ISTA 421/521 – Homework 4

Due: Wednesday, December 7, 5pm

26 points total

Shuo Yang

Graduate

Instructions

In this assignment you will work directly with the following three python scripts: `utils.py`, `train_autoecoder.py`, and `gradient.py`. You will fill in portions of these files that are currently unimplemented, following the instructions in the exercises, and you will include all three in your final submission.

Also included in the released code are the following auxiliary files, which should appear in the same directory: `load_MNIST.py` and `visualize.py`. You will not make any changes to these files.

In this assignment, we will use two databases from the MNIST dataset, which you can get here:

- Training Images: `http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz`
  Uncompresses to 45MB
- Training Labels: `http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz`
  Uncompresses to 60K

The top-level script to run is `train_autoencoder.py`. Initially, many parts of the script will not yet run as the functions they call are not yet implemented. On line 54 there is the command `sys.exit()`, which will gracefully exit the script at that point. As you implement more of the assignment, you can move this command further down the script file. When you have completed everything, you can remove the command entirely.

We have provided the function `load_MNIST_images` in `load_MNIST.py` to load these files into memory. Be sure to adjust the filepaths so that they point to where you have saved the MNIST data.

Some problems require you to include plots of the weights and output that you learn with the autoencoder. The helper function `utils.plot_and_save_results` is provided to help with saving the model (the theta values and model parameters), extracts the first layer weights and plots them (each patch in the resulting grid image corresponds to the weights associated with each hidden node), and also feeds forward the first 100 training patches as well as 100 patches the network has not seen before and generate a plot of what the autoencoder is able to decode for these patches. In order to use the function you will need to implement the functionality needed to train a network as well as the stand-alone `utils.autoencoder_feedforward` function, which you will complete in Exercise 4.

These problems are adapted from the UFDL demo/tutorial from Stanford:
`http://ufldl.stanford.edu/wiki/index.php/Neural_Networks`.
You can follow the tutorial to walk you through your implementation.

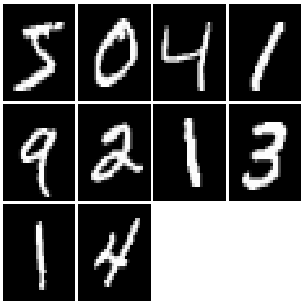
NOTE: You will see that in the code the steps of this tutorial are numbered, to indicate the steps of implementing the NN framework; these steps do not necessarily correspond to the exercise numbers in this document. It should be clear from the exercises below which part of the code you'll be working on.

1. [1 points] Exercise 1: 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.

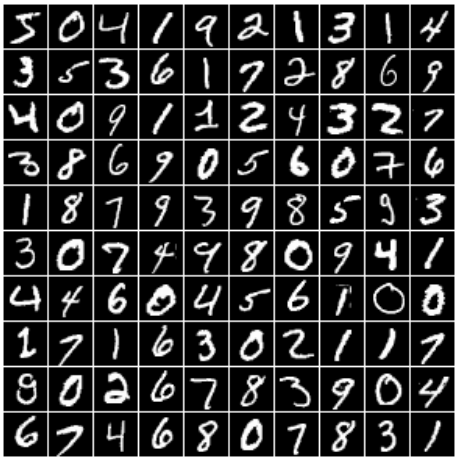
   Modify the script `train_autoencoder.py` so you plot the first 10, 50 and 100 `patches_train`. The function `plot_images` takes an optional filepath (string), which when specified will save the .png image to the filepath. Include in your written solution a figure for each of the three subsets (with caption!). Also plot the 100 `patches_test`. These will be used later to qualitatively assess how well you 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 again what your autoencoder produces.

   Solution: The figure for each of the three subsets of the training images is shown in Figure 1. The figure for the 100 test images is shown in Figure 2.



(a) First 10 image patch

(b) First 50 image patch



(c) First 100 image patch

Figure 1: Problem 1 - Plots of the first 10, 50 and 100 image patches

Figure 2: Problem 1 - Plot of 100 test image patch

2. [3 points] Exercise 2: 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`. (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 a script where 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, 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.

   Solution: The code snippet for initialization is shown below:

```
def initialize(hidden_size, visible_size):
    r = math.sqrt(6. / (hidden_size + visible_size + 1))
    w_l1 = numpy.random.uniform(-r, r, size=hidden_size*visible_size)
    b_l1 = numpy.zeros(hidden_size, dtype=numpy.float64)
    w_l2 = numpy.random.uniform(-r, r, size=hidden_size*visible_size)
    b_l2 = numpy.zeros(visible_size, dtype=numpy.float64)
    theta = numpy.concatenate((w_l1, w_l2, b_l1, b_l2))

    return theta
```

3. [14 points] Exercise 3: Write the cost function for 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, a you constructed them in the `initialize` function you constructed in Exercise 2. You will likely want to use the numpy `reshape` method (as we discussed in class).

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, note that on line 30 of `train_autoencoder.py` there is a boolean flag called `DEBUG`; if you set to `True`, 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).

Solution: The required functions are fully implemented.

