1. **Problem 1 (1 point):** Warmup: Load and visualize MNIST:

   You will need the files `visualize.py` and `load_MNIST.py` in the *same* directory. Execute `train_autoencoder.py` to load and visualize the MNIST dataset. Currently the code loads 100 images into `patches_train` and plots them (in Step 0).

   Modify the script `train_autoencoder.py` so you plot the first 10, 50 and 100 `patches_train`. The function `plot_images` takes an optional filepath (string), which when specified will save the .png image to the filepath. Include in your written solution a figure for each of the three subsets (each with a caption!). Also plot the 100 `patches_test`. These will be used later to qualitatively assess how well your trained autoencoder does in encoding and decoding images it has not been trained on – you'll want to see these original images in order to compare against what your autoencoder produces.

   Figure 1 depicts the first 10, 50, and 100 images in `patches_train` along with the 100 images included in `patches_test`. Below is an excerpt of `train_autoencoder.py`. As with all the code included in this write-up, most of the comments in the code have been excluded to conserve space.

```
for num in [10,50,100]:
    visualize.plot_images(patches_train[:, 0:num],
                          filepath='patches_train{:03d}.png'.format(num))

visualize.plot_images(patches_test[:, 0:100],
                      filepath='patches_test100.png')
```

2. **Problem 2 (3 pts):**

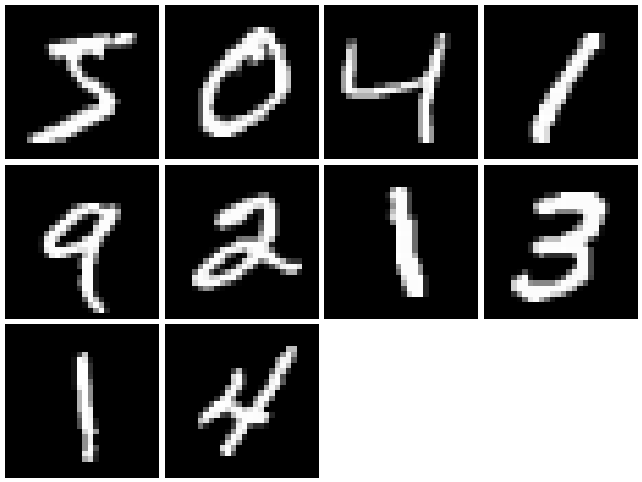
   Write the initialization script for the parameters in an autoencoder with a single hidden layer.

   You will implement this functionality in the function `initialize` in `utils.py` (found in the script `train_autoencoder.py`, this exercise comprises Step 2).

   In class we learned that a NN's parameters θ are the weights $W_{ij}^{(l)}$ and the offset (bias) parameters $b_i^{(l)}$. Write the script so that you initialize them given the size of the hidden layer and the visible layer. Then reshape them and concatenate them so they are all allocated in a single parameter vector.

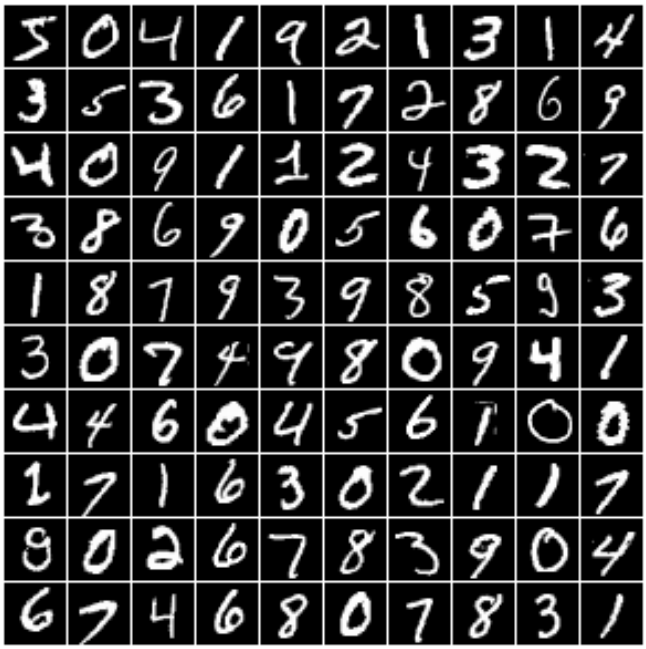
   Example: For an autoencoder with visible layer size 2 and hidden size 3, we would have 6 weights from the visible layer to the hidden layer (comprising the parameters in the weight matrix at layer 1: $W^{(1)}$) and 6 more weights from the hidden layer to the output layer (comprising the parameters in the weight matrix at layer 2: $W^{(2)}$). There are also vectors of bias weights, one for layer 1 ($b^{(1)}$) with one bias weight parameter for each of the hidden nodes (3 bias weights), and another for layer 2 ($b^{(2)}$) with 2 bias weight parameters for each node at the output layer. This will make a total of 6+6+3+2 = 17 parameters. The output of your initialize function should be an array of 17 elements, with order [$\{W^{(1)}\}$, $\{W^{(2)}\}$, $\{b^{(1)}\}$, $\{b^{(2)}\}$].

