用反向传播函数得到的参数，通过网络前向传播以获得预测。

1.1 数据可视化

在这一节，你应当将数据集读入你的项目中，并对数据集进行可视化。

ex4data1.mat 是我们的手写数字数据集，其中包含 5000 个手写数字的训练示例。mat 格式意味着数据被保存为本地 Octave/MATLAB 矩阵格式，而不是像 csv 文件那样的文本（ASCII）格式。

这些矩阵可以通过使用 loadmat()命令直接读取到您的程序中。

```
from scipy.io import loadmat
# 加载数据
def loaddata(path):
    data=loadmat(path)
    return data
```

在 ex4data1.mat 中有 5000 个训练示例，其中每个训练示例是一个 20 像素乘 20 像素的数字灰度图像。每个像素用一个浮点数表示，表示该像素的灰度。20 乘 20 的像素网格被“展开”成一个 400 维的向量。这些训练示例在我们的数据矩阵 X 中都变成了一行。这给了我们一个 5000 乘 400 的矩阵 X，其中每一行都是一个手写数字图像的训练示例。

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

训练集的第二部分是一个 5000 维的向量 y，其中包含了训练集的标签。其记录值为1到10的整数，在 Python 代码编写中的索引对应规则为整体加一，即索引0对应数字1，索引1对应数字2，...，索引9对应数字10。

接下来我们要可视化训练集的一个子集。

从 X 中随机选择 100 行，并将这 100 行传递给可视化数据函数。该函数将每 一行映射到 20 像素乘 20 像素的灰度图像，并将图像显示在一起。代码如下：

```
def displayData(x):
    indexs=np.random.choice(x.shape[0],100)
    images=x[indexs]
    fig,axs=plt.subplots(10,10,figsize=(20,20))
    for row in range(10):
        for col in range(10):
            axs[row,col].matshow(images[row*10+col].reshape(20,20).T,cmap='gray_r')
    plt.show()
```

你的输出效果将与下图类似。

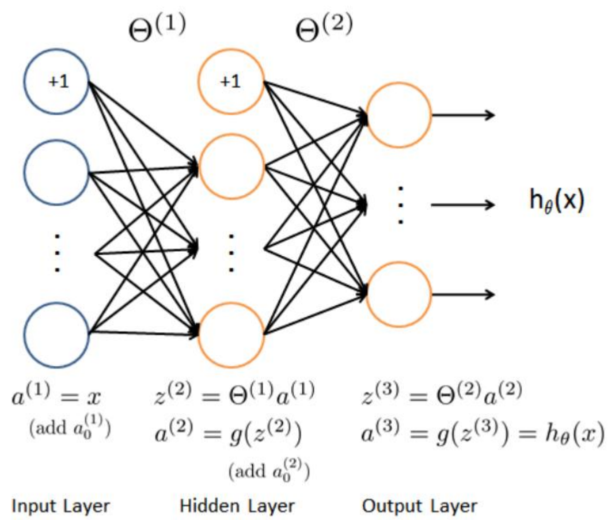
1.2 前向传播与损失函数

在这部分的练习中，你将实现一个神经网络，使用 ex4data1.mat 训练集来识别手写数字。

与上一次实验不同的是，你将不再使用我们已经训练过的神经网络中的参数，而是通过**实现反向传播算法来寻找适合的参数，并用这组参数进行预测。**

1.2.1 模型表示与初始化设置

我们的神经网络如上图所示。它由三层组成：**一个输入层，一个隐藏层和一个输出层。**



回想一下，我们的输入是数字图像的像素值。由于图像的大小是 20×20，这里我们设置了 400 个输入层单元（不包括总是输出+1 的额外偏置单元）。与之前一样，训练数据将被加载到变量 X 和 y 中。我们已经提供了一组已经训练过的网络参数（Θ(1)，Θ(2)）。这些参数存储在 ex4weights.mat 中，并将通过 loadmat()函数加载到 Theta1 和 Theta2 中。这些参数的尺寸适合一个神经网络，第二层有 25 个输入单元和 10 个输出单元（对应于 10 个数字类）。

在本实验中我们的初始化设置如下：

```python
# 初始化设置
# 确定输入层和输出层的单元数
input_size = 400 # 输入层单元
hidden_size = 25 # 隐藏层单元
num_labels = 10  # 输出层单元
lmbd = 1 # 正则化系数

# 随机初始化完整网络参数大小的参数数组
# 通常会把权重初始化为很小的值，接近于0
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labels * (hidden_size + 1)) - 0.5) * 0.25

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

# 将参数数组解开为每个层的参数矩阵
theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))
```