To debug the gradient computation, I set hidden layer size to 4 and input image patches to 10. Below is the output of running debugging code:

```
dhcp-10-134-225-165:hw4 shuoyang$ python train_autoencoder.py
========== DEBUG: checking gradient ==========
test_compute_gradient_numerical_estimate(): Start Test
    Testing that your implementation of
        compute_gradient_numerical_estimate()
        is correct
    Computing the numerical and actual gradient for 'simple_quadratic_function'
    The following two 2d arrays should be very similar:
```

```
          [ 38.  12.] [ 38.  12.]
    (Left: numerical gradient estimate; Right: analytical gradient)
    Norm of the difference between numerical and analytical num_grad:
          1.70974269866e-10
    (should be < 1.0e-09 ; I get about 1.7e-10)
test_compute_gradient_numerical_estimate(): DONE

Now test autoencoder_cost_and_grad() gradient against numerical estimate:
    Total number of parameters, theta.shape= (7060,)
Norm of the difference between numerical and autoencoder_cost_and_grad gradients:
     3.46753174214e-08
(should be at least < 1.0e-07)
Passed gradient check!
========== DEBUG: checking gradient DONE ==========
```

This shows that my gradient computation is implemented correctly.

4. [2 points] Exercise 4: 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 introduce 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). In this exercise, you'll now do this. 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!

   Solution. The required functions are fully implemented. The code snippet for feedforward is shown below:

```
def autoencoder_feedforward(theta, visible_size, hidden_size, data):
  ## convert theta into matrix/vector format for weights and biases

  # each row of w_l1 is a vector of weights associate each unit (input feature)
  # in layer 1 to an unit at layer 2 (hidden layer)
  w_l1 = theta[0 : hidden_size*visible_size].reshape(hidden_size, visible_size)
  # each row of w_l2 is a vector of weights associate each unit
  # in layer 2 to an unit in layer 3 (output layer)
  w_l2 = theta[hidden_size*visible_size :
    2*hidden_size*visible_size].reshape(visible_size, hidden_size)

  # bias term in layer 1 associated with units in layer 2
  b_l1 = theta[2*hidden_size*visible_size : 2*hidden_size*visible_size + hidden_size]
  # bias term in layer 2 associated with units in layer 3
  b_l2 = theta[2*hidden_size*visible_size + hidden_size : ]

  m = data.shape[1] # number of training set

  ## compute feedforward

  a_l1 = data # activation at the input layer is the input data itself
  # weighted sum of inputs to layer 2
  z_l2 = w_l1.dot(a_l1) + numpy.tile(b_l1, (m, 1)).transpose()
  a_l2 = sigmoid(z_l2) # activation at the hidden layer (layer 2)
  # weighted sum of inputs to layer 2
  z_l3 = w_l2.dot(a_l2) + numpy.tile(b_l2, (m, 1)).transpose()
  a_l3 = sigmoid(z_l3) # activation at the output layer (layer 3)

  output_activations = a_l3

  return output_activations
```

5. [2 points] Exercise 5: 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 use are generally not conducive to training; just note this and move on to exploring other values.

   After each run, code is in place in Step 6 of `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

training size	hidden layer size	# of parameters	run time	scipy success flag	# iteration hit
100 images	10	16474	36s	False	4001
100 images	50	79234	1m21s	False	4001
100 images	100	157684	1m55s	True	3222
100 images	250	393034	4m51s	True	3418
1000 images	10	16474	6m03s	False	4001
1000 images	50	79234	7m05s	False	4001
1000 images	100	157684	9m17s	False	4001
1000 images	250	393034	14m34s	False	4001

Table 1: Problem 5 - summary of each run

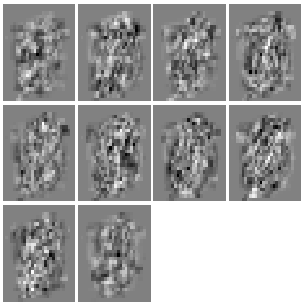
images, and also call `utils.autoencoder_feedforward()` with both the first 100 training patches and the 100 test patches to see what your autoencoder reproduces. The code currently defaults to save this using the same root pathname; you likely want to change the root pathname to something different 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?

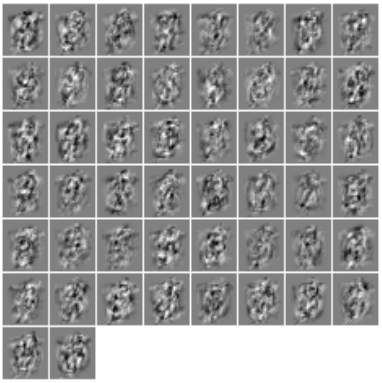
Solution: A summary of each run is shown in the Table 1. The plots for this problem is shown in Figure 3 - Figure 8, the captions suggest what each Figure is for. As we increase the hidden layer size from 10 to 50, and then to 100, the neural network can learn more about the structure (compressed) of the input, thus be able to closely resemble the corresponding input patches (both for training and testing input) for hidden layer size of 100. But from hidden layer size 100 to 250, the output doesn't look much different.

If hidden layer size is too small, for example, 10, we might lose some information about the input such that the neural network cannot closely resemble the input patches.

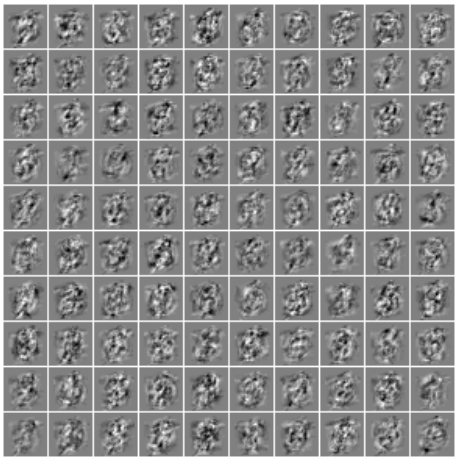
We can also see that as we increase the size of training patches from 100 to 1000, the neural network can better learn the input patterns with more input thus can resemble the input better.