   Tip: You can use the `np.concatenate` function to concatenate arrays in the desired order.



(a) 1st 10 Images in Training Set

(b) 1st 50 Images in Training Set



(c) 1st 100 Images in Training Set

(d) 1st 100 Images in Testing Set

Figure 1: The images in the training and testing sets.

In the code attached below, I used the uniform initialization scheme from Glorot (2010a) to determine the bounds r and $-r$ for sampling the weight parameters w_1 and w_2. Unlike the bias parameters b_1 and b_2, which are initialized with 0's, we should initialize the weight parameters to (1) break symmetry by ensuring all weights are randomly initialized (as opposed to assigning a uniform value across all weights) and to (2) ensure that the hidden units receive some signal by assigning them a non-zero weight.

```
def r(fan_in, fan_out):
    return np.sqrt(6/(fan_in + fan_out + 1))

def initialize(hidden_size, visible_size):
    w1 = np.random.uniform(low = -r(visible_size, hidden_size),
                           high = r(visible_size, hidden_size),
                           size = visible_size * hidden_size)
    w2 = np.random.uniform(low = -r(hidden_size, visible_size),
                           high = r(hidden_size, visible_size),
                           size = hidden_size * visible_size)
    b1 = np.zeros((hidden_size))
    b2 = np.zeros((visible_size))

    theta = np.concatenate((w1,w2,b1,b2))
    return theta
```

3. **Problem 3 (14 points):**

Implement a function for the cost of a 3-layer perceptron (which includes the case of an autoencoder) as well as the gradient for each of the parameters.

In this exercise you will work on implementing two functions: `autoencoder_cost_and_grad()` in `utils.py` and `compute_gradient_numerical_estimate()` in `gradient.py`. (In the script `train_autoencoder.py`, this exercise comprises Steps 3 and 4.)

In class we learned that we can use gradient descent to train a multi-layer perceptron NN using Backpropagation. In this exercise we will implement the core of the Backpropagation computation. However, we will use a more refined version of gradient descent called L-BFGS-B (`http://en.wikipedia.org/wiki/Limited-memory_BFGS`), which is readily implemented in the optimization library of scipy:

```
scipy.optimize.minimize(..., method='L-BFGS-B', ...)
```

For convenience, the `train_autoencoder.py` script is already set up to run `scipy.optimize.minimize` later.

To obtain the information needed to use `scipy.optimize.minimize`, you need to implement `autoencoder_cost_and_grad()`, which will compute both the `cost` (i.e., loss) and `grad` (gradient) (Step 3). `autoencoder_cost_and_grad()` will use the data to compute the forward pass resulting in the network output, calculate the overall error (`cost`), and then calculate the gradient (`grad`) for each parameter using the error backpropagation of Backprop Core. Note that you will need to extract from the `theta` array the weights and bias parameters for each layer, as you constructed them in the `initialize` function in Exercise 2. You will likely want to use the numpy `reshape` method.

The gradient array has to be the same size as the theta array (again, you'll need to construct this following the same parameter indexing you used for `theta`), while the cost is a scalar representing the difference between the network output and training target.

You will then implement the numerical gradient estimate in `gradient.compute_gradient_numerical_estimate()`, using the EPSILON numerical gradient estimate error function. Remember that for each parameter in the `theta` array, you'll compute the estimated gradient of the objective function with respect to that parameter by varying just that parameter a little (+/- EPSILON) while keeping the other parameters constant. EPSILON is set to 0.0001, and, as discussed in lecture, it is expected that the difference between the numerically estimated gradient and your gradient calculation in `autoencoder_cost_and_grad` will be very small (in my implementation the difference is around 10^{-9}).

To perform the test, set the `RUN_STEP_4_DEBUG_GRADIENT` parameter on line 18 of `train_autoencoder.py` to `True`; now when you run the script, it will run debugging code to check if your gradient is correct. You might want to load fewer images in this step (i.e., select fewer columns in the `patches` variable, say 10), and also reduce the hidden layer size (to say 2), so that you do not spend too much time waiting for all of the images to be processed (making these restrictions will still allow you to assess whether your gradient computation is correct); just remember to set you parameters back to their other values when you proceed to the next exercises.

