EE 660 Machine Learning from Signals: Foundations and Methods Project Title: Analysis of Unsupervised Feature Learning Methods for Image Classification

Divya Ramesh (dramesh@usc.edu)
5924-4440-58

December 8, 2014

Abstract

This project investigates the utility of the unsupervised feature learning algorithm for the task of image classification. The state of the art methods for image classification use a pre-learning stage with deep networks to achieve superior results. In this project, one such pre-learning phase has been investigated and found to give significant improvement over the baseline performance obtained using raw pixel data as inputs to a classifier. The main aim of this project was to be able to find the relationship between the different hyper parameters of the feature learning model and the classification performance. This aim has been significantly achieved as is shown by the extensive study carried out in the project.

1 Problem Statement and Goals

The project aimed to investigate the use of the unsupervised learning algorithm sparse auto encoders for the task of automatic feature learning on periodically divided image patches. The dataset chosen was a large image dataset CIFAR-10[1] comprising of 60,000 32×32 RGB images split into training, testing and validation sets. This is a multi class classification problem, where the number of classes is 10. The problem proved to be even more challenging to me because of the following modifications done to suit the goals of the course project:

- The entire dataset was not chosen due to the size and time constraints of the project
- An effort was made to study the effect of unsupervised learning of features in the reduced feature space, i.e., the grayscale space.
- A large number of hyper parameters that needed to be tuned to achieve optimal performance. There were hyper parameters in the feature learning phase, and in the feature classification phase. Thus the problem seemed to have two black box testing phases.

The end goal of this project was to learn how to tune the hyperparameters of a feature learning algorithm to achieve best performance in the task of classification.

2 Literature Review

Several state-of-the-art image and object recognition researchers have obtained superior performance with their algorithms by including a pre-training unsupervised feature learning phase. The most important work in this field came from the authors in [2] where this feature learning was carried out on the MNIST handwritten digit recognition database. This method has also been used in activity recognition, speech recognition and other domains, where just a single layer of unsupervised learning improves the classification performance considerably [3, 5, 7]. The main work in this project is based on [4] where the feature learning is carried out using sparse-auto encoders.

3 Problem Formulation and Setup

The feature learning is performed using an unsupervised learning algorithm called the sparse auto encoder. An auto encoder is a neural network based unsupervised learning algorithm that applied sparsity by means of backpropagation. The backpropagation is carried out by setting the values of the target equal to the input values. Thus, the netwrok tries to compress the input space in the hidden layer, and reconstructs the input at the output using the compressed

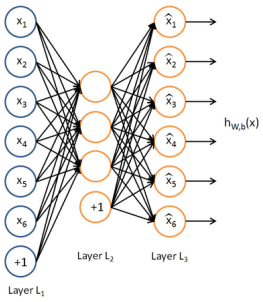


Figure 1: An autoencoder with one hidden layer

representation. The objective function thus tries to minimize the reconstruction error. If we have a minimum error in reconstruction, it means that the feature representation of the hidden layer is sufficient to capture the entire input space, and hence justifies for a valid feature representation. An image of the autoencoder is as shown in Fig. 1 [6]. A sparse autoencoder tries to enforce sparsity by enforcing a penalty parameter that keeps the activation of the hidden layers as close as possible to 0.

The problem consists of a huge dataset, which may not be linearly separable. The choice of features plays a very important role in determining the accuracy. Hand-engineered features may not always capture the best representations for the image classification task. Moreover, a non-linear mapping to a high dimensional feature space is almost always linearly separable. This fact is captured by the sparse auto encoder that uses a sigmoid activation function for each of its neural units.

In this problem, an attempt is made to study the effect of the hyperparameters trained on a simple classifier such as the linear Support Vector Machines(SVM). Although the parameters of the SVM have been tuned to achieve best performance of the dataset, the focus of this project has been on tuning the parameters of the unsupervised learning algorithm to achieve maximum performance.

4 Prior and Related Work

The entire experiment was carried out in MATLAB using the starter code provided by [6] for the sparse auto encoders. The starter code contained the L-BFGS algorithm for the minimization, but the functions for calculating sparse encoder cost, sampling of random patches and gradient checking had to be written from scratch. The feature convolution step which included extraction and pooling was also written from scratch. The entire work in this project was carried out solely for the purpose of this class.