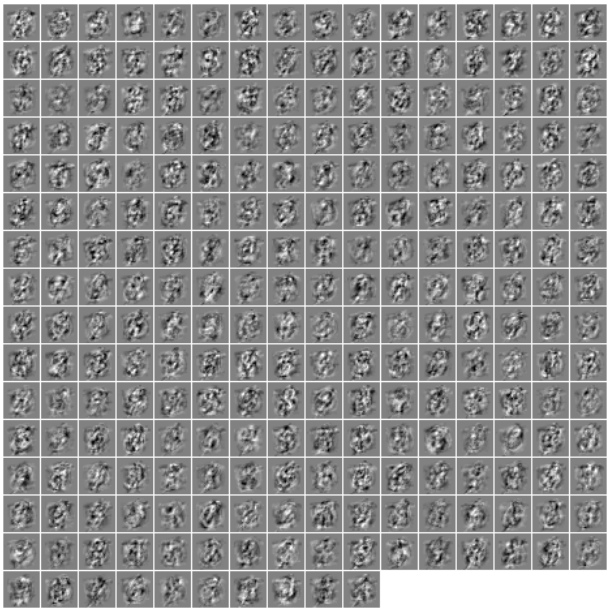
(a) hidden layer size: 10



(b) hidden layer size: 50

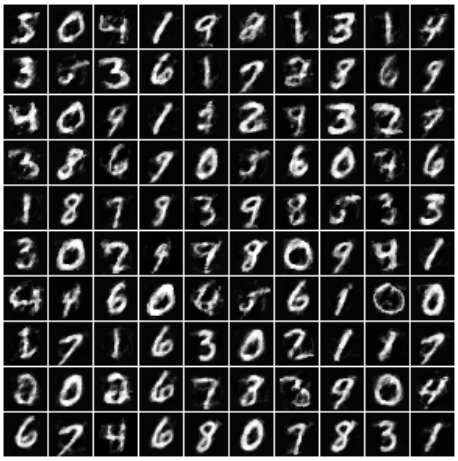


(c) hidden layer size: 100

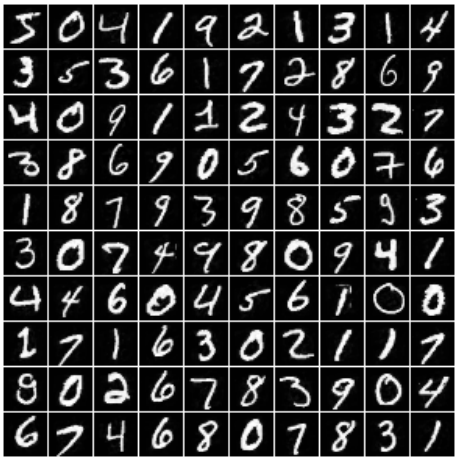


(d) hidden layer size: 250

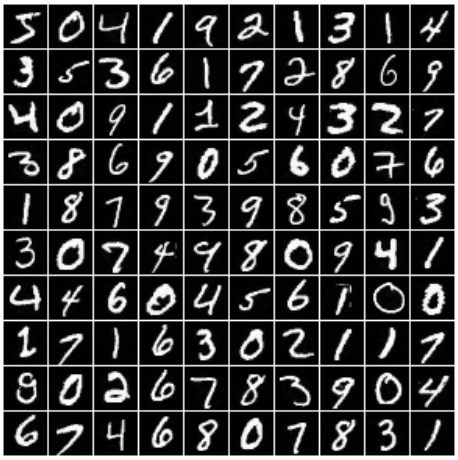
Figure 3: Problem 5 - layer 1 weight images with different hidden layer size for 100 training patches



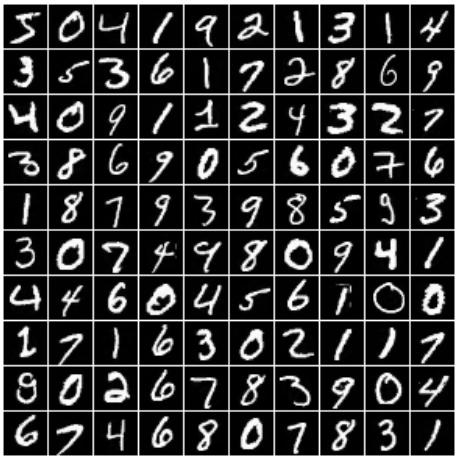
(a) hidden layer size: 10



(b) hidden layer size: 50



(c) hidden layer size: 100

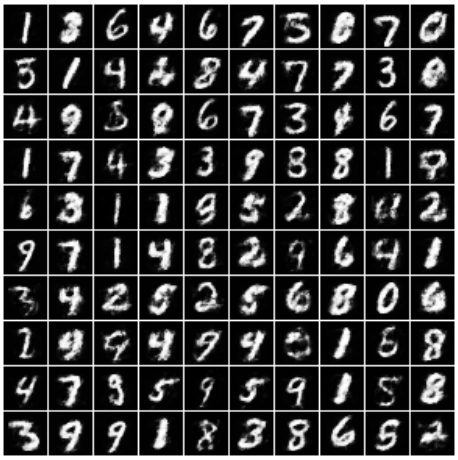


(d) hidden layer size: 250

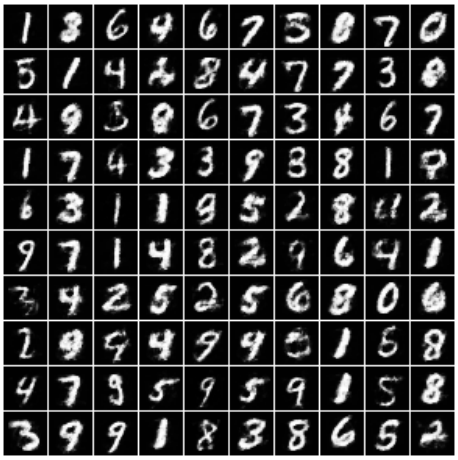
Figure 4: Problem 5 - layer 1 train decode images with different hidden layer size for 100 training patches