All equations used to compute the cost function and its gradients can be found in Lecture 24 notes. The Cost function (also known as the loss function) is composed of the squared error term and a weight decay term, such that

$$J(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(||h_\theta(x^{(i)}) - y^{(i)}||\right)^2\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2, \tag{1}$$

where m is the number of training examples and h_θ is some function of parameters θ, λ is a regularization constant, W_{ji} is the weight matrix.
The partial derivatives of the cost function used for gradient descent are:

$$\frac{\partial J(w,b)}{\partial W_{ij}^{(l)}} = \left[\frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial W_{ij}^{(l)}}\right] + \lambda W_{ij}^{l}. \tag{2}$$

$$\frac{\partial J(w,b)}{\partial b_i^{(l)}} = \frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial b_i^{(l)}}. \tag{3}$$

To compute the partial derivatives within the summation, we must first define $\delta_i^{n_l}$. For each output unit i in the output layer n_l,

$$\delta_i^{n_l} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}). \tag{4}$$

For each node l in layer l, such that $l = \{n_l - 1, n_l - 2, ..., 3, 2\}$,

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_l+1} W_{ji}^{(l)}\delta_j^{(l+1)} f'(z_i^{(l)})\right). \tag{5}$$

The partial derivatives in the summation from Equations (2) and (3) are computed as:

$$\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \tag{6}$$

$$\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}} = \delta_i^{(l+1)} \tag{7}$$

Equations (1), (2), and (3) have been encoded to `utils.py`.
The numerical gradient uses the symmetric difference quotient to estimate the slope:

$$J'(\theta) \approx \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}, \tag{8}$$

where $J'(\theta)$ is the slope of the cost function, J is the cost function, θ is a vector of weight and bias parameters, and ϵ is the small perturbation introduced in the cost function.
When running the debugger, (1) the implementation difference between the numerical gradient and true gradient and (2) the accuracy of the numerical gradient were both on the order of 10^{-9} to 10^{-10}, respectively.

```
def compute_gradient_numerical_estimate(J, theta, epsilon=0.0001):
    gradient = numpy.zeros(theta.shape)

    for num, elem in enumerate(theta):

        theta[num] = elem + epsilon
        cost_plus  = J(theta)[0]

        theta[num] = elem - epsilon
        cost_minus = J(theta)[0]

        theta[num] = elem
        gradient[num] = (cost_plus - cost_minus) / (2 * epsilon)

    return gradient

#---------------------------------------------------------------------#

def autoencoder_cost_and_grad(theta, visible_size, hidden_size, lambda_, data)
    m   = data.shape[1]
    len = visible_size * hidden_size

    w1 = theta[0 : len].reshape((hidden_size, visible_size))        # (h,v)
    w2 = theta[len : 2*len].reshape((visible_size, hidden_size))    # (v,h)
    b1 = theta[2*len : 2*len + hidden_size].flatten()               # (h)
    b2 = theta[2*len + hidden_size: ].flatten()                     # (v)
```

```
    # FORWARD PROPAGATION (Lecture 24, Slides 11-13)
    # HW5 #4: Vectorized Implementation of Forward Propagation
    # Moved to autoencoder_feedforward(theta, visible_size, hidden_size, data)

    tau = autoencoder_feedforward(theta, visible_size, hidden_size, data)
    z2  = tau[0 : hidden_size]
    a2  = tau[hidden_size : 2*hidden_size]
    z3  = tau[2*hidden_size : 2*hidden_size + visible_size]
    h   = tau[2* hidden_size + visible_size:]


    # COST FUNCTION (Equation on Lecture 24, Slide 15)
    # J(W,b) = Squared Error Term + Weight Decay Term (for regularization)

    cost = np.linalg.norm(h - data)**2/2./m +
           lambda_/2. * (np.linalg.norm(w1)**2 + np.linalg.norm(w2)**2)


    # BACKPROPAGATION (Lecture 24, Slides 15-16)
    # Step 1: Perform feedforward pass,
                  Computing activations for layers L_{2...n}.
    # Step 2: Compute "error terms." (Slide 16)
    # Step 3: Compute partial derivatives. (Slide 15)

    w1_grad = delta2.dot(data.T) / m + lambda_ * w1
    w2_grad = delta3.dot(a2.T) / m + lambda_ * w2
    b1_grad = np.sum(delta2, axis=1) /m
    b2_grad = np.sum(delta3, axis=1) / m


    grad = np.concatenate((w1_grad.flatten(), w2_grad.flatten(),
                           b1_grad, b2_grad))

    return cost, grad
```