5 Methodology

The steps in the setting up the problem are as outlined below:

1. The first 5000 samples from the training set was chosen to train the autoencoder to learn the features.
2. After obtaining the weights W an b, the entire set of 20000 images was used to extract the learned features, and these features were then fed to the linear SVM classifier.
3. The model was cross validated to find the best parameter of C corresponding to the maximum cross validation accuracy, and was re-trained using the found value.
4. Now the tedt set of 10,000 images was brought into picture to calculate the test accuracies. However, in order to compare the results with that of [4], the validation accuracies have been made use of.

6 Implementation

6.1 Feature Space

The dataset used to study this problem was the CIFAR-10 dataset[1] which consisted of 60000 32×32 color images in 10 classes, with 6000 images per class. There were 50000 training images and 10000 test images provided by the author. Each image was an RGB value of the type uint8.

6.2 Pre-Processing and Feature Extraction

The following steps were carried out to pre-process the data:

1. **Creation of reduced dataset**: The size of the dataset was reduced to just include 20,000 images for training. This was done by choosing the first two data batches from the CIFAR dataset. Further, the RGB color space was converted to the grayscale space in order to reduce the number of features.

2. **Unsupervised feature learning using sparse auto encoders**

   (a) **Generating training set:** The inputs to the auto encoders are a set of random patches extracted from random images of the training set. A set of 5000 images from the training set was chosen to help the auto encoder 'learn' the features. A random patch of size 8×8 is selected from an image picked at random, and is converted to a 64-dimensional vector to get a training sample that is $x \in \mathbb{R}^{64}$. 20,000 such patches were selected to create a matrix that is 64×20000 matrix. A plot of random 200 such patches is as shown in **??**.

   (b) **Minimization of sparse auto encoder objective:** The overall cost function to be reduced is as given below:

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \mathrm{KL}(\rho||\hat{\rho_j}), \tag{1}$$

   where,

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

   and $\mathrm{KL}(\rho||\hat{\rho_j})$ is the Kullback-Leiber(KL) divergence between a Bernoulli random variable with mean ρ, which is the sparsity parameter and another Bernoulli random variable with mean $\hat{\rho_j}$, which is the average activation of a hidden unit.

   The first term is an average sum-of-squares term and the second term is a regularization term. The wait decay parameter λ controls the relative importance of the two terms.

   The minimization of this objective function gives us the optimal set of weights and bias of the hidden layer, which are $\{W^{(1)}, b^{(1)}\}$.

   (c) **Training the sparse auto encoder** The auto encoder is then trained with the desired number of input units, using the L-BGFS algorithm. The L-BGFS algorithm was part of the starter code provided in [6].

3. **Feature extraction** The feature learning procedure outputs a mapping f that transforms the $\mathrm{N} = w \cdot w \cdot w$ input space to a K dimensional space, where K is the number of hidden units. The features are extracted by a convolution procedure where the mapping f is applied to equally spaced patches of size $w \times w$. w is called the *receptive field size*, and the spacing between patches is called the *stride*. These are some of the hyper parameters in the feature learning step.

6.3 Training Process

The training procedure can be divided into the following two phases:

1. **Feature Learning Phase:** In the feature learning phase, the parameters to be determined/chosen were:

   (a) The weights and bias parameters of the hidden layer

   (b) The number of hidden units, K

   (c) The receptive field size w

   (d) The stride length s

   (e) The number of random patches to be chosen

   (f) The number of samples in the subset of training samples to be chosen

   (g) The sparsity parameter of the auto encoder

   (h) The weight decay parameter of the objective function

   (i) The sparsity weights of the auto encoder

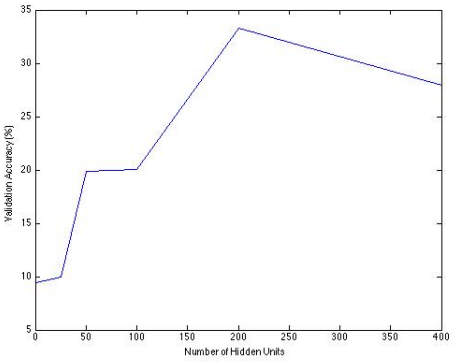


Figure 2: Validation Accuracy vs Number of Hidden Units

The weights and the bias parameters of the hidden layer are chosen based on the minimization of the cost function given in Eq. 1. The minimization of this cost function is in turn dependent on the rest of the other hyperparameters listed above. The number of hidden units K was chosen on validating the obtained weights visually. The number of hidden units tried were 25, 50, 100, 200 and 400. Further discussion on their selection is carried out in Section 6.4. The number of random patches to be chosen was determined heuristically, and was fixed at 20,000 after choosing the number of samples in the subset of training samples used exclusively for the purpose of feature learning. The feature learning was done from a random sample of 5000 training images, and the number of random patches was set at 20000 after a lot of experimentation. The receptive field size and the stride length, sparsity parameter, weight decay parameter and the sparsity weights were 8, 1, 0.01, 0.00001 and 3 respectively, as suggested in [4].