(a) hidden layer size: 10

(b) hidden layer size: 50

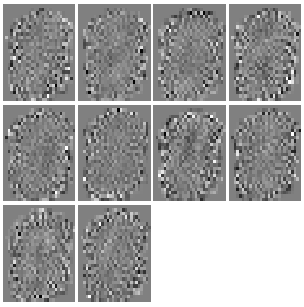


(c) hidden layer size: 100

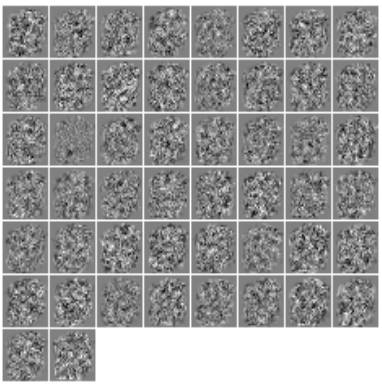


(d) hidden layer size: 250

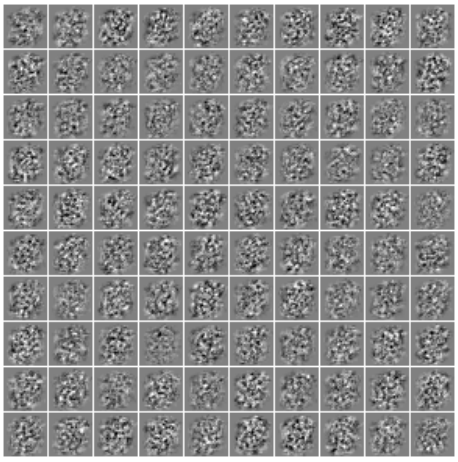
Figure 5: Problem 5 - layer 1 test decode images with different hidden layer size for 100 training patches



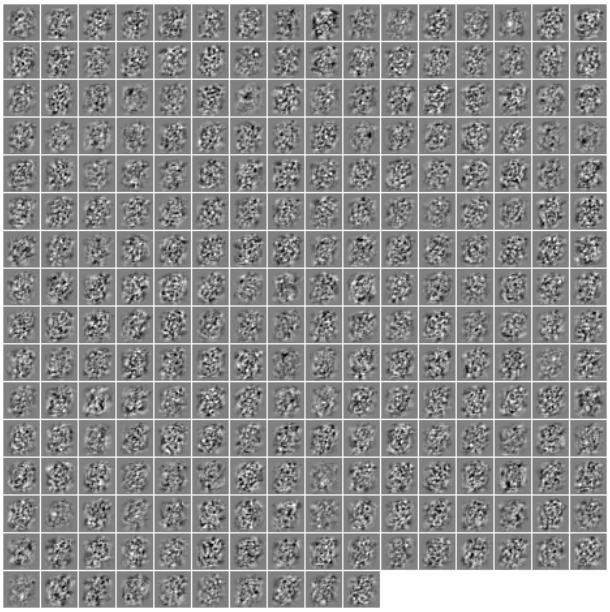
(a) hidden layer size: 10



(b) hidden layer size: 50

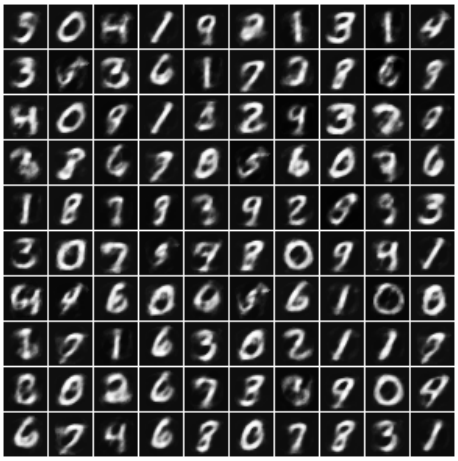


(c) hidden layer size: 100

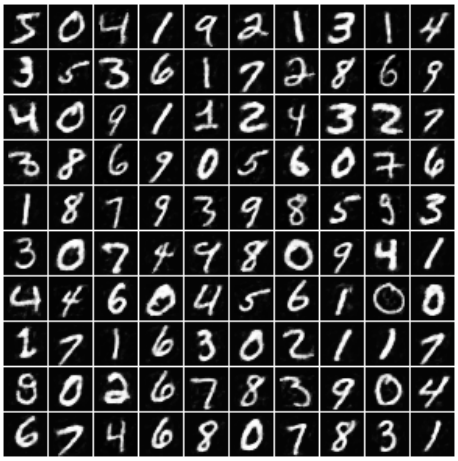


(d) hidden layer size: 250

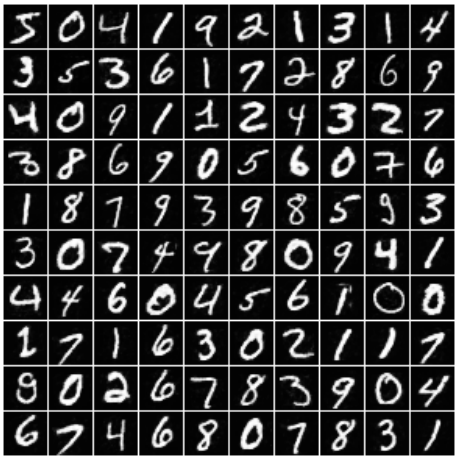
Figure 6: Problem 5 - layer 1 weight images with different hidden layer size for 1000 training patches



