5.3 $p(T|X, W, \Sigma) = \prod_{n=1}^{N} N(t_n | y(x_n, w), \Sigma)$

$$~~\phi~~ \ln p(T|X, W, \Sigma) = \frac{-N}{2}(\ln|\Sigma| + K\ln(2\pi)) - \frac{1}{2}\sum_{n=1}^{N}(t_n - y_n)^T \Sigma^{-1}(t_n - y_n)$$

↑ likelihood function

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}(t_n - y_n)^T \Sigma^{-1}(t_n - y_n)$$

↑ error function

Maximize likelihood with respect to Σ

$$-N/2 \ln|\Sigma| - 1/2 \sum_{n=1}^{N}(t_n - y_n)^T \Sigma^{-1}(t_n - y_n)$$

$$= -N/2 \ln|\Sigma| - 1/2 \, Tr\left[\Sigma^{-1}\sum_{n=1}^{N}(t_n - y_n)(t_n - y_n)^T\right]$$

$$\therefore \Sigma = \frac{1}{N}\sum_{n=1}^{N}(t_n - y_n)(t_n - y_n)^T$$

5.4 $p(t=1|x) = \sum_{h=0}^{1} p(t=1|h)\,p(h|x)$

$$= (1-\epsilon)y(x, w) + \epsilon(1 - y(x, w))$$

$$p\,(t|x) = p(t=1|x)^t(1 - p(t=1|x))^{1-t}$$

$$\therefore E(w) = -\sum_{n=1}^{N}(t_n \ln(1-\epsilon)y(x_n, w) + \epsilon(1 - y(x, w)) + (1-t_n)\ln(1 - (1-\epsilon)y(x_n, w) - \epsilon(1 - y(x_n, w))))$$

When ~~x~~ $\epsilon = 0$, this will reduce.

5.26 $\Omega_n = \frac{1}{2} \sum_k \left(\sum_i T_{ni} \frac{\partial y_{ni}}{\partial x_{ni}} \right)^2$

$= \frac{1}{2} \sum_k \left(\sum_i T_{ni} J_{nki} \right)^2$ ← Jacobian from textbook

$\alpha_j = \sum_i T_i \frac{\partial z_j}{\partial x_i} = \sum_i T_i \frac{\partial h(a_j)}{\partial x_i}$

$= \sum_i T_i \frac{\partial h(a_j)}{\partial a_j} \frac{\partial a_j}{\partial x_i}$

$= h'(a_j) \sum_i T_i \frac{\partial}{\partial x_i} a_j = h'(a_j) \beta_j$

$\beta_j = \sum_i T_i \frac{\partial a_j}{\partial x_i} = \sum_i T_i \frac{\partial \sum_{i'} w_{ji'} z_{i'}}{\partial x_i}$

$= \sum_i T_i \sum_{i'} \frac{\partial w_{ji'} z_{i'}}{\partial x_i} = \sum_i T_i \sum_{i'} w_{ji'} \frac{\partial z_{i'}}{\partial x_i}$

$= \sum_{i'} w_{ji'} \sum_i T_i \frac{\partial z_{i'}}{\partial x_i} = \sum_{i'} w_{ji'} \alpha_{i'}$

$\beta_{nj} = \sum_i w_{ji} \alpha_{ni}$

$= \sum_i w_{ji} x_{ni}$

$= \sum_i w_{ji} \sum_i T_{ni} \frac{\partial x_{ni}}{\partial x_{ni}}$

$= \sum_i w_{ji} T_{ni}$

$$\Omega_n = \frac{1}{2} \sum_k (y_{nk})^2$$

$$\Omega = \frac{1}{2} \sum_k a^2_{nk}$$

$$\frac{\partial \Omega_n}{\partial w_{rs}} = \sum_k (y_{nk})(\delta_{nkr} z_{ns})$$

$$\sum_k a_{nk}(\phi_{nkr} z_{ns} + \delta_{nkr} a_{ns})$$

$$\delta_{nhr} = h'(a_{hr}) \sum_i w_{ir} \delta_{nhr}$$

$$\phi_{nkr} = G\delta_{nkr}$$

$$= G(h'(a_{hr}) \sum_i w_{ir} \delta_{nhr})$$

$$= h''(a_{hr}) \beta_{nr} \sum_i w_{ir} \delta_{nhr}$$

$$= + h'(a_{hr}) \sum_i w_{ir} \phi_{nhr}$$

Implementation for the cost function followed the guidelines in the given notes. We calculate the forward propagation, average activation, back propagation, and error as given. Error between numerical and analytical gradients was found to be low (e^-11). In the end our cost function did not converge at the given 400 repetitions, so we increased it to 1000, and found that the minimize function took 833 repetitions. Overall, the results looked OK, at 400, 1000, and 2000 repetitions. We found clear lines, even though some of them are curved, which is not ideal. From our prediction function, we find that there are several cells with clear estimates, but most cells are grey. However, this is much cleaner than the raw output shown earlier.

