The results you need to get in ex4

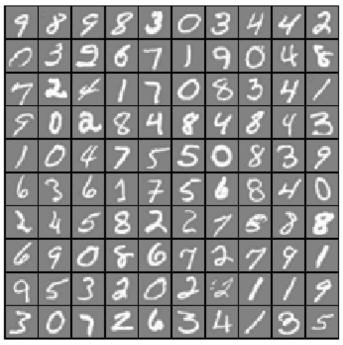
ex4

必做部分

第一部分 神经网络NN

1.1 数据可视化

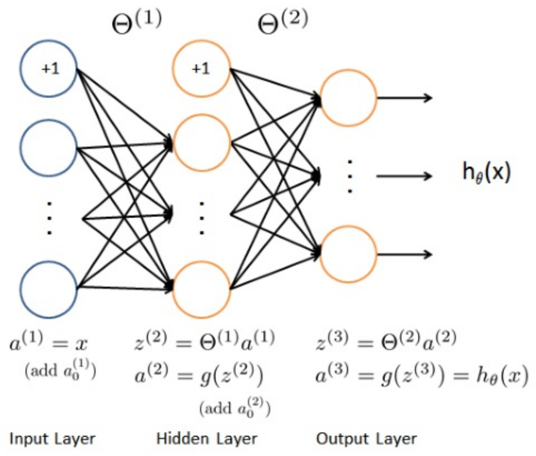
运行ex4.m，调用displayData函数，数据来自ex3data.mat中的5000个训练样本中的100个



注意输入部分：

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

1.2 模型展现



ex4weights.mat文件提供模型所需参数，Theta1（25*401）和Theta2（10*26），由ex4.m文件进行load，

1.3 前馈传播与损失函数

完成nnCostFunction.m文件，返回损失函数和J（θ）和梯度gradient，

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

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

注意：

1.ex4.m中的nn_params = [Theta1(:) ; Theta2(:)];表示已经将模型参数转换为向量形式，需要在nnCostFunction.m文件中将该向量重新转换出Theta1和Theta2权重矩阵

2.使用for-loop来进行计算

3.lambda = 0;表示目前不进行损失函数正则化

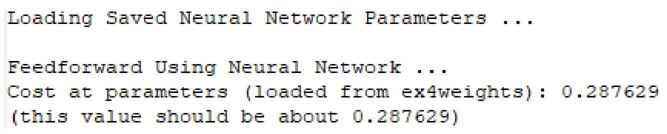
4.返回的参数梯度应当是一个unrolled展开的神经网络的偏微分向量

完成nnCostFunction.m文件：

注意：y为元素值为-10的列向量，J必须为单个实数值

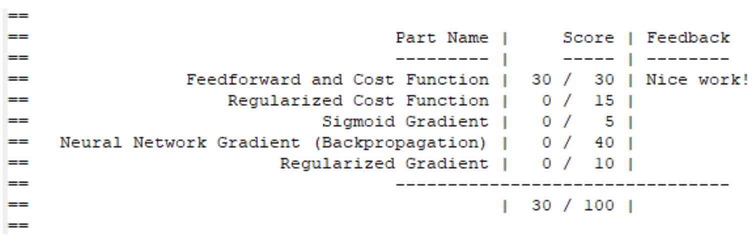
```
%Theta为下一层乘以当前层加1
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                  num_labels, (hidden_layer_size + 1));
%样本数量m
m = size(X, 1);
%初始化损失函数J和梯度gradient，也是需要返回的变量
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));
%正向传播产生推算值
a1 = [ones(m, 1) X];
z1 = a1 * Theta1';
a2 = [ones(m,1) sigmoid(z1)];
z3 = a2 * Theta2';
a3 = sigmoid(z3);
%矩阵斜对角线上为样本输出y的所有标签值
m_y = diag(ones(1,num_labels),0);
for i = 1:m
  J =J + (-log(a3(i,:)) * m_y(:,y(i)) - log(1-a3(i,:)) * (1-m_y(:,y(i))));
end
J = J/m;
```

输出:



```
Loading Saved Neural Network Parameters ...

Feedforward Using Neural Network ...
Cost at parameters (loaded from ex4weights): 0.287629
(this value should be about 0.287629)
```