4. **Problem 4 (2 points):**

   Implement feedforward as a stand-alone function `autoencoder_feedforward()` in `utils.py`

   In the previous exercise it was necessary to implement a single function that computes the cost and gradient of your autoencoder so that we can use the fancy `scipy.optimize.minimize` optimizer. In the service of doing that, you had to implement the feedforward computation, where given an input you calculate the output activations at the output (visible) layer of your autoencoder. In this exercise, you need to separate this functionality out into a standalone function so that you can visualize what your autoencoder is reproducing at the output layer given an input image. We *could* actually have this function be called as part of your implementation of `autoencoder_cost_and_grad()`, but I have chosen to make this a separate, standalone function because likely you want your implementation of the feedforward step in `autoencoder_cost_and_grad()` to be as efficient as possible and I didn't want to intro-

duce the extra constraint that feedforward in that implementation also had to be standalone (among other things, keeping track of the layer activations in feedforward allows them to be reused during backpropagation).

To implement stand-alone feedforward, all you need to do is copy the code you wrote in `autoencoder_cost_and_grad()` into the provided `autoencoder_feedforward()` stub in `utils.py` and make it so it returns a matrix of activations. Just like `autoencoder_cost_and_grad()`, this function takes a matrix where each column is a column vector representing an "unrolled" image patch, and there are 1 or more columns. The return matrix of `output_activations` will have the same format, but now the columns represent the output activations corresponding to the input patches. If you implemented your feedforward computation in `autoencoder_cost_and_grad()` so that it computes feedforward for each patch individually, that's fine – now just take each output activation as a column vector and concatenate the column vectors into a matrix. On the other hand, if you've already vectorized your feedforward computation, then you're likely done!

The output of this function will then be used to display the activations as images, each corresponding to the first 100 patches in the image data. This code will be used in the next exercise, after you've trained your autoencoder!

In forward propagation, we apply the following equations:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)] \tag{9a}$$
$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{9b}$$
$$a^{(2)} = f(z^{(2)}) \tag{9c}$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{9d}$$
$$h_{W,b}(x) = a^{(a)} = f(z^3), \tag{9e}$$

where activation function f is the Sigmoid function, $W_{i,j}^{(l)}$ is the weight between unit j in layer l and unit i in layer $l+1$, $b_i^{(l)}$ is the bias for unit i in layer $l+1$, and $a_i^{(l)}$ is the activation (output value) of unit i in layer l.

```
def autoencoder_feedforward(theta, visible_size, hidden_size, data):
    m   = data.shape[1]
    len = visible_size * hidden_size

    w1 = theta[0 : len].reshape((hidden_size, visible_size))       # (h,v)
    w2 = theta[len : 2*len].reshape((visible_size, hidden_size))   # (v,h)
    b1 = theta[2*len : 2*len + hidden_size].flatten()              # (h)
    b2 = theta[2*len + hidden_size: ].flatten()                    # (v)


    # FORWARD PROPAGATION (Lecture 24, Slides 11-13)
    z2 =  w1.dot(data) + np.repeat(b1,m).reshape((b1.shape[0],m)) # (h,m)
    a2 =  sigmoid(z2)                                             # (h,m)
    z3 =  w2.dot(a2) + np.repeat(b2,m).reshape((b2.shape[0],m))   # (v,m)
    h  =  sigmoid(z3)                                             # (v,m)

    return np.concatenate((z2, a2, z3, h))
```

5. **Problem 5 (2 points):** Use your autoencoder!

   If your gradient, as tested in Exercise 3, is sufficiently close to the numerical estimate, now you can train your autoencoder on the `patches_train`. Keep the weight decay term, `lambda_`, set to 0.0001.

   Train your autoencoder with different sizes of hidden layer. In particular, run the `train_autoencoder.py` script with hidden layers of (a) 10, (b) 50, (c) 100 and (d) 250. Try two different training sizes: the first 100 patches (the initial setting in the script), and also 1000 patches.

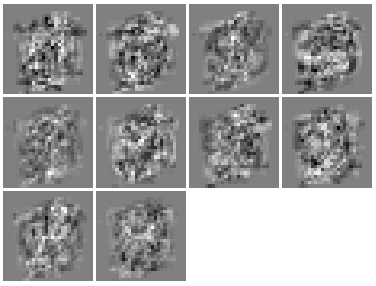
   Be aware that it will take a couple of minutes to run each training round, with the amount of time increasing as the number of hidden states increases (more parameters!). Not surprisingly, training with 1000 training patches (as opposed to 100) will take 10-times as long to train.