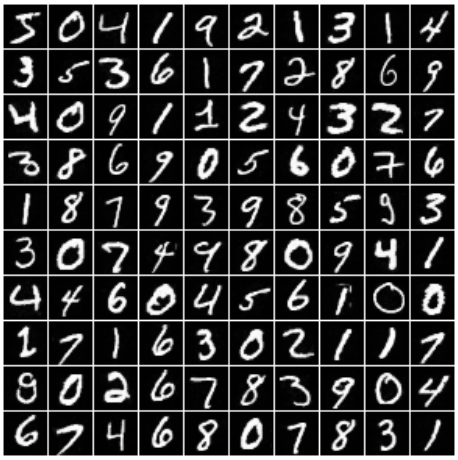
(a) hidden layer size: 10



(b) hidden layer size: 50

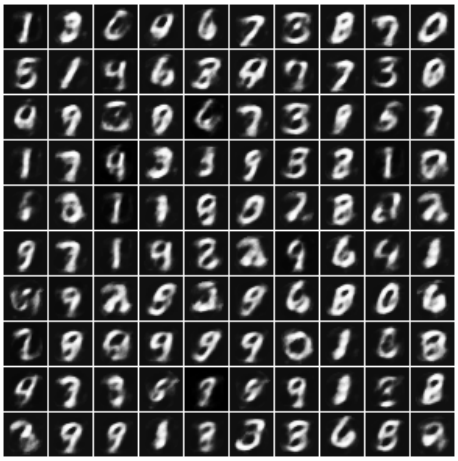


(c) hidden layer size: 100



(d) hidden layer size: 250

Figure 7: Problem 5 - layer 1 train decode images with different hidden layer size for 1000 training patches



(a) hidden layer size: 10

(b) hidden layer size: 50

(c) hidden layer size: 100

(d) hidden layer size: 250

Figure 8: Problem 5 - layer 1 test decode images with different hidden layer size for 1000 training patches

6. [6 points] Exercise 6: 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.

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.

As in Exercise 5, you will train using the first 100 patches, keeping `lambda_` at 0.0001, and try two different hidden layer sizes: 10, 50, 100 and 250. 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 and 250, 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.

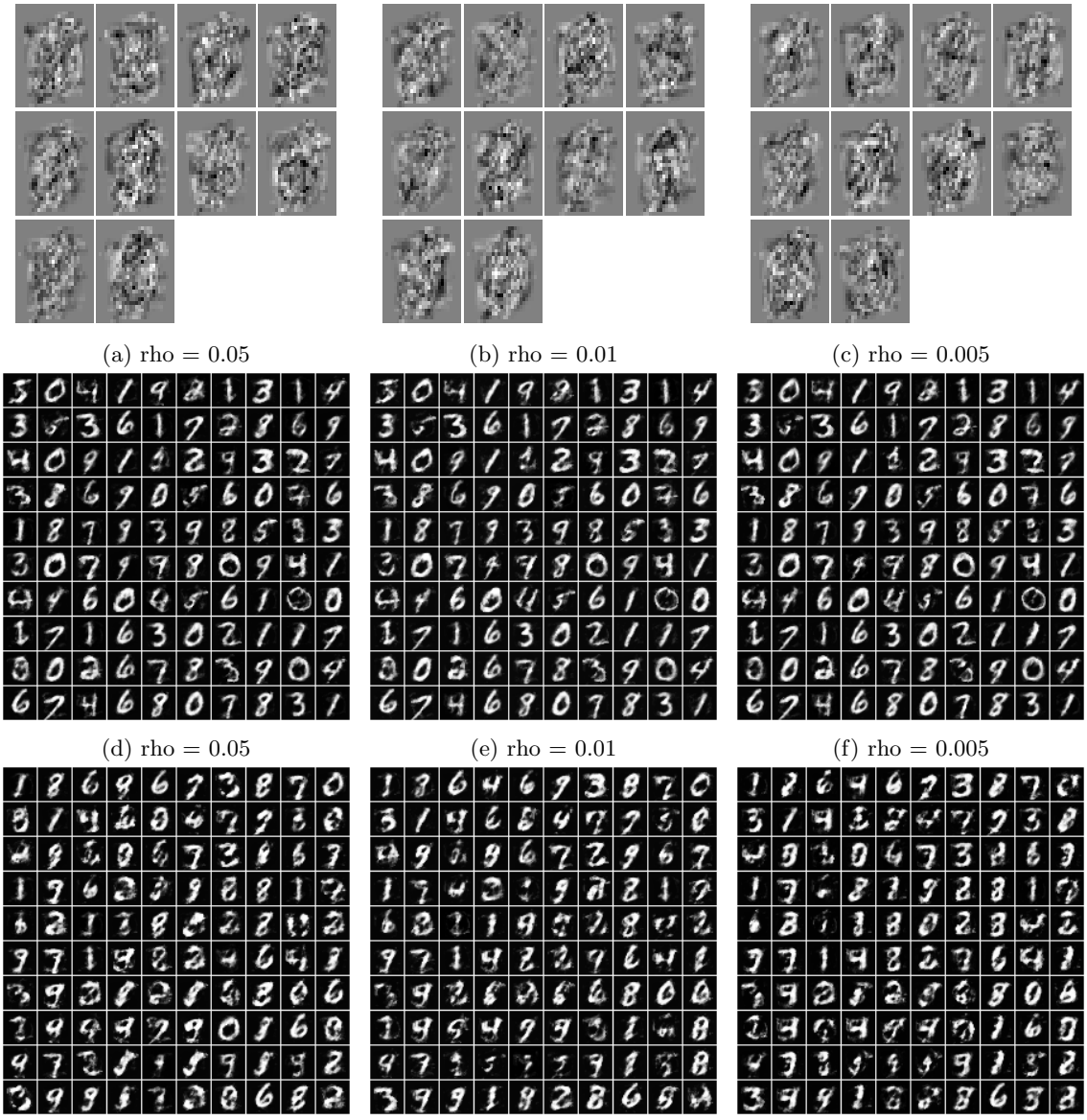
As in Exercise 5, report the patterns you see in both the weights as well as the output train and test decoding. Describe what you see in the images. Do you observe any general trends as you change the hidden layer size and corresponding sparsity parameters? Explain how this compares to the images in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

