# EEE443 Neural Networks

# Homework 3 Report

# Name: Oğuz Altan

# ID: 21600966

**Question 1)**

In this question, we implement an autoencoder neural network with a single hidden layer for unsupervised feature extraction from natural images, by minimizing cost function for autoencoder.

Part A)

We are given a dataset of images consisting of 16\*16 pixels 10240 images in RGB format. First, we preprocess the data by converting them to grayscale using a luminosity model and do further modifications on the data such as normalization and clipping the data range. 200 random sample patches in RGB format and normalized versions of these sample patches are given below:

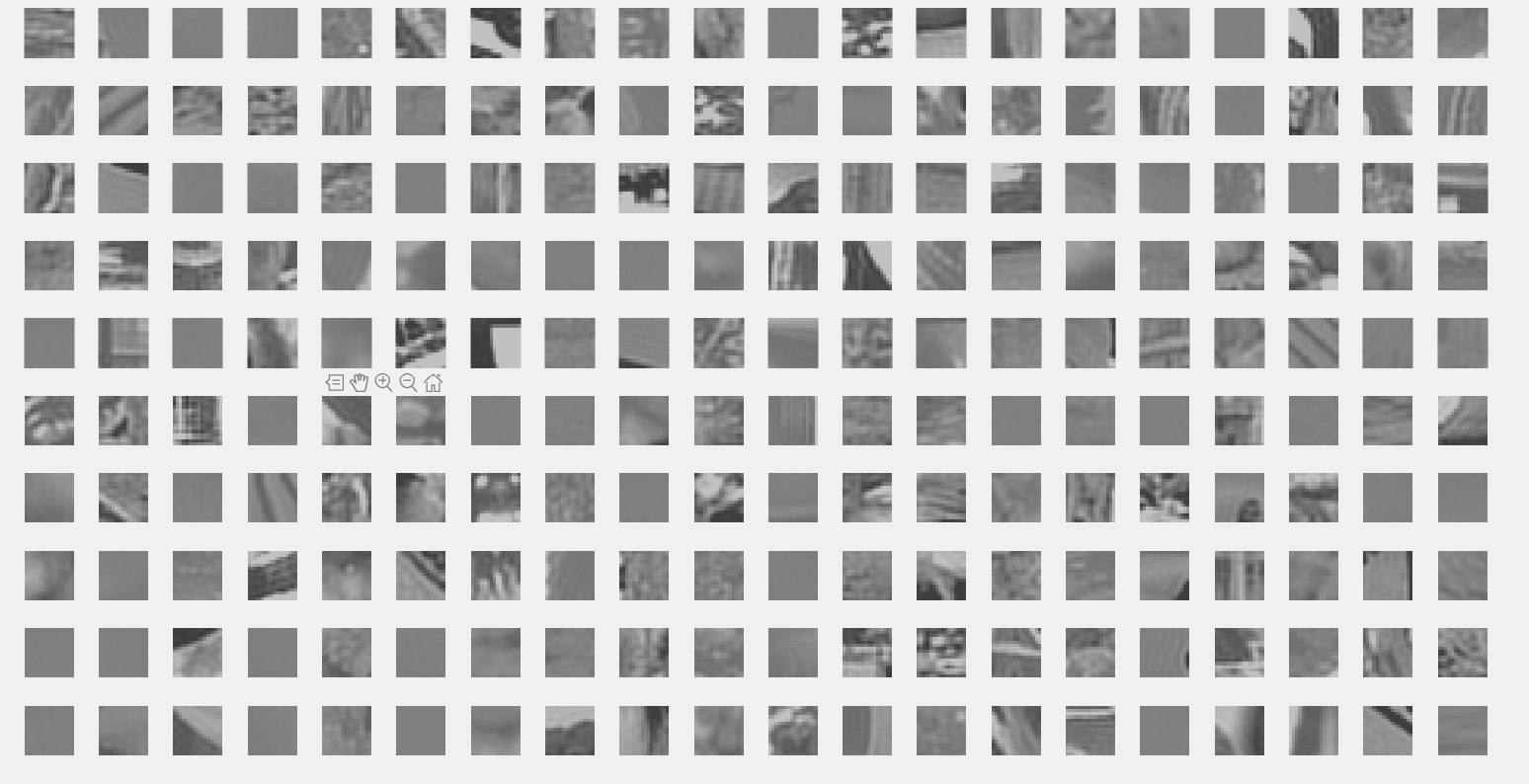


Figure 1: 200 Random Sample Patches in RBG

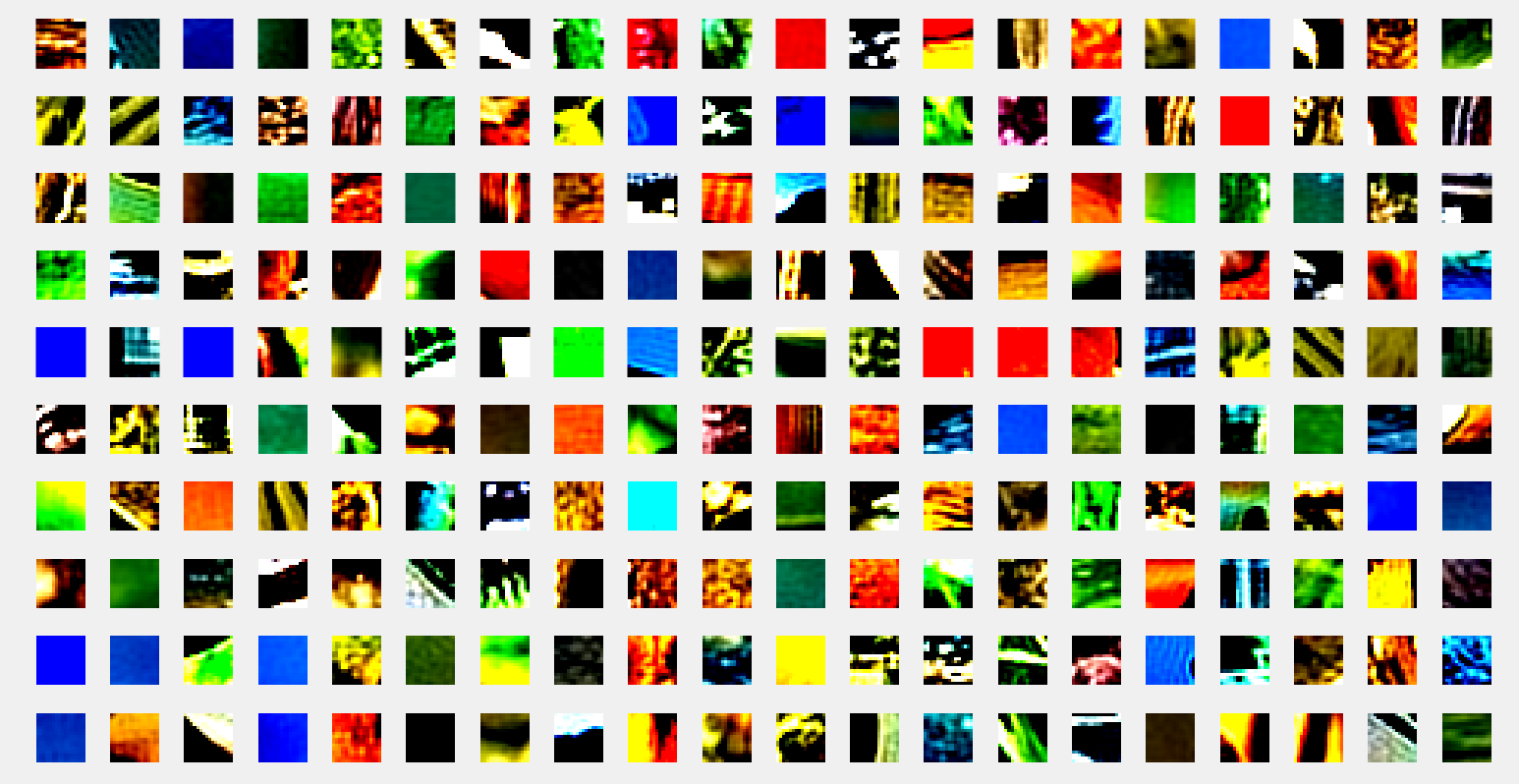


Figure 2: Normalized Versions of 200 Random Sample Patches

Inspecting two figures, it can be seen that normalized versions of the original RBG images are grayscaled versions and the we can detect the different textures and features such as edges and other basic features of original data. Another point is that as normalization brings color loss, monocolor images are almost completely gray, it is difficult to understand the texture on these kind of images.