Struggles during this assignment were solved using the checkNumericalGradient() function, which allowed us to isolate issues to the cost. I then reduced the number of hidden units and the number of sample images to compute results faster. Lastly, I removed all regularization terms and changed the cost function to be only mean squared error until I found the source of why my analytical gradients did not match the numerical gradients. After that, I undid the rest of the changes to achieve a serviceable cost function.

By playing around with the parameters we find that the decayWeight and the sparsityParam are necessary to obtain good results. By putting them too high, we converge too quickly to find meaningful weights, and by putting them too low, we do not converge, and the results become pixelated. The beta term is used to weigh the average hidden output, so it learns at any setting other than 0. By increasing the number of hidden units and increasing beta, we can still get good results, but they take much longer to compute, and is not noticeably better.

In [3]:
```
from __future__ import division
import numpy as np
from numpy.linalg import norm
from numpy.random import rand
from numpy.random import randint
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import PIL.Image
from scipy.optimize import fmin_l_bfgs_b as minimize
import math
from random import randint
```

In [4]:
```
## ---------------------------------------------------------------
def sampleIMAGES():
    # sampleIMAGES
    # Returns 10000 patches for training

    IMAGES = np.load('IMAGES.npy')  # load images from disk

    patchsize = 8                   # we'll use 8x8 patches
    numpatches = 10000

    # Initialize patches with zeros. Your code will fill in this matrix--one
    # row per patch, 10000 rows.
    patches = np.zeros((numpatches, patchsize*patchsize))

    ## ---------- YOUR CODE HERE --------------------------------------
    #  Instructions: Fill in the variable "patches" using data
    #  from IMAGES.
    #
    #  IMAGES is a 3D array containing 10 images,
    #  and Python indexes arrays by starting from 0.
    #  For instance, IMAGES[:,:,0] is a 512x512 array containing the 1st image,
    #  and to visualize it you can type
    #      from matplotlib import pyplot as plt
    #      plt.imshow(IMAGES[:,:,0])
    #      plt.set_cmap('gray')
    #      plt.show()
    #  (The contrast on these images look a bit off because they have
    #  been preprocessed using using "whitening."  See the lecture notes for
    #  more details.) As a second example, IMAGES[20:30,20:30,1] is an image
    #  patch corresponding to the pixels in the block (20,20) to (29,29) of
    #  Image 2

    for i in range(numpatches):

        image_index = randint(0, 9)

        start_index1 = randint(0, 512-patchsize)
        end_index1 = start_index1 + patchsize;
    for i in range(numpatches):

        image_index = randint(0, 9)

        start_index1 = randint(0, 512-patchsize)
        end_index1 = start_index1 + patchsize;
        start_index2 = randint(0, 512-patchsize)
        end_index2 = start_index2 + patchsize;

        patch = IMAGES[start_index1:end_index1, start_index2:end_index2, image_index]

        temp = np.reshape(patch, (patchsize*patchsize))
        patches[i,:] = temp

#     ## ---------------------------------------------------------------
#     # For the autoencoder to work well we need to normalize the data
#     # Specifically, since the output of the network is bounded between [0,1]
#     # (due to the sigmoid activation function), we have to make sure
#     # the range of pixel values is also bounded between [0,1]
    patches = normalizeData(patches)
    return patches

#print(sampleIMAGES())
```

```
## ---------------------------------------------------------------
def normalizeData(patches):
    # Squash data to [0.1, 0.9] since we use sigmoid as the activation
    # function in the output layer

    # Remove DC (mean of images).
    patches = patches-np.array([np.mean(patches,axis=1)]).T

    # Truncate to +/-3 standard deviations and scale to -1 to 1
    pstd = 3*np.std(patches)
    patches = np.fmax(np.fmin(patches,pstd),-pstd)/pstd

    # Rescale from [-1,1] to [0.1,0.9]
    patches = (patches+1)*0.4+0.1
    return patches
```

