DARTMOUTH ENGINEERING

Programming 1: (100 points)

This assigment is derived from one created by Andrew Ng while at Stanford. The Python implementation was created by Sai Ganesh and Shaowei Lin.

In this assignment you will implement a "sparse autoencoder." Autoencoders are neural networks designed to learn data representations - the representation here will be sparse, mostly zeros. We have included a reference document with notes on the topic.

There is a starter script notebook that contains helper functions. It also defines the following functions for you to fill in:

- sampleIMAGES()
- sparseAutoencoderCost()
- computeNumericalGradient()

In addition to completing the code, write a brief report discussing the obstacles you faced and the steps you took to solve them.

Step 1: Generate Training Data

The data in `images.npy` contains a numpy array with shape $(512, 512, 10)$, corresponding to 10 images, each 512 pixels square. To create a single training data point, randomly pick one of the 10 images and then randomly sample an 8×8 patch. Convert the patch into a 64-dimensional vector.

`sampleIMAGES()` should do this for 10,000 samples, returning an array of shape $(10000, 64)$. This should take only a few seconds. If it takes much longer, you may be making an unnecessary copy of the entire 512×512 image every iteration.

To check your implementation, run the code in Step 1 at the bottom of the script, which will visualize 100 random patches.

Step 2: Objective = Error + Penalty Terms

Implement code to compute the sparse autoencoder cost function $J_{\text{sparse}}(W, b)$ and the corresponding derivatives of J_{sparse} with respect to the different parameters. Use the sigmoid function for the activation functions:

$$f(z) = \frac{1}{1 + e^{-z}}$$

Complete this code in `sparseAutoencoderCost()`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \mathbb{R}^{s_1 \times s_2}$, $W^{(2)} \in \mathbb{R}^{s_2 \times s_3}$ and vectors $b^{(1)} \in \mathbb{R}^{s_2}$, $b^{(2)} \in \mathbb{R}^{s_3}$. For notational convenience, we will "unroll" these into a single long parameter vector θ with $s_1 s_2 + s_2 s_3 + s_2 + s_3$ elements. The code for converting between the two parameterizations is provided.

Debugging Tip: The objective function J_{sparse} is the sum of three terms: a squared error, an l_2 penalty (in neural networks, this is often called "weight decay"), and a sparsity penalty. It might help to first implement only the error terms (setting $\lambda = \beta = 0$) and then complete the gradient checking in the following section. Once you've verified the computation, go back and add the other terms and their derivatives.

Step 3: Gradient Checking

Following Section 2.3 of the provided lecture notes, implement code for gradient checking. Complete the code in `computeNumericalGradient()`. Please use `EPSILON`$= 10^{-4}$ as described in the lecture notes.

We have provided code in `checkNumericalGradient()` to help you test your code. This code defines a quadratic function $h : \mathbb{R}^2 \to \mathbb{R}$ given by $h(\mathbf{x}) = x_1^2 + 3x_1x_2$ and evaluates it at the point $\mathbf{x} = (4, 10)$. It allows you to check that your numerical gradient is very close to the true (analytically calculated) gradient.

After using `checkNumericalGradient()` to make sure that your implementation is correct, next use `computeNumericalGradient()` to test the gradient calculation of your `sparseAutoencoderCost()`. For details, see Step 3 at the bottom of the script. We strongly encourage you not to proceed until you've verified that your derivative computations are correct. While debugging, feel free to work with smaller cases, like 10 training data points and 1 or 2 hidden units.

Step 4: Training

Now that you have code that computes J_{sparse} and its derivatives, you're ready to minimize J_{sparse} with respect to its parameters. Rather than implement an optimization algorithm from scratch, we'll use a common algorithm called L-BFGS. This is provided for you in a function called

```
scipy.optimize.fmin_l_bfgs_b
```

which we import as `minimize` in the starter code[1]. We have provided the code to call `minimize` in Step 4 of the code. The function as written assumes the long θ parameterization as input.

Train a sparse autoencoder with 64 input units, 25 hidden units, and 64 output units. In the starter code we have provided a function for initializing the parameters. We set the biases $b_i^{(l)} = 0$ and the weights to random numbers drawn uniformly from the interval

$$\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}\right],$$

where n_{in} is the number of inputs feeding into a node and n_{out} is the number leaving a node. Note that different weight initialization schemes can cause widely different behavior. The values we have provided for the various parameters $(\lambda, \beta, \rho, \ldots)$ should work, but feel free to experiment.

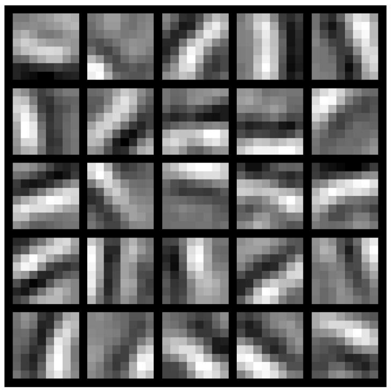
Debugging Tip: Once you've checked that your backpropagation algorithm is correct, make sure you're not doing numerical gradient-checking on every step. The reason we use backpropagation is that it's *much* faster than numerical estimation!

[1]You are not responsible for knowing how this algorithm works

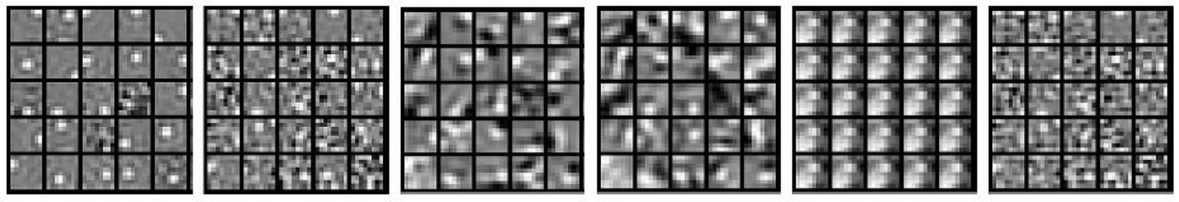
Step 5: Visualizing Results

After training the autoencoder, use `displayNetwork()` to visualize the learned weights and to save the visualization to a file `weights.jpg` that you will submit.

To receive full credit, you need to demonstrate that your autoencoder discovers that edges are a good representation of images. For example, this is a good result:



If your gradient calculation is incorrect or your parameters are poorly tuned, you may not see edges. The following images resulted from implementations with errors, and would not receive full credit:



Step 6: Classes and Objects

This step binds the code together into a single class, similar to how `scikit-learn` packages its algorithms. You do not need to write any code to execute this step.