   Also note that `scipy.optimize.minimize` may report some errors or failures (possibly often). This is generally OK; the only condition that is more concerning is when the 'number of iterations' (reported in the verbose output of the `scipy.optimize.minimize` function, after it has completed running) is very low, say less than 20, and the 'message' is 'ABNORMAL_TERMINATION_IN_LNSRCH'. In this case, it means there is some inherent instability that prevented the optimizer from updating any useful amount. This could be due to unlucky weight initialization, but more likely due to parameters interacting (this will be more prevalent in Exercise 6, when you implement the Sparse Autoencoder and try different `rho_` and `beta_` parameters). When you get this, try running a couple more times (since it is only completing a few iterations, these runs don't take long) – if you keep getting the same result, then it is safe to conclude that the parameters you're using are generally not conducive to training; just note this and move on to exploring other values.

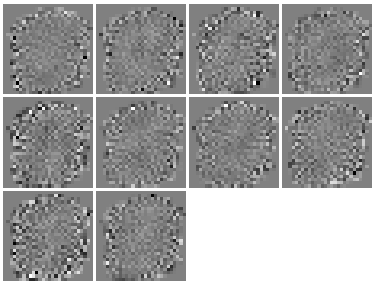
   After each run, code is in place in `train_autoencoder.py` to call `utils.plot_and_save_results`; as described in the initial instructions, this will save your model parameters, generate the layer 1 weight images, and also call `utils.autoencoder_feedforward()` with both the first 100 training patches and the 100 test patches to see what your autoencoder reproduces. The code currently defaults to save this using the same root pathname; you likely want to change the root pathname to something different for each run with different parameter settings, otherwise the output will be overwritten.

   Run each architecture (hidden layer sizes at 10, 50, 100 and 250), both for the 100 training patches, and the first 1000 training patches. Include in your writeup the layer 1 weight images and train and test decode images. Describe (in the image captions) the structure, if any, that you observe in each run. Do you observe any trend or changes in the weight patterns as you change the hidden layer size? How about the output activations for training and for testing? How closely do they resemble the corresponding input patches?

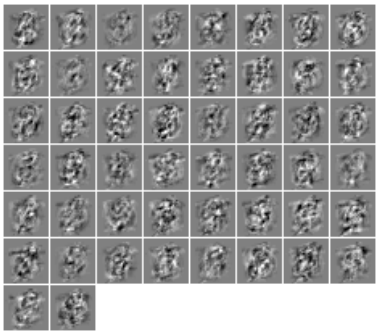
   Figure 3 depicts the images weights for corresponding training set size and hidden layer size as denoted. When keeping the number of hidden layers constant, increasing the training set size appears to wash out any features in the data set. This disparity is most easily seen in the comparison between 100 and 1000 image training set for 50 hidden layers autoencoder. This disparity between training set sizes diminishes with more hidden layers. Furthermore, a less noticeable effect of increasing the size of the autoencoder is that structure appears to be subtly increasing. Overall, this autoencoder does a poor job at edge detection in its current state.