In [19]:
```
## ---------------------------------------------------------------
def sparseAutoencoderCost(theta,visibleSize,hiddenSize,decayWeight,sparsityParam,beta,data):
    # visibleSize: the number of input units (probably 64)
    # hiddenSize: the number of hidden units (probably 25)
    # decayWeight: weight decay parameter lambda
    # sparsityParam: The desired average activation for the hidden units (denoted in the lecture
    #                           notes by the greek alphabet rho, which looks like a lower-case "p").
    # beta: weight of sparsity penalty term
    # data: Our 10000x64 matrix containing the training data.  So, data[i-1,:] is the i-th training example.

    # The input theta is a vector (because scipy.optimize.fmin_l_bfgs_b expects the parameters to be a vector).
    # We first convert theta to the (W1, W2, b1, b2) matrix/vector format, so that this
    # follows the notation convention of the lecture notes.
    W1,W2,b1,b2 = unravelParameters(theta,hiddenSize,visibleSize)

    # Cost and gradient variables (your code needs to compute these values).
    # Here, we initialize them to zeros.
    cost = 0
    W1grad = np.zeros(np.shape(W1))
    W2grad = np.zeros(np.shape(W2))
    b1grad = np.zeros(np.shape(b1))
    b2grad = np.zeros(np.shape(b2))

    ## ---------- YOUR CODE HERE --------------------------------------
    #  Instructions: Compute the cost/optimization objective J_sparse(W,b) for the Sparse Autoencoder,
    #                and the corresponding gradients W1grad, W2grad, b1grad, b2grad.
    #
    # W1grad, W2grad, b1grad and b2grad should be computed using backpropagation.
    # Note that W1grad has the same dimensions as W1, b1grad has the same dimensions
    # as b1, etc.  Your code should set W1grad to be the partial derivative of J_sparse(W,b) with
    # respect to W1.  I.e., W1grad(i,j) should be the partial derivative of J_sparse(W,b)
    # with respect to the input parameter W1(i,j).  Thus, W1grad should be equal to the term
    # [(1/m) \Delta W^{(1)} + \lambda W^{(1)}] in the last block of pseudo-code in Section 2.2
    # of the lecture notes (and similarly for W2grad, b1grad, b2grad).
    #
    # Stated differently, if we were using batch gradient descent to optimize the parameters,
    # the gradient descent update to W1 would be W1 := W1 - alpha * W1grad, and similarly for W2, b1, b2.
    #

    #forward
    m = data.shape[0]
    a2 = sigmoid(np.dot(data,W1)+np.array([b1]))
    z3 = np.dot(a2,W2)+np.array([b2])
    a3 = sigmoid(z3)

    #average activation
    rho_hat = np.average(a2,axis=0)
    #forward
    m = data.shape[0]
    a2 = sigmoid(np.dot(data,W1)+np.array([b1]))
    z3 = np.dot(a2,W2)+np.array([b2])
    a3 = sigmoid(z3)

    #average activation
    rho_hat = np.average(a2,axis=0)

    #backprop
    delta_3 = (a3-data)*a3*(1-a3)
    delta_2 = (np.dot(delta_3, W2.T) + (beta*((-sparsityParam/rho_hat)+((1-sparsityParam)/(1-rho_hat)) ))) * a2 * (1-a2)

    W1grad = np.dot(data.T, delta_2)
    W2grad = np.dot(a2.T, delta_3)

    b1grad = np.sum(delta_2, axis=0)
    b2grad = np.sum(delta_3, axis=0)

    W1grad = 1/m * W1grad + (decayWeight * W1)
    W2grad = 1/m * W2grad + (decayWeight * W2)

    b2grad = 1/m * b2grad
    b1grad = 1/m * b1grad

    #error
    mean_squared_error = 0.5*np.average(norm(a3-data, axis=1)**2)
    regularization_part = 0.5*decayWeight*(np.sum(W1**2) + np.sum(W2**2))
    sparsity_part = beta*np.sum([KL_divergence(sparsityParam, e) for e in rho_hat])
    cost = mean_squared_error + regularization_part + sparsity_part

#     #-------------------------------------------------------------------
#     # After computing the cost and gradient, we will convert the gradients back
#     # to a vector format (suitable for scipy.optimize.fmin_l_bfgs_b).
    grad = ravelParameters(W1grad,W2grad,b1grad,b2grad)
    return cost,grad
```