1.2.2 前向传播

现在，你将为神经网络实现前向传播。你需要完成以下代码。与上一实验的工作原理相同，但注意传入参数与返回值的不同，你的函数应该返回各个层的输入与输出，在后续的反向传播中我们将使用这些数据。

```python
def forward_propagate(X, theta1, theta2):

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

1.2.3 损失函数

回忆课上讲解的神经网络的损失函数：

$$J(\theta)=\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k)-(1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right],$$

其中 K = 10 是可能标签的总数。注意 $h_\theta(x^{(i)})_k=a_k^{(3)}$ 是第 k 个输出单元的激活(输出值)。另外，回想一下，(在变量y中)原始的标签是1, 2, ⋯, 10，为了训练神经网络，我们需要将标签重新编码为仅包含值 0 或 1 的向量，这里我们提供了函数 oneHotEncoder() 来完成这项工作。这个函数对 y

标签进行一次 one-hot 编码。 one-hot 编码将类标签 n（k类）转换为长度为 k 的向量，其中索引 n 为“hot”(1)，而其余为 0。

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots \quad \text{or} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

```python
def oneHotEncoder(arr):
    m = arr.shape[0]
    arr_onehot = np.zeros((m, 10))
    for i in range(m):
        arr_onehot[i][arr[i]-1] = 1
    return arr_onehot
```

损失函数的形式如图，注意返回值为标量：

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

    # 将参数数组重新塑造为每一层的参数矩阵
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))

    # 运行向前传播函数
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # 计算代价函数
    # 完成这部分

    return J # 返回值为标量
```

1.2.4 带正则化项的损失函数

回忆课上讲解的神经网络的损失函数（带正则化项）：

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

在上一小节完成的损失函数的基础上，加入正则化项。注意正则化项求和时，theta0不参与计算。

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

    # 将参数数组重新塑造为每一层的参数矩阵
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))

    # 运行向前传播函数
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # 计算代价函数
    # 完成这部分

    return J # 返回值为标量
```

1.3 反向传播算法

在这部分中，你要完成反向传播算法的实现，并将你的算法投入 Scipy 库的 scipy.optimize.minimize()方法来拟合神经网络参数。

1.3.1 Sigmoid函数梯度

Sigmoid函数的梯度公式为

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1-g(z))$$

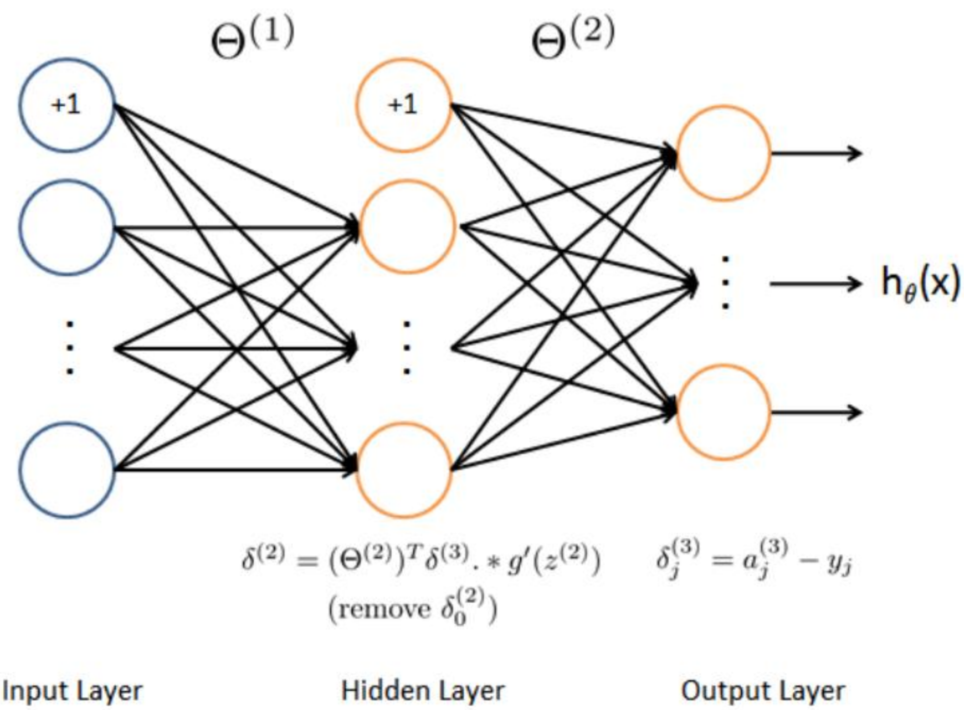
其中，
$$\mathrm{sigmoid}(z) = g(z) = \frac{1}{1+e^{-z}}.$$

以下是Sigmoid梯度的函数实现：