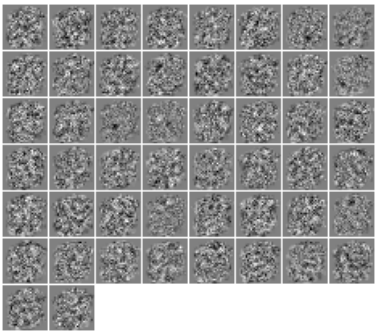
(a) 100 images & 10 hidden layers



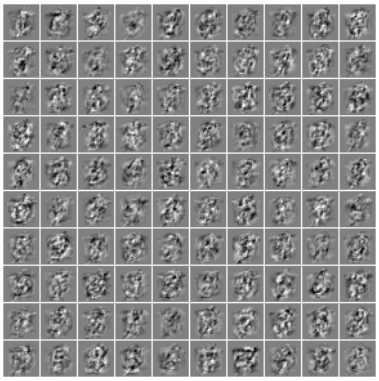
(b) 1000 images & 10 hidden layers



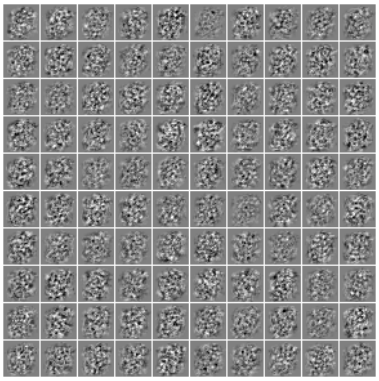
(c) 100 images & 50 hidden layers



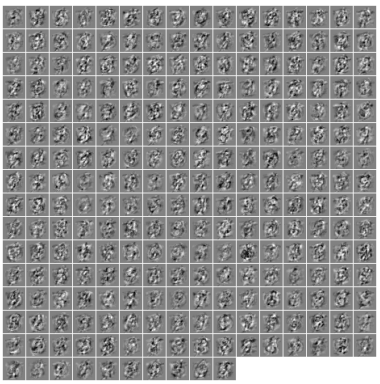
(d) 1000 images & 50 hidden layers



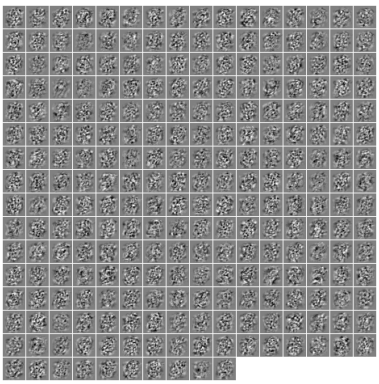
(e) 100 images & 100 hidden layers



(f) 1000 images & 100 hidden layers



(g) 100 images & 250 hidden layers



(h) 1000 images & 250 hidden layers

Figure 2: The images above depict the weights for the corresponding number of training patches and hidden layers as denoted. In general, this autoencoder performs edge detection poorly. For a given number of hidden layers, the training set size appears to negatively affect the performance of our weights, with more structure in smaller training sets. Given a constant number of training examples, more structure is observed with more hidden layers.

6. **Problem 6 (8 points):** Sparse Autoencoder

   In this exercise you will now implement the function `autoencoder_cost_and_grad_sparse()` in `utils.py`. For this, you will implement the sparsity penalty term as described in:
   `http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity`
   Most of the backpropagation algorithm you implemented in `autoencoder_cost_and_grad()` will remain exactly the same: you can copy your code from `autoencoder_cost_and_grad()` to use as your starting point for `autoencoder_cost_and_grad_sparse()`. Then follow the modifications describe in the above url (and covered in class) to add the sparsity penalty term to the cost and gradient computations.

   You can test your implementation by modifying the call under `RUN_STEP_3` (line 137) to call `utils.autoencoder_cost_and_grad_sparse`, and then again setting `RUN_STEP_3` and `RUN_STEP_4_DEBUG_GRADIENT` to `True` to test the that the gradient is still being computed properly.

   With the added sparsity constraint, under certain network architectures you should now be able to induce more patterns in the first layer weight activations.

   Modify `train_autoencoder.py` Step 5 so that it now uses `autoencoder_cost_and_grad_sparse()` to compute the new sparse-penalized cost and grad.

   Note that `autoencoder_cost_and_grad_sparse()` includes two new parameters: ρ (`rho_`), which determines the hidden layer activation limits imposed by the penalty, and β (`beta_`), which governs that amount of penalty applied relative to the regular cost function.

   You will train using the first 100 patches, and as in Exercise 5, you will keep `lambda_` at 0.0001 and try each of the three different hidden layer sizes: 10, 50, 100. You will also need to experiment with values for `rho_` and `beta_`. As a hint, try the following:

   For `hidden_size` = 10, try `beta_ = 0.2` with `rho_` = {0.05, 0.01, 0.005}
   For `hidden_size` = 50, try `beta_ = 0.1` with `rho_` = {0.05, 0.01, 0.005}
   For `hidden_size` = 100, try `beta_ = 0.01` with `rho_` = {0.05, 0.01, 0.005}
   Note that as you increase the number of nodes in the hidden layer, it is more effective to decrease `beta_`, which makes sense since as there are more hidden nodes, there collective impact becomes larger, so need to be discounted more.

   As in Exercise 5, report the patterns you see in both the weights as well as the output train and test decoding. Describe what you see in the images. Do you observe any general trends as you change the hidden layer size and corresponding sparsity parameters? Explain how this compares to the images in exercise 5.

