Programming Exercise: Sparse Autoencoder

Report:

1. In the sampleImages file, first set the size of the sample file and the number of sample files. Then create a 64*10000 empty matrix to receive the next sampled data. Second, the number of the picture and the range of the sample are obtained by randomly generated numbers. Finally, the 8*8 matrix is converted to a 64*1 vector by reshape, and the data of the 64*10000 matrix is updated.
2. The number of input and hidden layers is constructed first, and the matrix of W and B is defined. Then, an empty matrix corresponding to W and B is established, and this empty matrix also corresponds to the variable for calculating the gradient drop next. Second, the value of the output layer is obtained by computing Forward propagation. Then use Backpropagration to minimize the cost function, at which point the error of each neuron is calculated by establishing a loss function. Finally, the entire feedforward neural network is converted to a sparse autoencoder by adding sparse parameters. The final result is the parameters of the sparse encoder: W, b and loss function.

Loss function expression without sparse constraint:

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

KL divergence:

$$\mathrm{KL}(\rho||\hat{\rho}_j) = \rho\log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}$$

Loss function:

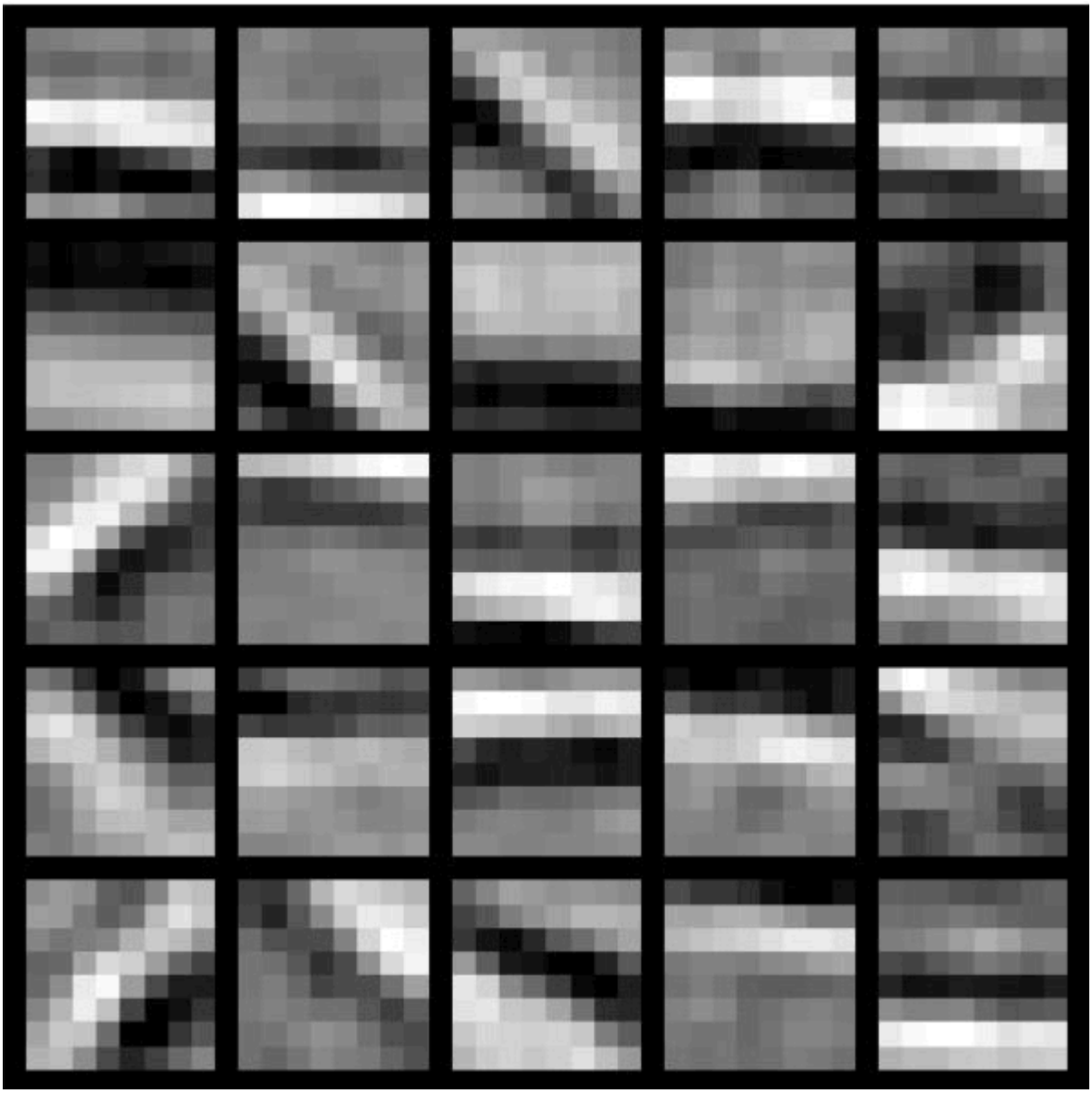
$$J_{\mathrm{sparse}}(W,b) = J(W,b) + \beta\sum_{j=1}^{s_2}\mathrm{KL}(\rho||\hat{\rho}_j),$$