第一次提交：



```
==
==                                   Part Name |     Score | Feedback
==                                   --------- |     ----- | --------
==                Feedforward and Cost Function |  30 /  30 | Nice work!
==                    Regularized Cost Function |   0 /  15 |
==                             Sigmoid Gradient |   0 /   5 |
==  Neural Network Gradient (Backpropagation) |   0 /  40 |
==                         Regularized Gradient |   0 /  10 |
==                                   --------------------------------
==                                             |  30 / 100 |
==
```

1.4 损失函数正则化

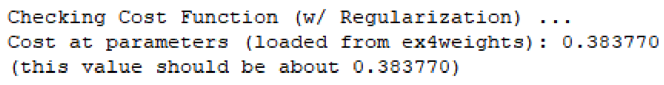
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

注意：我们可以假设整个神经网络共有3层，但是每一层的单元数需要任意化，Theta1和Theta2也应该可以是任意大小的，正则化不应该包含两个权重矩阵的第一列。

代码：

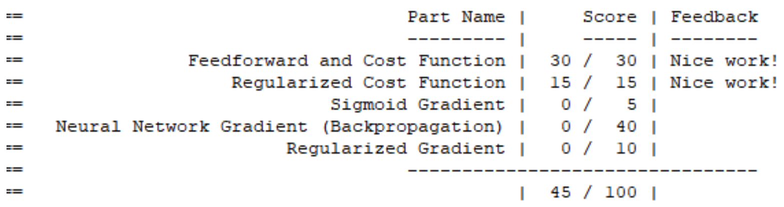
```
t1=0;
t2=0;
for i = 1:size(Theta1,1)
  for j = 2:size(Theta1,2)
    t1 = t1 + Theta1(i,j) ^2;
  end
end
for i = 1:size(Theta2,1)
  for j = 2:size(Theta2,2)
    t2 = t2 + Theta2(i,j) ^2;
  end
end
J = J + lambda * ( t1 + t2) / (2 * m);
```

输出结果：



```
Checking Cost Function (w/ Regularization) ...
Cost at parameters (loaded from ex4weights): 0.383770
(this value should be about 0.383770)
```

第二次提交：



```
==                                   Part Name |     Score | Feedback
==                                   --------- |     ----- | --------
==                Feedforward and Cost Function |  30 /  30 | Nice work!
==                    Regularized Cost Function |  15 /  15 | Nice work!
==                             Sigmoid Gradient |   0 /   5 |
==    Neural Network Gradient (Backpropagation) |   0 /  40 |
==                         Regularized Gradient |   0 /  10 |
==                                   --------------------------------
==                                             |  45 / 100 |
```

第二部分 反向传播算法

2.1 sigmoid 梯度

完成sigmoidGradient.m文件，其需要能够作用于矩阵中的每一个元素，完成后使用数值进行测试

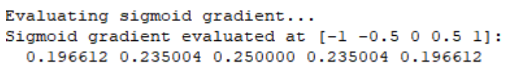
代码：

```
g = zeros(size(z));
%调用逻辑函数，注意这里直接矩阵相乘错误，需要元素相乘
g=sigmoid(z).*(1-sigmoid(z));
```

ex4.m文件中会进行校验：

```
g = sigmoidGradient([-1 -0.5 0 0.5 1]);
fprintf('Sigmoid gradient evaluated at [-1 -0.5 0 0.5 1]:\n ');
fprintf('%f ', g);
```

结果:



```
Evaluating sigmoid gradient...
Sigmoid gradient evaluated at [-1 -0.5 0 0.5 1]:
  0.196612 0.235004 0.250000 0.235004 0.196612
```

第三次提交：

```
=                                             Part Name |     Score | Feedback
=                                             --------- |     ----- | --------
=                     Feedforward and Cost Function |  30 /  30 | Nice work!
=                         Regularized Cost Function |  15 /  15 | Nice work!
=                                  Sigmoid Gradient |   5 /   5 | Nice work!
=    Neural Network Gradient (Backpropagation) |   0 /  40 |
=                             Regularized Gradient |   0 /  10 |
=                                             --------------------------------
=                                                   |  50 / 100 |
```