The sparse autoencoder adds a sparsity constraint on the hidden units. We define the sparsity parameter as

$$\rho = \widehat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^(i))\right], \tag{10}$$

the average activation of hidden unit j averaged over the training set. ρ is typically a small value close to 0, thereby forcing a_j (the activations of hidden units j) to be nearly inactively with a value close to 0. This results in a penalized cost function of

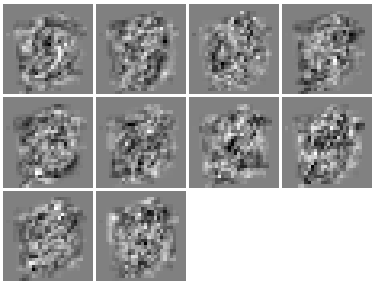
$$J_{\text{sparse}}(W,b) = J(W,b) + \left[\beta\sum_{j=1}^{s_2}\rho\log\frac{\rho}{\widehat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\widehat{\rho}_j}\right], \tag{11}$$

where β is the weight of the sparsity term and the bracketed term is the Kullback-Leibler (KL) divergence between two Bernoulli random variables with means ρ and $\widehat{\rho}_j$. The KL divergence measures the differences of two distributions. Futhermore, $\delta_i^{(2)}$ must be updated to include the KL divergence term, such that

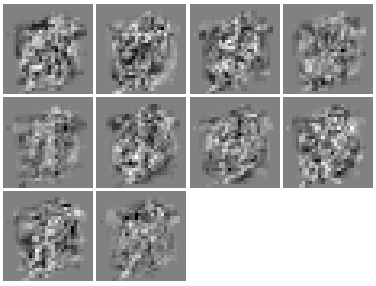
$$\delta_i^{(2)} = \left[\left(\sum_{j=1}^{s_2} W_{ji}^{(2)}\delta_j^{(3)}\right) + \beta\left(-\frac{\rho}{\widehat{\rho}_i} + \frac{1-\rho}{1-\widehat{\rho}_i}\right)\right] f'(z_i^{(2)}). \tag{12}$$

When running the sparse autoencoder, the difference between the true gradient and the numerical approximation is roughly few $\times\ 10^{-8}$. As expected, the most notable difference between the original and sparse versions is the amount of structure the sparse version is able to reconstruct, given the same initial parameters. This structure becomes more well-defined as we increase the number of hidden units in the sparse autoencoder. As we further restrict the activations by decreasing the sparsity parameter ρ, the structure subtly appears to be more well-defined.

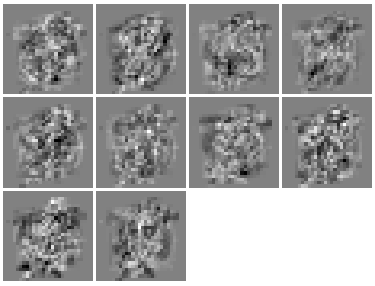
```
def autoencoder_cost_and_grad_sparse(theta, visible_size, hidden_size,
                                     lambda_, rho_, beta_, data):
    m   = data.shape[1]
    len = visible_size * hidden_size

    w1 = theta[0 : len].reshape((hidden_size, visible_size))        # (h,v)
    w2 = theta[len : 2*len].reshape((visible_size, hidden_size))    # (v,h)
    b1 = theta[2*len : 2*len + hidden_size].flatten()               # (h)
    b2 = theta[2*len + hidden_size: ].flatten()                     # (v)


    # FORWARD PROPAGATION (Lecture 24, Slides 11-13)
    tau = autoencoder_feedforward(theta, visible_size, hidden_size, data)
    z2  = tau[0 : hidden_size]                                      # (h,m)
    a2  = tau[hidden_size : 2*hidden_size]                          # (h,m)
    z3  = tau[2*hidden_size : 2*hidden_size + visible_size]         # (v,m)
    h   = tau[2* hidden_size + visible_size:]                       # (v,m)
```



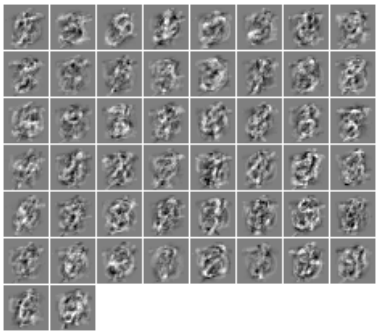
(a) 10 hidden layers with $\beta = 0.2$ and $\rho = 0.05$



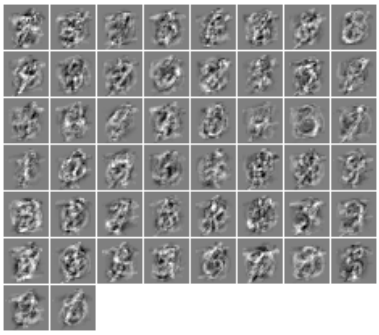
(b) 10 hidden layers with $\beta = 0.2$ and $\rho = 0.01$