2. **Classification Phase:** In the classification phase, the parameters to be chosen were:

   (a) The type of SVM - linear, polynomial, kernel SVM

   (b) The choice of values for the cost C and gamma in the case of kernel SVM

   The type of SVM was converted to a model selection procedure, and the values of the cost C and gamma were determined by 4-fold cross-validation.

The size of the dataset was 20,000 training images in total, out of which 5000 images were chosen to learn the features from the unsupervised learning algorithm. The SVM classifier, however, was trained on the entire feature space resulting from 20,000 training images. The number of dimensions of the features was determined by the number of hidden units, thus were 25, 50, 100, 200 or 400. It was observed from the images of the weight representation that 25 hidden units captured very few edge information with respect to the others, while 400 captured too many edges, often giving rise to duplicate edge information. Thus, in order to avoid underfitting and overfitting, a hidden size of 200 was fixed upon to derive the final results.

6.4 Esperiments and Results

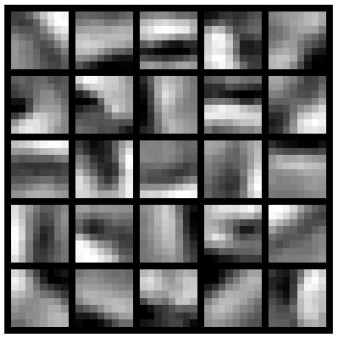
6.4.1 Model Selection

The model selection phase consisted of choosing from different classifiers. I chose the linear Support Vector Machine classifier in order to be able to compare my results with that of the paper being referred to for this project. The model selection for the learning phase was influenced by the prior knowledge of the problem which included the type of data, literature review of current and existing methods, and the size of the dataset chosen to study the specific problem.

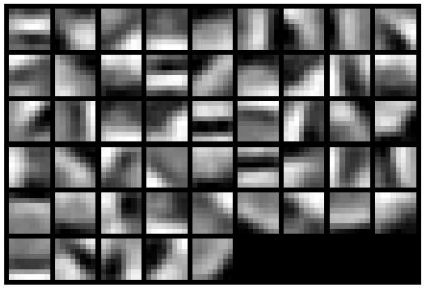
6.4.2 Effect of Number of Hidden Units

The number of hidden units affected the learning of the weights. The Fig. 3 shows the different feature representations learned by the hidden layers. It can be observed from Fig. 3a that 25 hidden units only capture the horizontal, vertical and diagonal edge information present in the dataset. As the numbr increases, the hidden units are able to learn features that are typically represented by wavelets such as Gabor and Haar. The intuition behind the selection of hidden units is this: The number of features should be sufficient to capture the entire image representation. Ideally, we would like this new feature mapping to contain lower dimensions than that of the original input space. Since the size of the images was 32×32, we would like to have a feature representation that is ¡ 900 dimensions. On

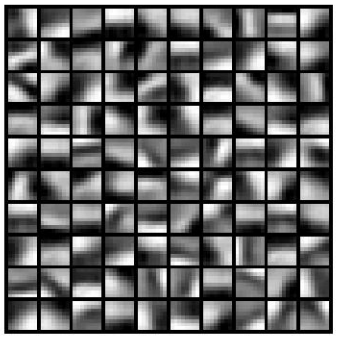
observing Fig. 3e, we can see that it has a number of duplicate feature representations. This could possibly mean an overfitting scenario. Similarly, the case of 25 units may possibly mean an underfitting scenario. Based on the visual representation of the weights, 200 hidden units were chosen to give a final validation accuracy. A plot of the validation accuracy with respect to the number of hidden units for 4-fold cross validation is as shown in Fig. 2. It is to be noted from the figure that 0 hidden units corresponds to the case of training the linear classifier on the reduced dataset (not the size referenced in the original paper) with just raw pixels as input. The validation accuracy is at 9.45% which is a quarter of the results reported in the paper. This can be due to the size of the reduced dataset, which contains only 5000 image samples for feature learning and 20000 images for training as opposed to 40000 and 50000 respectively. It can also be seen that 200 hidden units produce a significant improvement over the baseline performance by achieving 33% accuracy on the reduced dataset. However, the results are far worse than that reported in the original paper.