2.2 随机的初始化

symmetry breaking的随机初始化很重要，训练神经网络已经不能使用全部为0的权重矩阵，会导致没一个隐藏层的单元全部相同。我们将初始权或者能够参数在

$$[-\epsilon_{init}, \epsilon_{init}]$$

中进行随机选择，而这个范围的选定也是根据

$$\frac{\sqrt{6}}{\sqrt{L_{in}+L_{out}}}$$

公式的，这里使epsilon_init为0.12，现在完成randInitializeWeights.m文件，无需再次提交。我们可以先随机生成一个参数矩阵（row为当前输出层单元数，column为当前输入层单元数加1（考虑到bias值）），随机矩阵的元素为0-1的任意值，乘以两倍的epsilon_init,减去一倍的epsilon，则随机矩阵中的元素值将会全部处于上述范围内。

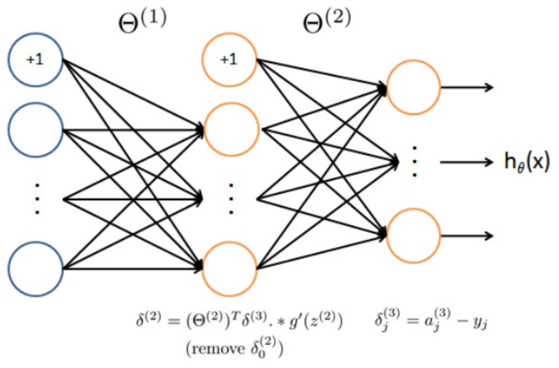
代码已经给出：

```
W = zeros(L_out, 1 + L_in);
epsilon_init=0.12;
W=rand(L_in,L_out+1)*2*epsilon-epsilon;
```

ex4.m文件中已经将随即生成的Theta1和Theta2参数矩阵叠加在一起展开成了向量的形式：

```
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);
%展开参数矩阵为向量形式
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];
```

2.3 实现反向传播算法(获取尚未正则化的梯度值）



$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$

每一层的code均需要计算出error项，表示该节点需要对output输出的误差负多大的责任，输出层单元的误差可以使用样本输出值减去推测值直接计算，而隐藏层的单元则可以使用后一层的加权平均值。完成nnCostFunction.m文件的Part2部分，如果遇到“nonconformant arguments”报错，使用size函数查询矩阵变量维度是否有误。

nnCostFunction.m文件中添加代码：

```
X = [ones(m,1) X];
for i = 1:m
  a_1 = X(i,:)';
  z_2 = Theta1*a_1;
  a_2 = sigmoid(z_2);
  a_2 = [1;a_2];
  z_3 = Theta2*a_2;
  a_3 = sigmoid(z_3);
  delta_3 = a_3 - m_y(:,y(i));
  delta_2 = (Theta2' * delta_3) .* sigmoidGradient([1;z_2]);
  %fprintf(['J:%f'],size(sigmoidGradient(z_2),2));
  delta_2= delta_2(2:end);
  Theta1_grad=Theta1_grad+delta_2*a_1';
  Theta2_grad=Theta2_grad+delta_3*a_2';
end
Theta1_grad=Theta1_grad/m;
Theta2_grad=Theta2_grad/m;
```

2.4 梯度校验

ex4.m文件中进行梯度校验：

```
fprintf('\nChecking Backpropagation... \n');
```

% 调用checkNNGradients函数进行校验，将反向传播求得的梯度与近似方法（只能用于小型神经网络）求得的梯度进行比较，如果相对误差小于10-9，则反向传播算法正常实现