In [7]:
```
## ------------------------------------------------------------------
def sigmoid(x):
    # Here's an implementation of the sigmoid function, which you may find useful
    # in your computation of the costs and the gradients.  This inputs a (row or
    # column) vector (say (z1, z2, z3)) and returns (f(z1), f(z2), f(z3)).
    return 1/(1+np.exp(-x))

def KL_divergence(x, y):

    return x * np.log(x / y) + (1 - x) * np.log((1 - x) / (1 - y))
```

```python
def KL_divergence(x, y):

    return x * np.log(x / y) + (1 - x) * np.log((1 - x) / (1 - y))

## ---------------------------------------------------------------
def unravelParameters(theta,hiddenSize,visibleSize):
    # Convert theta to the (W1, W2, b1, b2) matrix/vector format
    W1 = theta[0:hiddenSize*visibleSize].reshape(visibleSize,hiddenSize)
    W2 = theta[hiddenSize*visibleSize:2*hiddenSize*visibleSize].reshape(hiddenSize,visibleSize)
    b1 = theta[2*hiddenSize*visibleSize:2*hiddenSize*visibleSize+hiddenSize]
    b2 = theta[2*hiddenSize*visibleSize+hiddenSize:]
    return W1,W2,b1,b2


## ---------------------------------------------------------------
def ravelParameters(W1,W2,b1,b2):
    # Unroll the (W1, W2, b1, b2) matrix/vector format to the theta format.
    return np.concatenate((W1.ravel(),W2.ravel(),b1.ravel(),b2.ravel()))


## ---------------------------------------------------------------
def checkNumericalGradient():
    # This code can be used to check your numerical gradient implementation
    # in computeNumericalGradient()
    # It analytically evaluates the gradient of a very simple function called
    # simpleQuadraticFunction (see below) and compares the result with your numerical
    # solution. Your numerical gradient implementation is incorrect if
    # your numerical solution deviates too much from the analytical solution.

    # Evaluate the function and gradient at x = [4; 10]; (Here, x is a 2d vector.)
    x = np.array([4,10])
    value,grad = simpleQuadraticFunction(x)

    # Use your code to numerically compute the gradient of simpleQuadraticFunction at x.
    # (The notation "lambda x: simpleQuadraticFunction(x)[0]" creates a function
    # that only returns the cost and not the grad of simpleQuadraticFunction.)
    numgrad = computeNumericalGradient(lambda x: simpleQuadraticFunction(x)[0],x)

    # Visually examine the two gradient computations.  The two columns
    # you get should be very similar.
    print (np.array([numgrad,grad]).T)
    print ("The above two columns you get should be very similar.")
    print ("(Left-Your Numerical Gradient, Right-Analytical Gradient)\n\n")

    # Evaluate the norm of the difference between two solutions.
    # If you have a correct implementation, and assuming you used EPSILON = 0.0001
    # in computeNumericalGradient.m, then diff below should be 2.1452e-12
    diff = norm(numgrad-grad)/norm(numgrad+grad)
    print (diff)
    print ("Norm of the difference between numerical and analytical gradient (should be < 1e-9)\n\n")
```

```
## ---------------------------------------------------------------
def checkNumericalGradient():
    # This code can be used to check your numerical gradient implementation
    # in computeNumericalGradient()
    # It analytically evaluates the gradient of a very simple function called
    # simpleQuadraticFunction (see below) and compares the result with your numerical
    # solution. Your numerical gradient implementation is incorrect if
    # your numerical solution deviates too much from the analytical solution.

    # Evaluate the function and gradient at x = [4; 10]; (Here, x is a 2d vector.)
    x = np.array([4,10])
    value,grad = simpleQuadraticFunction(x)

    # Use your code to numerically compute the gradient of simpleQuadraticFunction at x.
    # (The notation "lambda x: simpleQuadraticFunction(x)[0]" creates a function
    # that only returns the cost and not the grad of simpleQuadraticFunction.)
    numgrad = computeNumericalGradient(lambda x: simpleQuadraticFunction(x)[0],x)

    # Visually examine the two gradient computations.  The two columns
    # you get should be very similar.
    print (np.array([numgrad,grad]).T)
    print ("The above two columns you get should be very similar.")
    print ("(Left-Your Numerical Gradient, Right-Analytical Gradient)\n\n")

    # Evaluate the norm of the difference between two solutions.
    # If you have a correct implementation, and assuming you used EPSILON = 0.0001
    # in computeNumericalGradient.m, then diff below should be 2.1452e-12
    diff = norm(numgrad-grad)/norm(numgrad+grad)
    print (diff)
    print ("Norm of the difference between numerical and analytical gradient (should be < 1e-9)\n\n")


## ---------------------------------------------------------------
def simpleQuadraticFunction(x):
    # this function accepts a 2D vector as input.
    # Its outputs are:
    #   value: h(x1, x2) = x1^2 + 3*x1*x2
    #   grad: A 2-dim vector that gives the partial derivatives of h with respect to x1 and x2
    value = x[0]**2+3*x[0]*x[1]
    grad = np.zeros(np.shape(x))
    grad[0] = 2*x[0]+3*x[1]
    grad[1] = 3*x[0]
    return value, grad
```

