HW Assignment 2 (Due by 9:00am on Feb 16)

1 Theory (100 points)

1. [Visualization of Hidden Units, 20 points]
   Prove that the input vector $x$ ($||x||_2 \leq 1$) that maximally activates the hidden layer unit $a_i^{(2)}$ of a sparse autoencoder has the form shown below:

   $$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{s_1}(W_{ij}^{(1)}J^2)}}$$

2. [Logistic Sigmoid, 10 points]
   Prove that the derivative of the logistic sigmoid $\sigma$ is $\sigma'(x) = \sigma(x) \times (1 - \sigma(x))$.

3. [Matrix Computations, 10 points]
   Let $U \in \mathbb{R}^{k \times m}$ and $X \in \mathbb{R}^{n \times m}$. Let $u_i$ and $x_i$ be the $i$-th columns of $U$ and $X$, respectively, for $1 \leq i \leq m$. Prove that $UX^T = \sum_{i=1}^{m} u_ix_i^T$.

4. [Universal Approximation, 40 points]
   Let $NN$ be a neural network with 2 input units, 1 hidden layer with $h$ units, and 1 output unit, using sigmoid as activation function. Furthermore, consider a training set that contains the following 4 examples i.e., the truth table of the logical XOR function:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

   (a) Is there a neural network $NN$ that perfectly classifies this training set? Prove your answer. If the answer is yes, what is the minimum $h$ for which there is a network with $h$ hidden neurons that fits the training data? Prove it.

   (b) Consider the same neural network $NN$, but without activation functions in the hidden layer. Answer the same questions as at item (a) above.

5. [PCA, 20 points]
   Let $y^{(k)} = UTx^{(k)}$ be the rotation of $x^{(k)}$ through PCA, where $U$ is the matrix of eigenvectors of the covariance matrix of the zero-mean dataset $X = [x^{(1)}, ..., x^{(m)}]$.

   Show that $\lambda_j = \frac{1}{m} \sum_{k=1}^{m}(y_{j}^{(k)})^2$. 
2 Implementation (150 points)

Download the skeleton code from http://ace.cs.ohio.edu/~razvan/courses/dl6900/hw/hw02.zip. Implement two versions of the autoencoder in Python, using (1) SciPy and (2) TensorFlow, and evaluate them on natural images (-t natural) and MNIST digits (-t digits). You will use the computeNumericalGradient.py code that you have written for the first assignment. Make sure that you organize your code in folders as shown in the table below.

```
dl6900/
  hw02/
    scipy/
      sampleNaturalImages.py
      sampleDigitImages.py
      sparseAutoencoder.py
      computeNumericalGradient.py
      output-natural.txt
      output-digits.txt
      weights_natural.jpg
      weights_digits.jpg
      sparseAutoencoderExercise.py
      checkNumericalGradient.py
      display_network.py
      IMAGES.mat
    tensorflow/
      sampleNaturalImages.py
      sampleDigitImages.py
      sparseAutoencoder.py
      output-natural.txt
      output-digits.txt
      weights_natural.jpg
      weights_digits.jpg
      sparseAutoencoderExercise.py
      display_network.py
      IMAGES.mat
  mnist/
```

Write code only in the files indicated in bold. The visualization of the hidden units should be saved in the jpg files listed above.

2.1 SciPy Implementation (100 points)

Coding effort: my implementation has 28 lines of code in sparseAutoencoder.py + 7 lines of code in sampleNaturalImages.py + 7 lines of code in sampleDigitImages.py.

1. Sampling images: The training data for the autoencoder will be created from random natural or digit images. For digit images, write code in the function `sampleDigitImages()` that returns 20,000 random 28x28 images from the entire MNIST dataset (including
training, validation, and testing). The images should be distinct. For natural images, write code in `sampleNaturalImages()` that returns 10,000 random 8x8 patches from the set of 10 512x512 natural images stored in IMAGES.mat. These images have been whitened, so the pixel values are not necessarily in [0, 1]. Consequently, the pixel values are further normalized by calling `normalizeData()`.

2. **Cost & Gradient:** You will need to write code for the function `sparseAutoencoderCost()` in sparseAutoencoder.py that computes the cost and the gradient. The cost and gradient should be computed according to the formulas shown on slides 5-9 in Lecture 2a. Use the sigmoid function for the activation function.