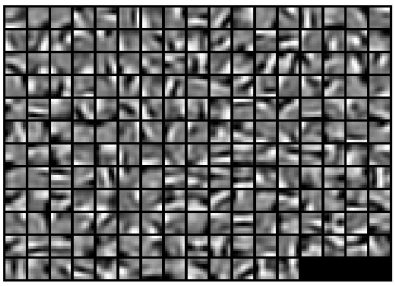
(a) 25 Hidden Units



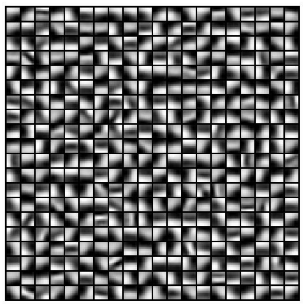
(b) 50 Hidden Units



(c) 100 Hidden Units



(d) 200 Hidden Units



(e) 400 Hidden Units

Figure 3: Learned Features Representation of the Hidden Units from CIFAR-10 Dataset

6.4.3 Number of Training Samples for Feature Learning

The number of training samples for feature learning was not much of an important factor, once the number of training samples was above certain threshold. The feature learning was tried with 5000, 10,000 and 20,000 samples and the

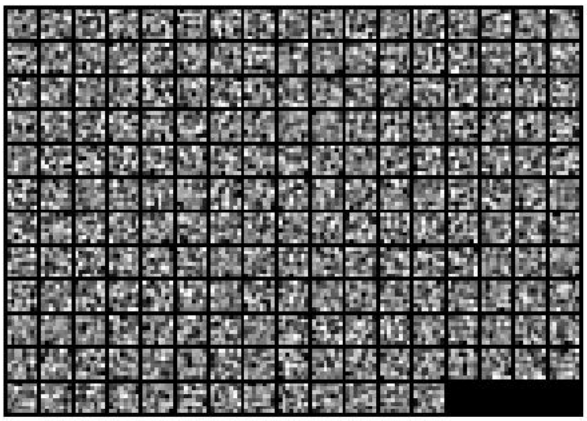


Figure 4: A very low sparsity parameter

accuracy was not shown to improve much. However, the number of random patches extracted from the training samples did matter. For numbers less than 10,000 patches, the validation accuracy did not show any significant improvement over that of raw pixels. A simple heuristic measure of 1:4 was chosen, and hence the number of patches extracted were 20,000 from a random feature learning training sample of 5000 images.

6.4.4 Effect of Sparsity Parameter and Sparsity Weights

The authors in the paper have suggested a sparsity parameter of 0.01, and mention that it should be fine for most applications similar to the CIFAR-10. Thus, the sparsity parameter was kept constant. However, on varying the sparsity weight β, the features learned from the hidden units showed a significant change. On increasing the weight, the features were more sharp, and on reducing the weights close to zero, the hidden units almost learned nothing useful. One such result is as shown in Fig. 4.

6.4.5 Validation and the Number of Folds

4-fold or 3-fold validation strategy seemed to give the best validation accuracy, considering the huge number of classes and the reduced dataset. Cross-validation was carried out to select the parameter C of the linear Support Vector Machines classifier. The best validation accuracy was obtained with 200 hidden units and a C value of 10e3. Higher values may have resulted in better accuracy values, but given the limited time and the huge size of the dataset, it was not a trivial task.

6.4.6 Final Results

The final test accuracy was found to be 31.13% by training a sparse encoder with 5000 random samples and 20,000 patches randomly samples from a set of 20,000 images. The SVM classifier was trained with 20,000 images, each of which contributed 4 feature vectors of dimension 200 concatenated along the feature dimensions, resulting in a total of 800 dimensions. Feature reduction was not performed as the features were not by themselves highly variant across dimensions. The test accuracy although better than the baseline performance of 9.45%, does not match the numbers cited in the reference paper, and this is due to the limited size of the dataset. Furthermore, the authors used L2 regularization of the Linear SVM to improve their results. Furthermore, the authors of the paper used the RGB data, while the entire experiments in this projet was done using gray-scale data. This is an important distinction to be noted for the interpretation of the results.