Solution: A summary of each run for 100 image patches is shown in the Table 2. The plots for this problem is shown in Figure 9 - Figure 12. As we increase the hidden layer size, the neural network can learn more about the compressed structure of input, thus can better resemble the input.

For smaller hidden layer size (10 and 50), decreasing the sparsity parameter doesn't make the predication better because the network will suppress more hidden layer units, thus we will lose more information about the input.

But for larger hidden layer size (100 and 250), decreasing the sparsity parameter will help the neural network learn the input structure better, because fewer units are in activation and carry more compressed input representation. This effect is obvious if see the weights image in Figure 12 where rho = 0.005, where we can see some dark spots in the weight image.

Comparing to images in exercise 5, the sparse authencoder is able to learn input structure better.

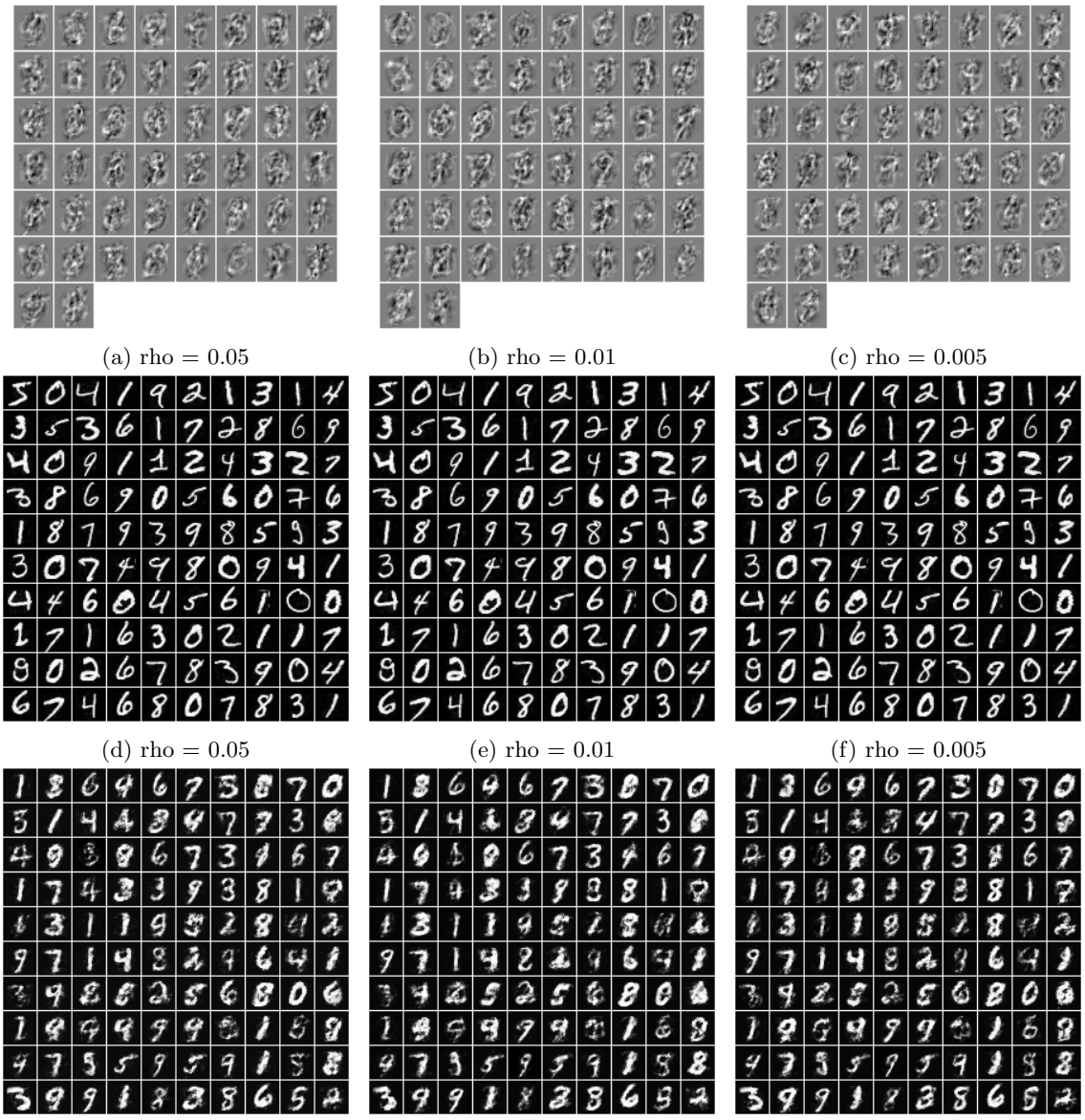


(a) rho = 0.05 (b) rho = 0.01 (c) rho = 0.005

(d) rho = 0.05 (e) rho = 0.01 (f) rho = 0.005

(g) rho = 0.05 (h) rho = 0.01 (i) rho = 0.005

Figure 9: Problem 6 - plots with hidden layer size = 10 and beta = 0.2 for 100 training patches. 1st row: weights image, 2nd row: train decode, 3rd row: test decode

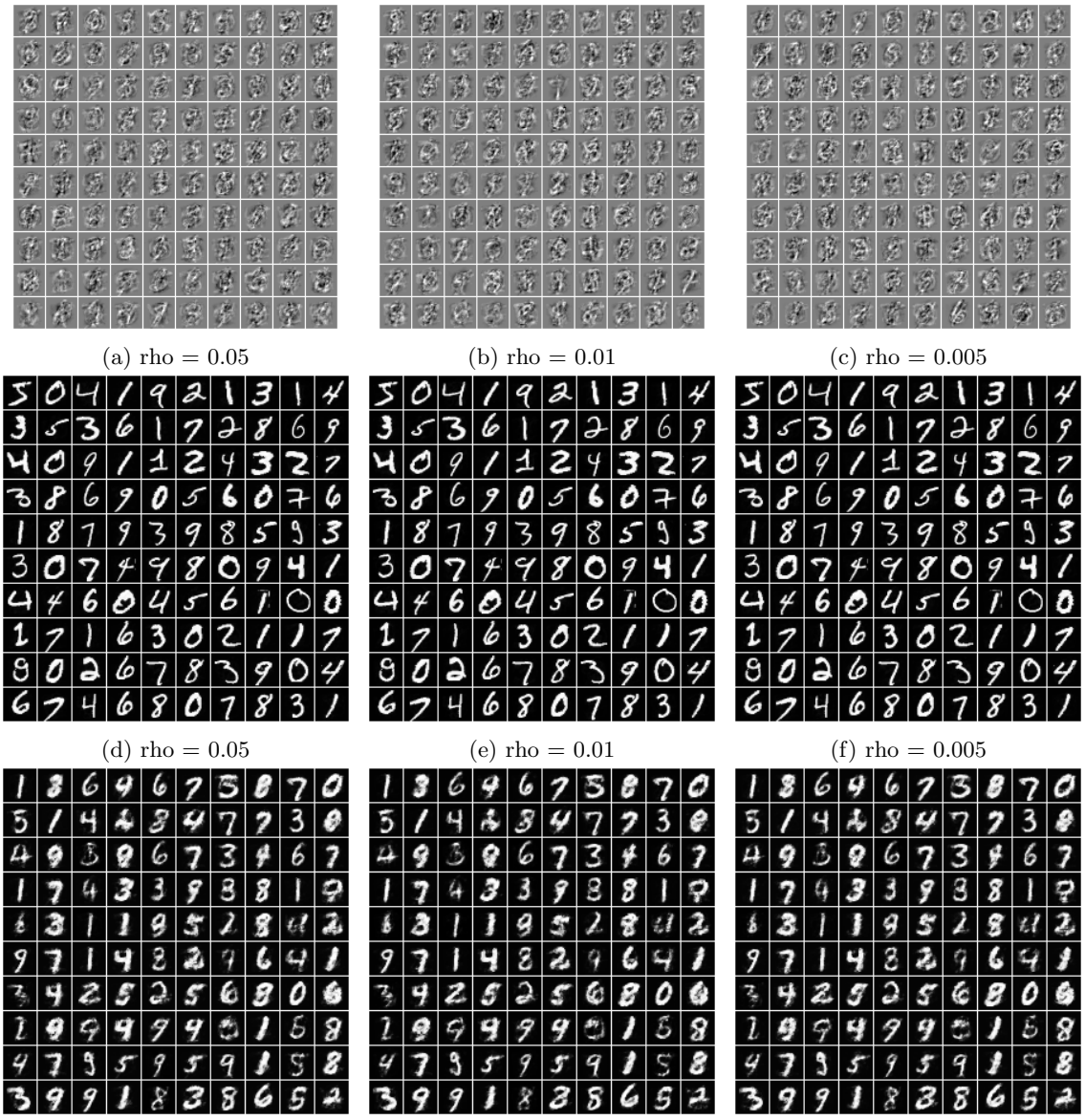


(a) rho = 0.05 (b) rho = 0.01 (c) rho = 0.005

(d) rho = 0.05 (e) rho = 0.01 (f) rho = 0.005

(g) rho = 0.05 (h) rho = 0.01 (i) rho = 0.005

Figure 10: Problem 6 - plots with hidden layer size = 50 and beta = 0.1 for 100 training patches. 1st row: weights image, 2nd row: train decode, 3rd row: test decode