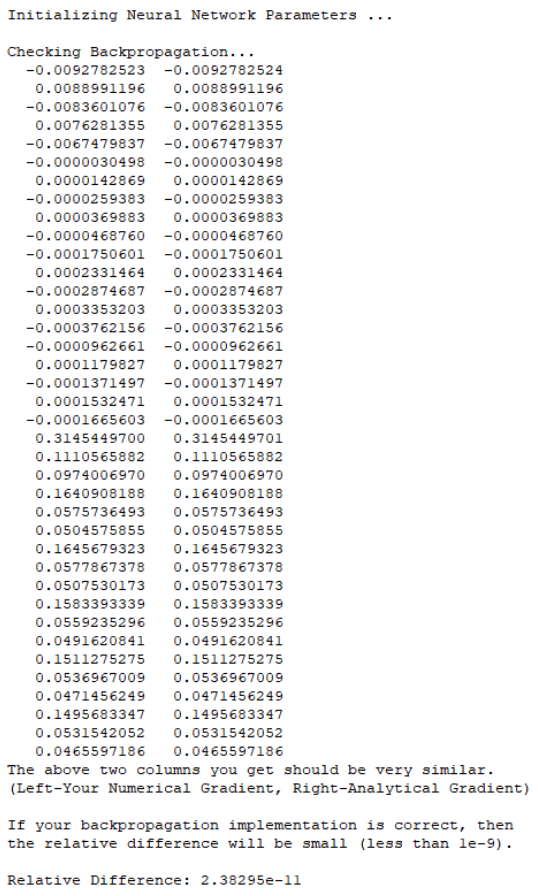
```
checkNNGradients;
```

checkNNGradients又会调用computeNumericalGradient函数：

```
numgrad = zeros(size(theta));
perturb = zeros(size(theta));
e = 1e-4;
for p = 1:numel(theta)
    % Set perturbation vector
    perturb(p) = e;
    loss1 = J(theta - perturb);
    loss2 = J(theta + perturb);
    % Compute Numerical Gradient
    numgrad(p) = (loss2 - loss1) / (2*e);
    perturb(p) = 0;
end
```

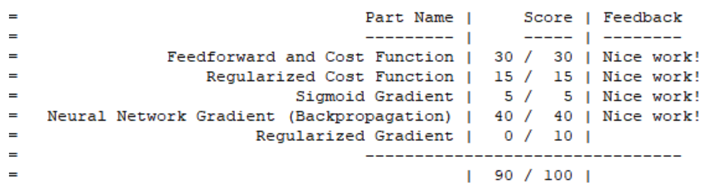
注意：在验证算法正确后正式运行算法之前，关闭梯度校验checkNNGradients，以免影响运行效率；computeNumericalGradient的方法也可以用于检测其余模型的损失函数和梯度（如：逻辑回归）；1的添加项只有在a_2已经计算出来之后才可以添加，不能在z_2上就添加，否则会导致相对误差较大。

输出结果：误差值在可接受范围内，因此对损失函数和梯度自信



```
Initializing Neural Network Parameters ...

Checking Backpropagation...
  -0.0092782523  -0.0092782524
   0.0088991196   0.0088991196
  -0.0083601076  -0.0083601076
   0.0076281355   0.0076281355
  -0.0067479837  -0.0067479837
  -0.0000030498  -0.0000030498
   0.0000142869   0.0000142869
  -0.0000259383  -0.0000259383
   0.0000369883   0.0000369883
  -0.0000468760  -0.0000468760
  -0.0001750601  -0.0001750601
   0.0002331464   0.0002331464
  -0.0002874687  -0.0002874687
   0.0003353203   0.0003353203
  -0.0003762156  -0.0003762156
  -0.0000962661  -0.0000962661
   0.0001179827   0.0001179827
  -0.0001371497  -0.0001371497
   0.0001532471   0.0001532471
  -0.0001665603  -0.0001665603
   0.3145449700   0.3145449701
   0.1110565882   0.1110565882
   0.0974006970   0.0974006970
   0.1640908188   0.1640908188
   0.0575736493   0.0575736493
   0.0504575855   0.0504575855
   0.1645679323   0.1645679323
   0.0577867378   0.0577867378
   0.0507530173   0.0507530173
   0.1583393339   0.1583393339
   0.0559235296   0.0559235296
   0.0491620841   0.0491620841
   0.1511275275   0.1511275275
   0.0536967009   0.0536967009
   0.0471456249   0.0471456249
   0.1495683347   0.1495683347
   0.0531542052   0.0531542052
   0.0465597186   0.0465597186
The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)

If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.38295e-11
```

此时提交：



```
=                                   Part Name |     Score | Feedback
=                                   --------- |     ----- | --------
=               Feedforward and Cost Function |  30 /  30 | Nice work!
=                   Regularized Cost Function |  15 /  15 | Nice work!
=                            Sigmoid Gradient |   5 /   5 | Nice work!
=   Neural Network Gradient (Backpropagation) |  40 /  40 | Nice work!
=                        Regularized Gradient |   0 /  10 |
=                                   --------------------------------
=                                             |  90 / 100 |
```