7 Summary and Conclusion

In this project, I have analyzed the utility of an unsupervised learning algorithm - sparse auto encoder for the purpose of feature learning of a large image database, the CIFAR-10. The hyperparameters have been experimented with and the results have been analyzed. A comparison with the baseline performance, defined by training the classification

algorithm on raw pixel data is made, and is found to be significantly better. The power of unsupervised learning algorithms for feature learning has been demonstrated by achieving satisfactory results from a simple linear classifier.

8 Future Work

I would like to try the following as an extension of this project:

- Include the entire CIFAR-10 dataset for training
- Vary the stride length and receptive field size of the auto encoder and observe its effects
- Compare the performance with some of the other state-of-the are unsupervised learning algorithms
- Extend the method to the CIFAR-100 dataset, using more complex classification models

References

[1] Alex Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, 2009.

[2] Boureau, Y-lan, and Yann L. Cun. "Sparse feature learning for deep belief networks." Advances in neural information processing systems. 2008.

[3] Ranzato, M., et al. "Unsupervised learning of invariant feature hierarchies with applications to object recognition." Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on. IEEE, 2007.

[4] Coates, Adam, Andrew Y. Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning." *International Conference on Artificial Intelligence and Statistics*. 2011.

[5] Niebles, Juan Carlos, Hongcheng Wang, and Li Fei-Fei. "Unsupervised learning of human action categories using spatial-temporal words." International journal of computer vision 79.3 (2008): 299-318.

[6] Andrew Y. Ng. CS294A Lecture Notes. `http://deeplearning.stanford.edu/tutorial/`.

[7] Lee, Honglak, et al. "Unsupervised feature learning for audio classification using convolutional deep belief networks." Advances in neural information processing systems. 2009.