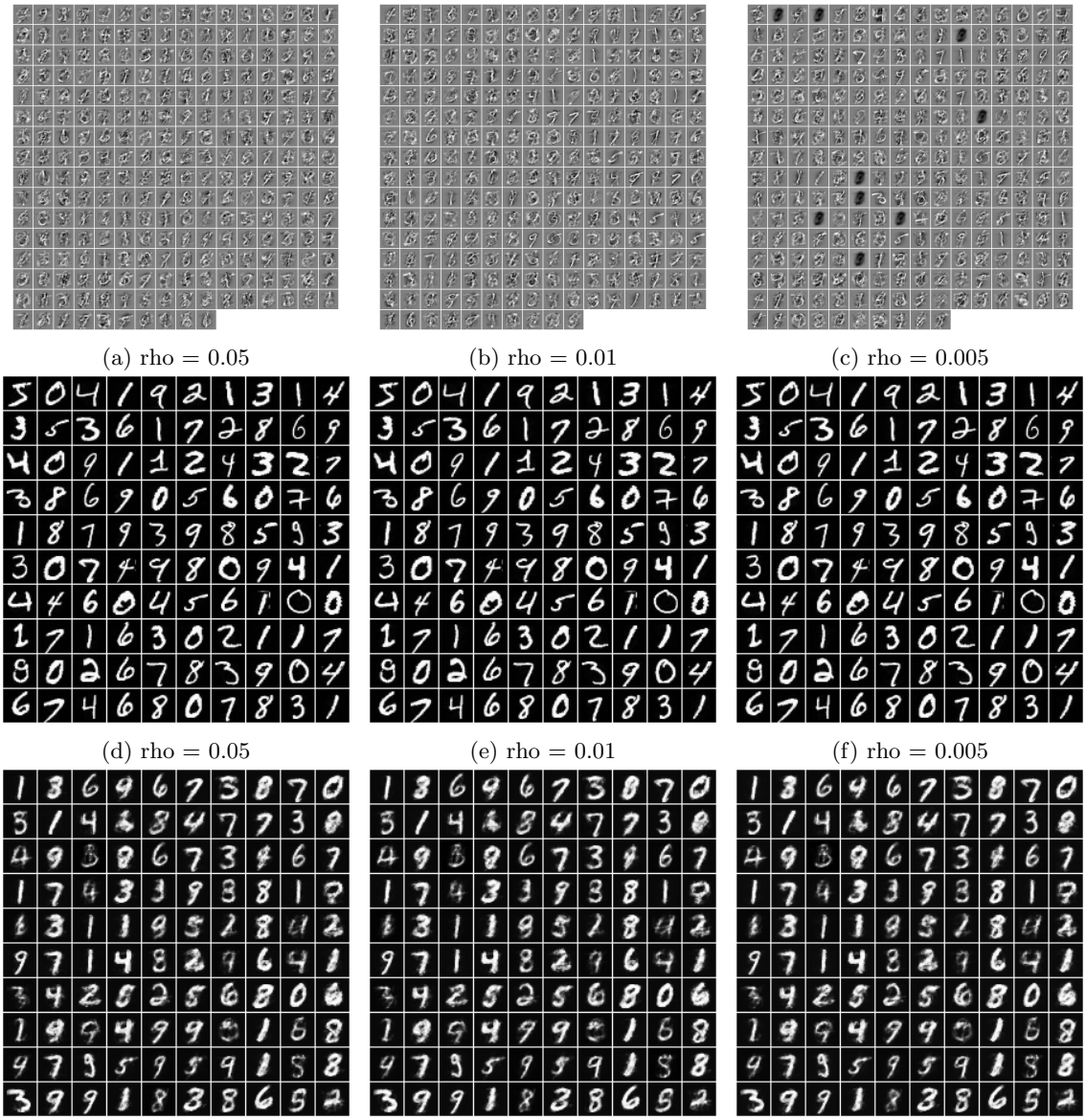


(a) rho = 0.05 (b) rho = 0.01 (c) rho = 0.005

(d) rho = 0.05 (e) rho = 0.01 (f) rho = 0.005

(g) rho = 0.05 (h) rho = 0.01 (i) rho = 0.005

Figure 11: Problem 6 - plots with hidden layer size = 100 and beta = 0.01 for 100 training patches. 1st row: weights image, 2nd row: train decode, 3rd row: test decode



(a) rho = 0.05 (b) rho = 0.01 (c) rho = 0.005

(d) rho = 0.05 (e) rho = 0.01 (f) rho = 0.005

(g) rho = 0.05 (h) rho = 0.01 (i) rho = 0.005

Figure 12: Problem 6 - plots with hidden layer size = 250 and beta = 0.01 for 100 training patches. 1st row: weights image, 2nd row: train decode, 3rd row: test decode

hidden layer size	beta	rho	run time	scipy success flag	# iteration hit
10	0.2	0.05	39s	False	4001
10	0.2	0.01	38s	False	4001
10	0.2	0.005	38s	False	4001
50	0.1	0.05	1m28s	False	4001
50	0.1	0.01	1m20s	False	4001
50	0.1	0.005	1m32s	False	4001
100	0.01	0.05	2m01s	True	3136
100	0.01	0.01	1m26s	True	2513
100	0.01	0.005	2m19s	True	3600
250	0.01	0.05	1m52s	True	964
250	0.01	0.01	2m03s	True	997
250	0.01	0.005	1m42s	True	918

Table 2: Problem 5 - summary of each run