```python
# 计算sigmoid函数的梯度

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

1.3.2 反向传播



现在，你将实现反向传播算法。回想一下反向传播算法背后的原理。给定一个训练示例$\left(x^{(t)}, y^{(t)}\right)$，我们将首先运行一个“向前传递”来计算整个网络的所有激活值，包括假设$h_\Theta(x)$的输出值。然后，对于层l中的每个节点j，我们想要计算一个“误差项”$\delta_j^{(l)}$，它度量该节点对输出中的任何错误“负责”的程度。对于一个输出节点，我们可以直接测量网络激活与真实目标值之间的差值，并使用它来定义$\delta_j^{(3)}$(第3层是输出层)。对于隐藏单元，您将根据层$(l+1)$中节点的误差项的加权平均计算$\delta_j^{(l)}$。

BP算法的伪代码如下：

Backpropagation算法中的梯度计算

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

1. Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j).
2. For $i = 1$ to m

   Set $a^{(1)} = x^{(i)}$

   Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

   Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

   Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

   $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ 为每个样本累计误差delta
3. $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \begin{cases} \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m} \Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$

在具体算法的实现上，我们使用前向传播函数调用得到的返回值计算各层的误差（第3层与第2层）。输出层（第3层）的误差：

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

面向隐含层（第2层）的误差反向传播：

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$

之后使用误差梯度计算模型参数梯度：

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$$

通过将累积梯度除以m，获得神经网络成本函数的(非正则化)梯度：

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

带正则化项的梯度为：

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} \qquad \text{for } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \qquad \text{for } j \geq 1$$

请完成以下反向传播函数。注意返回值中损失 J 应为标量，grad为一维数组，长度为(10285,)。

```
# 反向传播算法
# 扩展代价函数以执行反向传播并返回代价和梯度
def backprop(params, input_size, hidden_size, num_labels, X, y, lmbd):
    m = X.shape[0]
    X = np.matrix(X)
    y = np.matrix(y)
    # 将参数数组重新塑造为每一层的参数矩阵
    theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size + 1))))
    theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size + 1))))

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

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

    # 计算代价函数
    """完成此部分代码"""
    # 加上正则化项
     """完成此部分代码"""

    # 执行反向传播
     """完成此部分代码"""

    # 加上梯度正则化项
     """完成此部分代码"""

    # 将梯度矩阵分解为单个数组
    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

    return J, grad
```

1.3.3 训练模型

完成上面的任务后，调用minimize函数来训练我们的模型，探索适合的参数。由于目标函数不太可能完全收敛，我们对迭代次数做了限制，为250次。

```python
from scipy.optimize import minimize

fmin = minimize(fun=backprop,
                x0=params,
                args=(input_size, hidden_size, num_labels, X, y_onehot, lmbd),
                method='TNC',
                jac=True,
                options={'maxiter': 250})
```

训练的结果应该和下图近似。

```python
print(fmin)
```

```
     fun: 0.34867101919659305
     jac: array([-2.71958387e-04, -2.11590224e-06,  1.91824736e-06, ...,
        1.62805874e-04,  2.18548897e-05,  1.08015433e-04])
 message: 'Max. number of function evaluations reached'
    nfev: 251
     nit: 22
  status: 3
 success: False
       x: array([-0.30487074, -0.01057951,  0.00959124, ...,  1.16133735,
       -2.00362072, -0.86245273])
```

1.4 评估模型

1.4.1 前向传播预测

在上一节我们完成了参数的训练，得到了一组神经网络参数，接下来在我们的数据集上使用这组参数进行前向传播。

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

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

np.argmax函数返回一组数中最大值的索引。

1.4.2 计算准确率

当你采用lmbd=1，迭代次数=250的设置进行训练，得到的准确率大致为99%。

```
# 计算准确度
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 = 99.44%
```