2.5 正则化神经网络

损失函数已经进行过正则化，现在对梯度gradient进行正则化，我们可以添加附加项到前面反向传播求出的dradient后面，注意不能对参数矩阵的第一列（对应于bias偏差项）进行正则化

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

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

nnCostFunction.m文件中添加代码：

Theta1_grad=Theta1_grad+[zeros(size(Theta1,1),1) lambda*Theta1(:,2:end) ./m];

Theta2_grad=Theta2_grad+[zeros(size(Theta2,1),1) lambda*Theta2(:,2:end) ./m];

另一种写法：

Theta1_grad = 1/m * Theta1_grad + lambda/m * Theta1;

Theta1_grad(:,1) = Theta1_grad(:,1) - lambda/m * Theta1(:,1);

Theta2_grad = 1/m * Theta2_grad + lambda/m * Theta2;

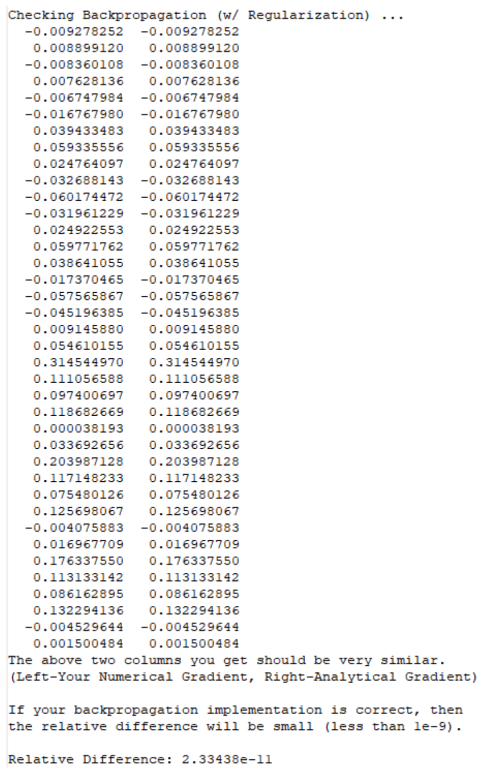
Theta2_grad(:,1) = Theta2_grad(:,1) - lambda/m * Theta2(:,1);

再次进行梯度校验，lambda设置为3：

lambda = 3;

checkNNGradients(lambda);

运行结果:偏差依然在可接受范围之内

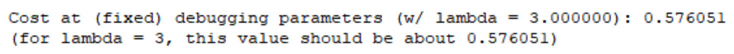


```
Checking Backpropagation (w/ Regularization) ...
  -0.009278252  -0.009278252
   0.008899120   0.008899120
  -0.008360108  -0.008360108
   0.007628136   0.007628136
  -0.006747984  -0.006747984
  -0.016767980  -0.016767980
   0.039433483   0.039433483
   0.059335556   0.059335556
   0.024764097   0.024764097
  -0.032688143  -0.032688143
  -0.060174472  -0.060174472
  -0.031961229  -0.031961229
   0.024922553   0.024922553
   0.059771762   0.059771762
   0.038641055   0.038641055
  -0.017370465  -0.017370465
  -0.057565867  -0.057565867
  -0.045196385  -0.045196385
   0.009145880   0.009145880
   0.054610155   0.054610155
   0.314544970   0.314544970
   0.111056588   0.111056588
   0.097400697   0.097400697
   0.118682669   0.118682669
   0.000038193   0.000038193
   0.033692656   0.033692656
   0.203987128   0.203987128
   0.117148233   0.117148233
   0.075480126   0.075480126
   0.125698067   0.125698067
  -0.004075883  -0.004075883
   0.016967709   0.016967709
   0.176337550   0.176337550
   0.113133142   0.113133142
   0.086162895   0.086162895
   0.132294136   0.132294136
  -0.004529644  -0.004529644
   0.001500484   0.001500484
The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)

If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.33438e-11
```