Implicit expression of hidden nodes after adding sparse constraints

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

3. Calculate the gradient of the loss equation by using epsilon, which is required in advance.

4. Final image result below:



Source Code:

ComputeNumericalGradient.m:

```
function numgrad = computeNumericalGradient(J, theta)
% numgrad = computeNumericalGradient(J, theta)
% theta: a vector of parameters
% J: a function that outputs a real-number. Calling y = J(theta) will return the
% function value at theta.

% Initialize numgrad with zeros
numgrad = zeros(size(theta));

%% ---------- YOUR CODE HERE --------------------------------------
% Instructions:
% Implement numerical gradient checking, and return the result in numgrad.
% (See Section 2.3 of the lecture notes.)
% You should write code so that numgrad(i) is (the numerical approximation to) the
% partial derivative of J with respect to the i-th input argument, evaluated at theta.
% I.e., numgrad(i) should be the (approximately) the partial derivative of J with
% respect to theta(i).
%
% Hint: You will probably want to compute the elements of numgrad one at a time.
%check gradient for theta
%use for to go through each theta
for i=1:(size(theta))
  %apply formula below
  pos=theta
  pos(i)=0.0001+pos(i)
  neg=theta
  neg(i)=neg(i)-0.0001
  numgrad(i)=(J(pos)-J(neg))/((0.0001)*2)
end
%% ---------------------------------------------------------------
end
```

sampleIMAGES.m

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

load IMAGES;    % load images from disk

patchsize = 8;  % we'll use 8x8 patches
numpatches = 10000;
```

```
% 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
%check "numpatches" patches
for i=1:numpatches
   random_col=randi((512-patchsize));
   random_row=randi((512-patchsize));
   r1=random_row-1+patchsize
   r2=random_col-1+patchsize
   col=reshape(IMAGES(random_row:r1,random_col:r2,size(IMAGES,3)),[64,1]);
   patches(:,i)=col;
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


%% ---------------------------------------------------------------
function patches = normalizeData(patches)

% Squash data to [0.1, 0.9] since we use sigmoid as the activation
% function in the output layer

% Remove DC (mean of images).
patches = bsxfun(@minus, patches, mean(patches));

% Truncate to +/-3 standard deviations and scale to -1 to 1
pstd = 3 * std(patches(:));
patches = max(min(patches, pstd), -pstd) / pstd;
```

```
% Rescale from [-1,1] to [0.1,0.9]
patches = (patches + 1) * 0.4 + 0.1;

end
```

sparseAutoencoderCost.m

```
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
sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data, 2))),2)/size(data,2), 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.
%
%Forward propagation
FP=sigmoid((W2*(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))))+repmat(reshape
(b2,visibleSize,1),1,size(data,2))));
%cost result
cost=lambda/2*(sum(sum(W2.^2))+sum(sum((W1.^2))))+sum(sum((data-
FP).^2))/(2*size(data,2))+beta*sum(sparsityParam*log(sparsityParam./(sum(sigmoid(W1*data+repmat(
reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)))+(1-sparsityParam)*log((1-sparsityParam)./(1-
(sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)))));
%Back propagration
W2grad=lambda*W2+(-(data-FP).*FP.*(1 -
FP))*(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))))'/size(data,2);
b2grad=sum((-(data-FP).*FP.*(1-FP)),2)/size(data,2);
W1grad=lambda*W1+((W2'*(-(data-FP).*FP.*(1-FP))+(repmat(reshape(beta*(-
sparsityParam./(sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)
)+(1-sparsityParam)./(1-
(sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)))),hiddenSize,1
),1,size(data,2)))).*(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2)))).*(1-
(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))))))*data'/size(data,2);
b1grad=sum((W2'*(-(data-FP).*FP.*(1-FP))+(repmat(reshape(beta*(-
sparsityParam./(sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)
)+(1-sparsityParam)./(1-
(sum(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))),2)/size(data,2)))),hiddenSize,1
),1,size(data,2)))).*(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2)))).*(1-
(sigmoid(W1*data+repmat(reshape(b1,hiddenSize,1),1,size(data,2))))),2)/size(data,2);
%-------------------------------------------------------------------
% 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
```