3. **Vectorization:** It is very important to vectorize your code so that it runs quickly.

4. **Gradient checking:** Once you implemented the cost and the gradient in `sparseAutoencoderCost()`, verify that your gradient code is correct by running the `sparseAutoencoderExercise.py` in `-- debug` mode. This will use the `computeNumericalGradient.py` code that you wrote for the previous assignment. The norm of the difference between the numerical gradient and your analytical gradient should be small, less than $10^{-9}$.

5. **Feature learning:** Training the autoencoder is done using L-BFGS for 400 epochs, through the SciPy function `scipy.optimize.fmin_l_bfgs_b()`. If completely vectorized, training the model on 20,000 random samples from the entire MNIST dataset should take about 20 minutes on california. Training on the 10,000 random patches from natural images should be much faster, due to the smaller number of samples and parameters.

6. **Visualization:** To visualize a learned feature, the code computes an input image that would maximally activate the corresponding hidden neuron. This is done using the formula in the first theory question above, which is implemented in the `displayNetwork()` function. The learned features should be similar to the ones shown in Figure 1: for natural images they should resemble Gabor edges, whereas for digits they should resemble pen strokes.
2.2 TensorFlow Implementation (50 points)

Coding effort: my implementation has 14 lines of code in sparseAutoencoder.py.

You will need to write code in sparseAutoencoder.py, inside the function `loss()`. Your code should build the sparse autoencoder model up to where it may be used for inference and add operations to compute the loss. This requires creating two sets of weights and biases variables, one for the hidden layer and one for the output layer.

The code for running gradient descent with minibatches is provided in sparseAutoencoderExercise.py. While you do not need to change this code, it is recommended that you read it and understand how it works. In particular:

1. **Threading and Queues**: In this TensorFlow implementation, multiple threads prepare training examples and push them in the queue. `tf.train.slice_input_producer` creates a FIFO queue for holding / providing the training input images, whereas `tf.train.batch` creates a FIFO queue for holding / providing the training mini-batches. Both functions return `dequeue` operations, such as ‘images’, that will become part of the computation graph. A `QueueRunner` for each queue is added to the current Graph’s QUEUE_RUNNER collection. Each QueueRunner holds the list of enqueue operations that need to be run in threads. Once the graph is constructed, the `tf.train.start_queue_runners` function asks each QueueRunner in the graph to start its threads running the enqueuing operations. A training thread will execute the training op that dequeues mini-batches from the queue. The [Threading and Queues](#) tutorial gives an overview on how to use threads and queues. This architecture has many benefits, as highlighted in the [Reading data how to](#) tutorial, which also gives an overview of functions that simplify the construction of input pipelines.

2. **Coordinator and QueueRunner**: The TensorFlow Session object is multithreaded, so multiple threads can easily use the same session and run ops in parallel. However, it is not always easy to implement a Python program that manages all the threads: threads must be able to stop together, exceptions must be caught and reported, and queues must be properly closed when stopping. TensorFlow provides two classes to help: `tf.Coordinator` and `tf.QueueRunner`. These two classes are designed to be used together. The Coordinator class helps multiple threads stop together and report exceptions to a program that waits for them to stop. The QueueRunner class is used to create a number of threads cooperating to enqueue tensors in the same queue. For an overview, see the [Coordinator and QueueRunner](#) tutorial. If all goes well, the code runs the training steps and the queues will be filled by the background threads. Since we set an epoch limit, at some point an attempt to dequeue examples will get an `tfOutOfRangeError`. This is the TensorFlow equivalent of ”end of file” (EOF) – this means the epoch limit has been reached and no more examples are available.

### 3 Submission

Turn in a hard copy of your homework report at the beginning of class on the due date. Electronically submit on Blackboard a hw02.zip file that contains the hw02 folder in which you change code only in the 5 required files. The screen output produced when running the softmaxExercise.py code should be redirected to (saved into) the `output.txt` files.
On a Linux system, creating the archive can be done using the command:

```
> zip -r hw02.zip hw02.
```

Please observe the following when handing in homework:

1. Structure, indent, and format your code well.
2. Use adequate comments, both block and in-line to document your code.
3. On the theory assignment, clear and complete explanations and proofs of your results are as important as getting the right answer.
4. Make sure your code runs correctly when used in the directory structure shown above.