In [8]:
```
## ---------------------------------------------------------------
def computeNumericalGradient(J,theta):
    # numgrad = computeNumericalGradient(J, theta)
    # theta: a vector of parameters
    # J: a function that outputs a real-number.
    # Calling y = J(theta) will return the function value at theta.
```

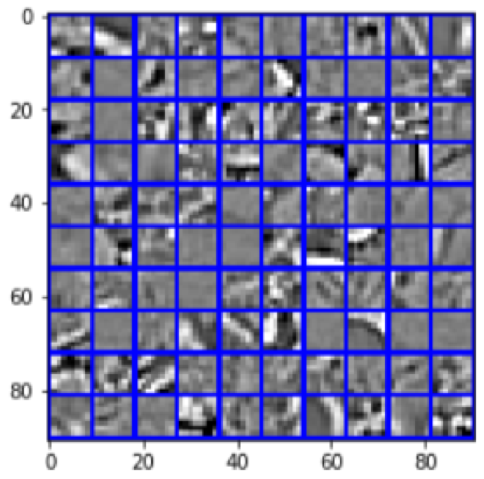
In [8]:
```python
## ---------------------------------------------------------------
def computeNumericalGradient(J,theta):
    # numgrad = computeNumericalGradient(J, theta)
    # theta: a vector of parameters
    # J: a function that outputs a real-number.
    # Calling y = J(theta) will return the function value at theta.

    # Initialize numgrad with zeros
    numgrad = np.zeros(np.shape(theta))

    ## ---------- YOUR CODE HERE --------------------------------------
    # Instructions:
    # Implement numerical gradient checking, and return the result in numgrad.
    # (See Section 2.3 of the lecture notes.)
    # You should write code so that numgrad(i) is (the numerical approximation to) the
    # partial derivative of J with respect to the i-th input argument, evaluated at theta.
    # I.e., numgrad(i) should be the (approximately) the partial derivative of J with
    # respect to theta(i).
    #
    # Hint: You will probably want to compute the elements of numgrad one at a time.


    epsilon = 1e-4
    num = len(theta)
    e = np.eye(num)

    for i in range(num):
        numgrad[i] = (J(theta + epsilon * e[i]) - J(theta - epsilon * e[i])) / (2* epsilon)

        if i % 100 == 0:
            print ("Computing gradient for input:", i)


    ## ---------------------------------------------------------------
    return numgrad
```