测试损失函数：

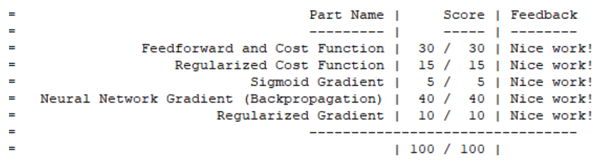
debug_J = nnCostFunction(nn_params, input_layer_size,hidden_layer_size, num_labels, X, y, lambda);

fprintf(['\n\nCost at (fixed) debugging parameters (w/ lambda = %f): %f ' '\n(for lambda = 3, this value should be about 0.576051)\n\n'], lambda, debug_J);



```
Cost at (fixed) debugging parameters (w/ lambda = 3.000000): 0.576051
(for lambda = 3, this value should be about 0.576051)
```

此时提交：



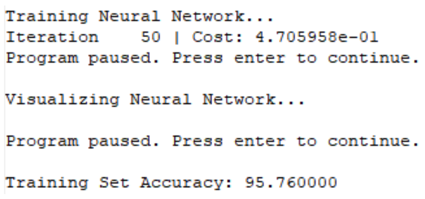
```
=                                    Part Name |     Score | Feedback
=                                    --------- |     ----- | --------
=                 Feedforward and Cost Function |  30 /  30 | Nice work!
=                     Regularized Cost Function |  15 /  15 | Nice work!
=                              Sigmoid Gradient |   5 /   5 | Nice work!
=   Neural Network Gradient (Backpropagation) |  40 /  40 | Nice work!
=                          Regularized Gradient |  10 /  10 | Nice work!
=                                    --------------------------------
=                                              | 100 / 100 |
```

2.6 使用fmincg拟合参数

ex4.m文件将会调用fincg函数来训练数据集参数，训练完成后计算准确率，设置更高的迭代次数可能获得更高的准确率（在设置迭代次数为100时，准确率达到了98.08%）：

```
fprintf('\nTraining Neural Network... \n');
options = optimset('MaxIter', 50);
lambda = 1;
%对损失函数进行short hand
costFunction = @(p) nnCostFunction(p, ...
                        input_layer_size, ...
                        hidden_layer_size, ...
                        num_labels, X, y, lambda);
[nn_params, cost] = fmincg(costFunction, initial_nn_params, options);
%训练完成后的参数为向量，重新转换出两个权重矩阵
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
            hidden_layer_size, (input_layer_size + 1));
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
            num_labels, (hidden_layer_size + 1));
%准确度计算
pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

结果：



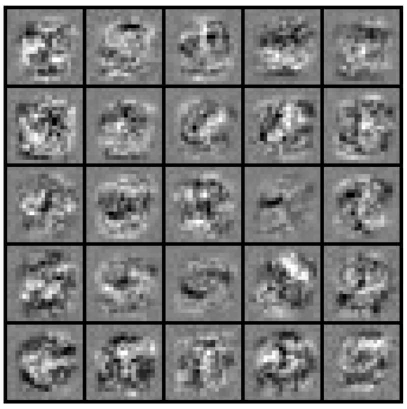
```
Training Neural Network...
Iteration    50 | Cost: 4.705958e-01
Program paused. Press enter to continue.

Visualizing Neural Network...

Program paused. Press enter to continue.

Training Set Accuracy: 95.760000
```

第三部分 隐藏层可视化

了解你的神经网络正在如何进行学习，对隐藏层单元捕获的东西进行表现，将图片从20*20转换成1*400，乘以Theta1之后转换成20*20,显示图片，注意不能涉及到偏差项。

fprintf('\nVisualizing Neural Network... \n');

displayData(Theta1(:, 2:end));



选做部分

3.1 可选练习

对于同一个神经网络，尝试不同的学习设置，观察不同的推算效果（正则化参数lambda，学习步长的数量）。前面已经更改过迭代次数，这里尝试更改正则化参数。100次迭代加上lambda为3，准确率降为了96.7%

总结：神经网络是能够生成高度复杂决策边界的强大模型，如果没有正则化，其很可能会过拟合训练集，几乎达到100%的准确度，而在测试集上表现较差，我们只要将lambda减小，将迭代次数MaxIter变大就可以观察到这一结论，此时隐藏单元的可视化也会发生改变。