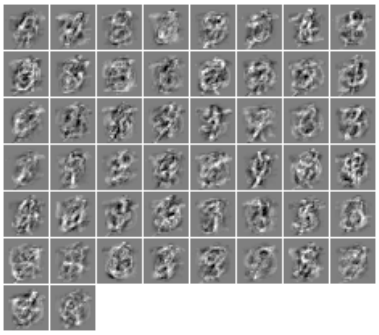
(c) 10 hidden layers with $\beta = 0.1$ and $\rho = 0.005$



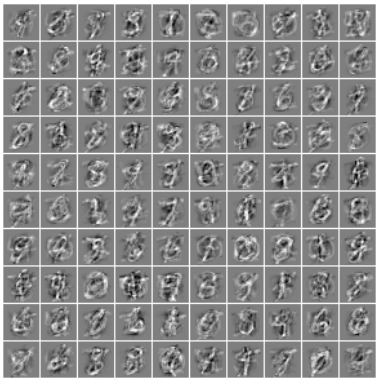
(d) 50 hidden layers with $\beta = 0.1$ and $\rho = 0.05$



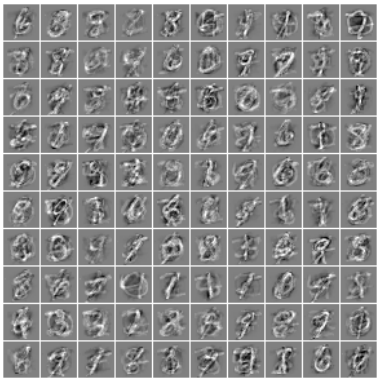
(e) 50 hidden layers with $\beta = 0.1$ and $\rho = 0.01$



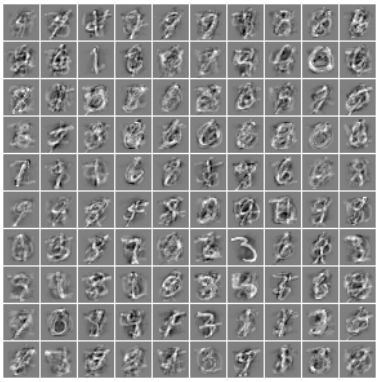
(f) 50 hidden layers with $\beta = 0.1$ and $\rho = 0.005$



(g) 100 hidden layers with $\beta = 0.1$ and $\rho = 0.05$



(h) 100 hidden layers with $\beta = 0.1$ and $\rho = 0.01$



(i) 100 hidden layers with $\beta = 0.1$ and $\rho = 0.005$

Figure 3: The images above depict the weights for the corresponding hidden layers and sparsity parameters as denoted. In contrast to the performance of the original autoencoder, the performance of the sparse autoencoder drastically increases with the number of hidden layers: as the number of hidden layers increase, the structure observed in the data becomes more defined. Varying the sparsity parameter appears to affect the results minimally, with greater definition observed in the results from the smaller sparsity parameters.

```
# COST FUNCTION (Equation on Lecture 24, Slide 15)
cost = np.linalg.norm(h - data)**2/2./m +
       lambda_/2. * (np.linalg.norm(w1)**2 + np.linalg.norm(w2)**2)

#-------------------------------------------------------------#

# SPARSITY PARAMETER
# http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity

rhat = np.sum(a2, axis=1) / m

# Kullback-Leibler (KL) Divergence
kl = np.sum(rho_ * np.log(rho_/rhat) +
            (1-rho_) * np.log((1-rho_)/(1-rhat)))
cost += beta_ * kl

#-------------------------------------------------------------#

# BACKPROPAGATION (Lecture 24, Slides 15-16)
# Step 1: Perform feedforward pass,
          Computing activations for layers L_{2...n}.
# Step 2: Compute "error terms." (Slide 16)
#         Use original equation for delta3.
#         Use modified version for delta2.

sparsity_der = beta_ * (-rho_/rhat + (1-rho_)/(1-rhat))
delta3 = -(data - h) * derivative(z3)
delta2 = (w2.T.dot(delta3) +
          np.repeat(sparsity_der,m).reshape((
          sparsity_der.shape[0],m))) * derivative(z2)

#-------------------------------------------------------------#

# Step 3: Compute partial derivatives. (Slide 15)
w1_grad = delta2.dot(data.T) / m + lambda_ * w1
w2_grad = delta3.dot(a2.T) / m + lambda_ * w2
b1_grad = np.sum(delta2, axis=1) / m
b2_grad = np.sum(delta3, axis=1) / m

grad = np.concatenate((w1_grad.flatten(), w2_grad.flatten(),
                       b1_grad, b2_grad))

return cost, grad
```