In [9]:
```
## ---------------------------------------------------------------
def initializeParameters(hiddenSize,visibleSize):
    # Initialize parameters randomly based on layer sizes.
    r = np.sqrt(6)/np.sqrt(hiddenSize+visibleSize+1)
    W1 = rand(visibleSize,hiddenSize)*2*r-r
    W2 = rand(hiddenSize,visibleSize)*2*r-r
    b1 = np.zeros((hiddenSize,1))
    b2 = np.zeros((visibleSize,1))

    # Convert weights and bias gradients to the vector form.
    # This step will "unroll" (flatten and concatenate together) all
    # your parameters into a vector, which can then be used
    # with scipy.optimize.fmin_l_bfgs_b.
    theta = ravelParameters(W1,W2,b1,b2)
    return theta


## ---------------------------------------------------------------
def displayNetwork(A,optNormEach=False,optNormAll=True,numColumns=None,imageWidth=None,cmapName='gray',
                   borderColor='blue',borderWidth=1,verbose=True,graphicsLibrary='matplotlib',saveName=''):
    # This function visualizes filters in matrix A. Each row of A is a
    # filter. We will reshape each row into a square image and visualizes
    # on each cell of the visualization panel. All other parameters are
    # optional, usually you do not need to worry about them.
    #
    # optNormEach: whether we need to normalize each row so that
    # the mean of each row is zero.
    #
    # optNormAll: whether we need to normalize all the rows so that
    # the mean of all the rows together is zero.
    #
    # imageWidth: how many pixels are there for each image
    # Default value is the squareroot of the number of columns in A.
    #
    # numColumns: how many columns are there in the display.
    # Default value is the squareroot of the number of rows in A.

    # compute number of rows and columns
    nr,nc = np.shape(A)
    if imageWidth==None:
        sx = np.ceil(np.sqrt(nc))
        sy = np.ceil(nc/sx)
    else:
        sx = imageWidth
        sy = np.ceil(nc/sx)
    if numColumns==None:
        n = np.ceil(np.sqrt(nr))
        m = np.ceil(nr/n)
    if numColumns==None:
        n = np.ceil(np.sqrt(nr))
        m = np.ceil(nr/n)
    else:
        n = numColumns
        m = np.ceil(nr/n)
    n = np.uint8(n)
    m = np.uint8(m)
    if optNormAll:
        A = A-A.min()
        A = A/A.max()

    # insert data onto squares on the screen
    k = 0
    buf = borderWidth
    sy = int(sy)
    sx = int(sx)
    array = -np.ones([buf+m*(sy+buf),buf+n*(sx+buf)])
    for i in range(1,m+1):
        for j in range(1,n+1):
            if k>=nr:
                continue
            B = A[k,:]
            if optNormEach:
                B = B-B.min()
                B = B/float(B.max())
            B = np.reshape(np.concatenate((B,-np.ones(sx*sy-nc)),axis=0),(sx,-1))
            array[(buf+(i-1)*(sy+buf)):(i*(sy+buf)),(buf+(j-1)*(sx+buf)):(j*(sx+buf))] = B
            k = k+1

    # display picture and save it
    cmap = plt.cm.get_cmap(cmapName)
    cmap.set_under(color=borderColor)
    cmap.set_bad(color=borderColor)
    if graphicsLibrary=='PIL':
        im = PIL.Image.fromarray(np.uint8(cmap(array)*255))
        if verbose:
            im.show()
        if saveName!='':
            im.save(saveName)
    elif graphicsLibrary=='matplotlib':
        plt.imshow(array,interpolation='nearest',
                   norm=colors.Normalize(vmin=0.0,vmax=1.0,clip=False))
        plt.set_cmap(cmap)
        if verbose:
            plt.show()
        if saveName!='':
            plt.savefig(saveName)
```

```
## ----------------------------------------------------------------
class SparseAutoencoder():

    def __init__(self,visibleSize,hiddenSize=25,sparsityParam=0.01,beta=3,decayWeight=0.0001):
        # hyperparameters
        self.visibleSize = visibleSize      # number of input units
        self.hiddenSize = hiddenSize        # number of hidden units
        self.sparsityParam = sparsityParam  # desired average activation of hidden units
        self.beta = beta                    # weight of sparsity penalty term
        self.decayWeight = decayWeight      # weight decay parameter
        # parameters
        self.W1 = None  # array of shape (visibleSize, hiddenSize)
        self.W2 = None  # array of shape (hiddenSize, visibleSize)
        self.b1 = None  # vector of length (hiddenSize)
        self.b2 = None  # vector of length (visibleSize)

    def fit(self,patches,maxiter=400):
        # this function trains the weights of the sparse autoencoder.
        if (self.W1 is None or self.W2 is None or self.b1 is None or self.b2 is None):
            theta = initializeParameters(self.hiddenSize,self.visibleSize)
        else:
            theta = ravelParameters(self.W1,self.W2,self.b1,self.b2)
        sparseAutoencoderArgs = (self.visibleSize,self.hiddenSize,self.decayWeight,
                                 self.sparsityParam,self.beta,patches)
        opttheta,cost,messages = minimize(sparseAutoencoderCost,theta,fprime=None,
                                          args=sparseAutoencoderArgs,maxiter=400)
        self.W1,self.W2,self.b1,self.b2 = unravelParameters(opttheta,self.hiddenSize,self.visibleSize)

    def predict(self,samples):
        # this function returns the output layer activations (estimates)
        z2 = np.dot(samples,self.W1)+np.array([self.b1])
        a2 = sigmoid(z2)
        z3 = np.dot(a2,self.W2)+np.array([self.b2])
        a3 = sigmoid(z3)
        return a3

    def score(self,patches):
        # computes the cost function of the sparseAutoencoder
        theta = ravelParameters(self.W1,self.W2,self.b1,self.b2)
        return sparseAutoencoderCost(theta,self.visibleSize,self.hiddenSize,self.decayWeight,
                                     self.sparsityParams,self.beta,patches)[0]
```