Appendix

Codes

```
function patches = sampleCIMAGES()
% sampleIMAGES
% Returns 10000 patches for training

load ('/Users/dramesh/Documents/MATLAB/UnsupervisedFeatureLearning/Data/Data20k');    % load images from di


patchsize = 8;  % we'll use 8x8 patches
numpatches = 10000;
%epsilon = 1e-5;

% Initialize patches with zeros.  Your code will fill in this matrix--one
% column per patch, 10000 columns.
patches = zeros(patchsize*patchsize, numpatches);

%% ---------- YOUR CODE HERE --------------------------------------
%  Instructions: Fill in the variable called "patches" using data
%  from IMAGES.
%
%  IMAGES is a 3D array containing 10 images
%  For instance, IMAGES(:,:,6) is a 512x512 array containing the 6th image,
%  and you can type "imagesc(IMAGES(:,:,6)), colormap gray;" to visualize
%  it. (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(21:30,21:30,1) is an image
%  patch corresponding to the pixels in the block (21,21) to (30,30) of
%  Image 1

for i = 1: numpatches

    % Select a random image from the 10 training images
    randImage = randi(5000,1);

    % Select the top left co-ordinates of the random patch
    imgTopL = randi(size(CIMAGES20k,1)-patchsize,1);
    imgTopR = randi(size(CIMAGES20k,2)-patchsize,1);

    patch = CIMAGES20k(imgTopL:imgTopL+patchsize-1, imgTopR:imgTopR+patchsize-1, randImage) ;

    patches(:, i)=reshape(patch, patchsize*patchsize, 1);

end


%% ---------------------------------------------------------------
% 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);

end


%% ---------------------------------------------------------------
```

```
% Convolution and pooling

function features = convolve_and_pool()

    [W1, b1] = trainae;

   %load betatest;

   features = generatePatches(W1,b1);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%

function convolvedData = generatePatches(W,b)

    load ('/Users/dramesh/Documents/MATLAB/UnsupervisedFeatureLearning/Data/Data20k.mat');    % load images

 %   load labelData;

    convolvedData = zeros(4*size(W,1),size(CIMAGES20k,3));

    for i = 1:size(convolvedData,2)
        i
        convolvedData(:,i) = splitData(CIMAGES20k(:, :, i), W, b);
    end

end

function featureVectors = splitData(image, W, b)

   image1 = image(1:16, 1:16);
   image2 = image(1:16, 17:32);
   image3 = image(17:32, 1:16);
   image4 = image(17:32, 17:32);

   f1 = extract_and_pool(image1, W, b);
   f2 = extract_and_pool(image2, W, b);
   f3 = extract_and_pool(image3, W, b);
   f4 = extract_and_pool(image4, W, b);

   featureVectors = [f1;f2;f3;f4];
end

function f = extract_and_pool(impatch, W, b)

    wpatch11 = reshape(impatch(1:8, 1:8), 8*8, 1);
    f11 = W*normalizeData(wpatch11) + b;
    f11 = sigmoid(f11);
    wpatch12 = reshape(impatch(1:8, 9:16), 8*8, 1);
    f12 = W*normalizeData(wpatch12) + b;
    f12 = sigmoid(f12);
    wpatch13 = reshape(impatch(9:16, 1:8), 8*8, 1);
    f13 = W*normalizeData(wpatch13) + b;
    f13 = sigmoid(f13);
    wpatch14 = reshape(impatch(9:16, 9:16), 8*8, 1);
    f14 = W*normalizeData(wpatch14)+ b;
    f14 = sigmoid(f14);

    f = f11 + f12 + f13 + f14;
```

```
end

function sigm = sigmoid(x)

    sigm = 1 ./ (1 + exp(-x));
end

function whitepatch = imwhiten(patch)

      epsilon = 1e-5;
     avg = mean(patch, 1);     % Compute the mean pixel intensity value separately for each patch.
    patch = patch - repmat(avg, size(patch, 1), 1);
    sigma = patch * patch' / size(patch, 2);
    [U,S,~] = svd(sigma);

    whitepatch = diag(1./sqrt(diag(S) + epsilon)) * U' * patch;

end

function [cost,grad] = sparseAutoencoderCost(theta, visibleSize, hiddenSize, ...
                                             lambda, sparsityParam, beta, data)

% visibleSize: the number of input units (probably 64)
% hiddenSize: the number of hidden units (probably 25)
% lambda: weight decay parameter
% 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 64x10000 matrix containing the training data.  So, data(:,i) is the i-th training example.

% The input theta is a vector (because minFunc 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 = reshape(theta(1:hiddenSize*visibleSize), hiddenSize, visibleSize);
W2 = reshape(theta(hiddenSize*visibleSize+1:2*hiddenSize*visibleSize), visibleSize, hiddenSize);
b1 = theta(2*hiddenSize*visibleSize+1:2*hiddenSize*visibleSize+hiddenSize);
b2 = theta(2*hiddenSize*visibleSize+hiddenSize+1:end);

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


% Equation 6 and 7, forward propagation

z2 = bsxfun(@plus, W1*data, b1);
a2 = sigmoid(z2);

z3 = bsxfun(@plus, W2*a2, b2);
%hWbx = sigmoid(z3);
hWbx = z3;

JWbxy = 1/2*(hWbx - data).^2;

res = lambda/2 * (sum(sum(W1.^2))+ sum(sum(W2 .^ 2)));

JWb = sum(sum(JWbxy)/size(data,2)) + res;

pj = sum(a2,2)/size(data,2);

term = zeros(hiddenSize, 1);
term = term + sparsityParam .* log(sparsityParam ./ pj) + (1 - sparsityParam) .* log((1 - sparsityParam) ./

% Back propagation to compute derivatives
fdash3 = hWbx.*(1-hWbx);
%delta3 = -(data - hWbx).* fdash3;
delta3 = -(data - hWbx);

fdash2 = a2.*(1-a2);
delta2 = bsxfun(@plus, (W2'*delta3), (beta*(-sparsityParam./pj + (1-sparsityParam)./(1-pj)))).*fdash2;

% The desired partial derivatives
W2grad = delta3*a2'/size(data,2) + lambda*W2;
b2grad = sum(delta3, 2)/size(data,2) ;

W1grad = delta2*data'/size(data,2)+lambda*W1;
b1grad = sum(delta2,2)/size(data,2);


cost = JWb + beta * sum(term);
%-------------------------------------------------------------------
% After computing the cost and gradient, we will convert the gradients back
% to a vector format (suitable for minFunc).  Specifically, we will unroll
% your gradient matrices into a vector.

grad = [W1grad(:) ; W2grad(:) ; b1grad(:) ; b2grad(:)];

end

%-------------------------------------------------------------------
% 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)).

function sigm = sigmoid(x)

    sigm = 1 ./ (1 + exp(-x));
end
```