In [24]:
```
if __name__ == "__main__":
    #  This function contains code that helps you get started on the
    #  programming assignment. You will need to complete the code in
    #  sampleIMAGES(), sparseAutoencoderCost() and computeNumericalGradient().
    #  For the purpose of completing the assignment, you do not need to
    #  change the code in this function.
    #
    ##======================================================================
    ## STEP 0: Here we provide the relevant parameters values that will
    #  allow your sparse autoencoder to get good filters; you do not need to
    #  change the parameters below.

    visibleSize = 8*8      # number of input units
    hiddenSize = 25        # number of hidden units
    sparsityParam = 0.01   # desired average activation of the hidden units.
                           # (This was denoted by the Greek alphabet rho, which
                           # looks like a lower-case "p" in the lecture notes).
    decayWeight = 0.0001   # weight decay parameter
    beta = 3               # weight of sparsity penalty term

    ##======================================================================
    ## STEP 1: Implement sampleIMAGES
    #
    #  After implementing sampleIMAGES, the display_network command should
    #  display a random sample of 200 patches from the dataset

    patches = sampleIMAGES()
    displayNetwork(patches[:100,:])

    #  Obtain random parameters theta
    theta = initializeParameters(hiddenSize, visibleSize);

#     ##======================================================================
#     ## STEP 2: Implement sparseAutoencoderCost
#     #
#     #  You can implement all of the components (squared error cost, weight decay term,
#     #  sparsity penalty) in the cost function at once, but it may be easier to do
#     #  it step-by-step and run gradient checking (see STEP 3) after each step.  We
#     #  suggest implementing the sparseAutoencoderCost function using the following steps:
#     #
#     #  (a) Implement forward propagation in your neural network, and implement the
#     #      squared error term of the cost function.  Implement backpropagation to
#     #      compute the derivatives.  Then (using decayWeight=beta=0), run Gradient Checking
#     #      to verify that the calculations corresponding to the squared error cost
#     #      term are correct.
#     #
#     #  (b) Add in the weight decay term (in both the cost function and the derivative
#     #      calculations), then re-run Gradient Checking to verify correctness.
#     #
#     #  (c) Add in the sparsity penalty term, then re-run Gradient Checking to
#     #      verify correctness.
```

```
    cost,grad = sparseAutoencoderCost(theta,visibleSize,hiddenSize,
                                      decayWeight,sparsityParam,beta,patches)

#     ##======================================================================
#     ## STEP 3: Gradient Checking
#     #
#     # Hint: If you are debugging your code, performing gradient checking on smaller models
#     # and smaller training sets (e.g., using only 10 training examples and 1-2 hidden
#     # units) may speed things up.

#     # First, lets make sure your numerical gradient computation is correct for a
#     # simple function.  After you have implemented computeNumericalGradient(),
#     # run the following:
    checkNumericalGradient()

#     # Now we can use it to check your cost function and derivative calculations
#     # for the sparse autoencoder.
    print ("Computing numerical gradient of sparseAutoencoderCost...")
    numgrad = computeNumericalGradient(lambda x: sparseAutoencoderCost(
        x,visibleSize,hiddenSize,decayWeight,sparsityParam,beta,patches)[0],theta)

# #     # Use this to visually compare the gradients side by side
    print (np.array([numgrad,grad]).T)

# #     # Compare numerically computed gradients with the ones obtained from backpropagation
    diff = norm(numgrad-grad)/norm(numgrad+grad)
    print (diff)  # Should be small. In our implementation, these values are
                 # usually less than 1e-9.

#                 # When you got this working, Congratulations!!!
```



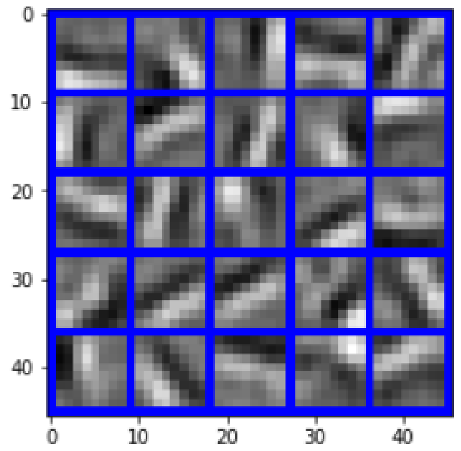
```
Computing gradient for input: 0
[[38. 38.]
 [12. 12.]]
The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)


2.1452381569477388e-12
Norm of the difference between numerical and analytical gradient (should be < 1e-9)


Computing numerical gradient of sparseAutoencoderCost...
Computing gradient for input: 0
Computing gradient for input: 100
Computing gradient for input: 200
Computing gradient for input: 300
Computing gradient for input: 400
Computing gradient for input: 500
Computing gradient for input: 600
Computing gradient for input: 700
Computing gradient for input: 800
Computing gradient for input: 900
Computing gradient for input: 1000
Computing gradient for input: 1100
Computing gradient for input: 1200
Computing gradient for input: 1300
Computing gradient for input: 1400
Computing gradient for input: 1500
Computing gradient for input: 1600
Computing gradient for input: 1700
Computing gradient for input: 1800
Computing gradient for input: 1900
Computing gradient for input: 2000
Computing gradient for input: 2100
Computing gradient for input: 2200
Computing gradient for input: 2300
Computing gradient for input: 2400
Computing gradient for input: 2500
Computing gradient for input: 2600
Computing gradient for input: 2700
Computing gradient for input: 2800
Computing gradient for input: 2900
Computing gradient for input: 3000
Computing gradient for input: 3100
Computing gradient for input: 3200
[[ 0.56324522  0.56324522]
 [ 0.75311979  0.75311979]
 [ 0.90874557  0.90874557]
 ...
 [-0.04312523 -0.04312523]
 [-0.03224107 -0.03224107]
 [-0.0349739  -0.0349739 ]]
7.453820821199506e-11
```

In [29]:
```
#     ##=====================================================================
#     ## STEP 4: After verifying that your implementation of
#     #  sparseAutoencoderCost() is correct, You can start training your sparse
#     #  autoencoder with scipy.optimize.fmin_l_bfgs_b (L-BFGS).

#     #  Randomly initialize the parameters
theta = initializeParameters(hiddenSize,visibleSize)

#     #  Use L-BFGS to minimize the function. Generally,
#     #  for scipy.optimize.fmin_l_bfgs_b to work, you
#     #  need a function with two outputs: the
#     #  function value and the gradient. In our problem,
#     #  sparseAutoencoderCost() satisfies this.
#     #  Here, we set the maximum number of iterations
#     #  of L-BFGS to run to be 400 (or until convergence).
opttheta,cost,messages=minimize(sparseAutoencoderCost,theta,fprime=None,maxiter=1000,
                                args=(visibleSize,hiddenSize,decayWeight,sparsityParam,beta,patches))
```

In [30]:
```
print(messages)
```

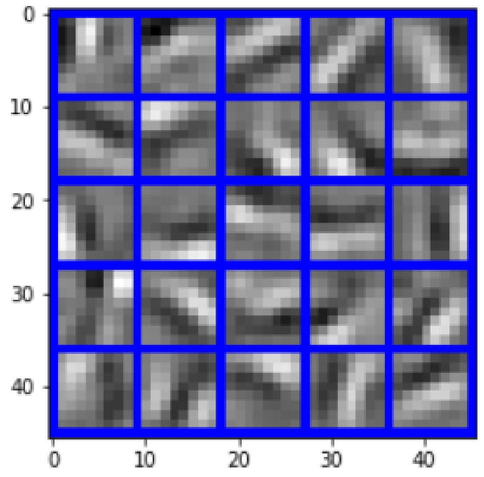
```
{'grad': array([ 1.42862776e-06,  1.78036898e-06,  7.25158881e-07, ...,
        3.34385046e-06,  2.19595489e-06, -3.45146318e-06]), 'task': b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL', 'funca
lls': 868, 'nit': 833, 'warnflag': 0}
```

In [32]:
```
#     ##=====================================================================
#     ## STEP 5: Visualization
#     # save the visualization to a file

W1,W2,b1,b2 = unravelParameters(opttheta,hiddenSize,visibleSize)
displayNetwork(W1.T,saveName='weights.png')
```



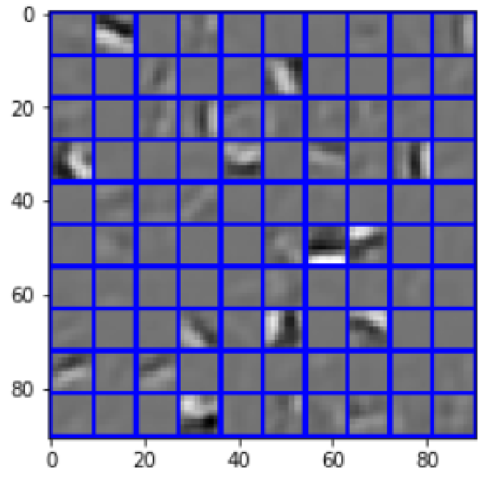
```
<Figure size 432x288 with 0 Axes>
```

In [34]:
```
#     ##==================================================================
#     ## STEP 6: Classes and objects
#     # bind the functions together into a SparseAutoencoder class
sae = SparseAutoencoder(visibleSize=64,hiddenSize=25,sparsityParam=0.01,
                        beta=3,decayWeight=0.0001)
sae.fit(patches,maxiter=2000)
displayNetwork(sae.W1.T,saveName='weights_2000.png')
```



```
<Figure size 432x288 with 0 Axes>
```

In [35]:
```
estimates = sae.predict(patches[:100,:])
displayNetwork(estimates)
```



0
20
40
60
80
0
20
40
60
80