Part B)

We want to minimize the cost function below:

This cost function consists of three separate terms. First term is Mean Squared Error, which is average squared-error between the desired response and the network output across training samples. Second term is regularization term and the third one is Kullback-Leibler divergence term that shows how much the average activation of hidden neurons is different from the sparsity .

KL (Kullback-Leibler) divergence term between a Bernouilli variable with mean and another with mean can be expressed as below:

where is the average of hidden layer neuron activations, calculated through the dataset.

Part C)

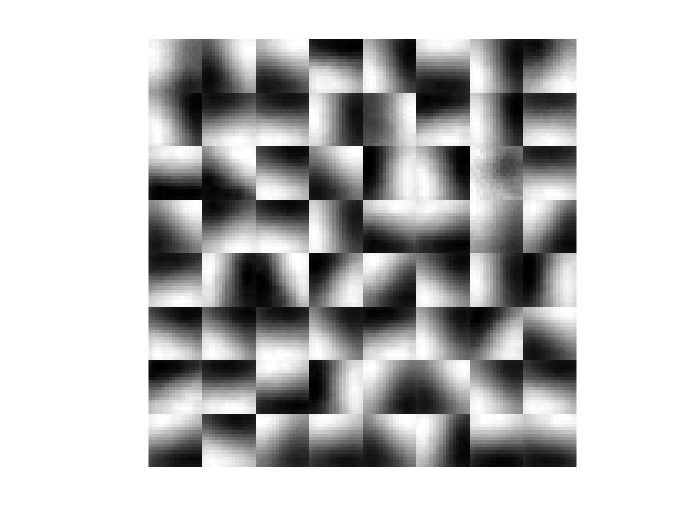


Figure 3: Hidden Layer Weights as a Separate Image for Each Hidden Layer Neuron

The parameters that I found which work well are and . Our hidden layer consists of 64 neurons and first layer of connection weights as a separate image for each neuron in the hidden layer are shown above. These features show the edges and orientations in the corresponding normalized image, therefore the images in the figure above, Figure 3, are decent in representing the original image data.

Part D)

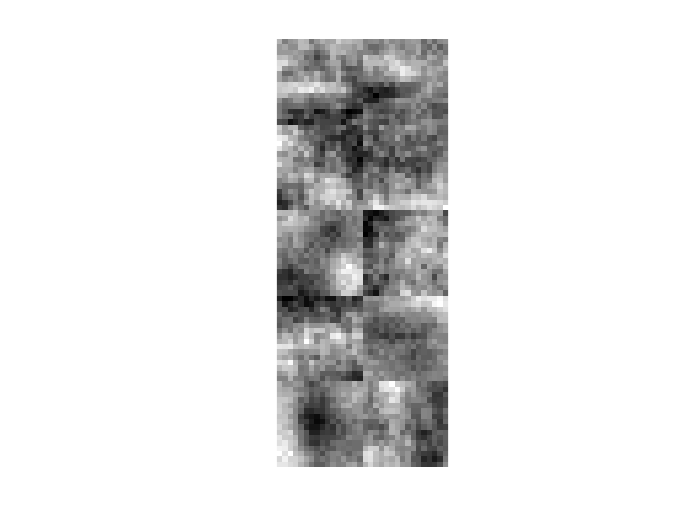
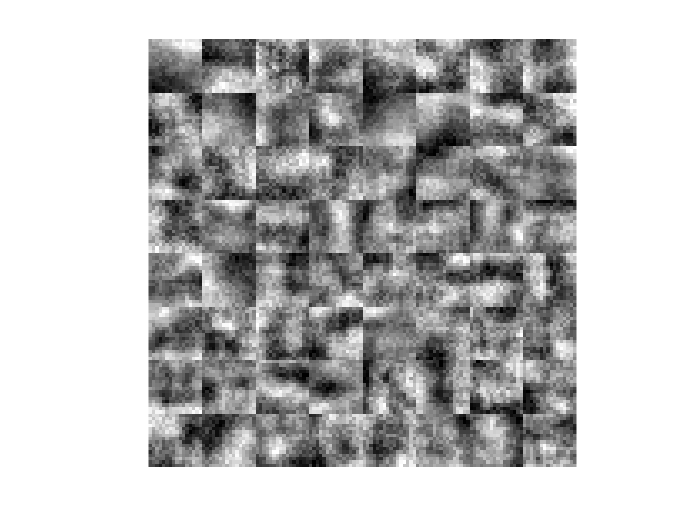


Figure 4: and L\_Hidden = 10

Figure 5: and L\_Hidden = 64

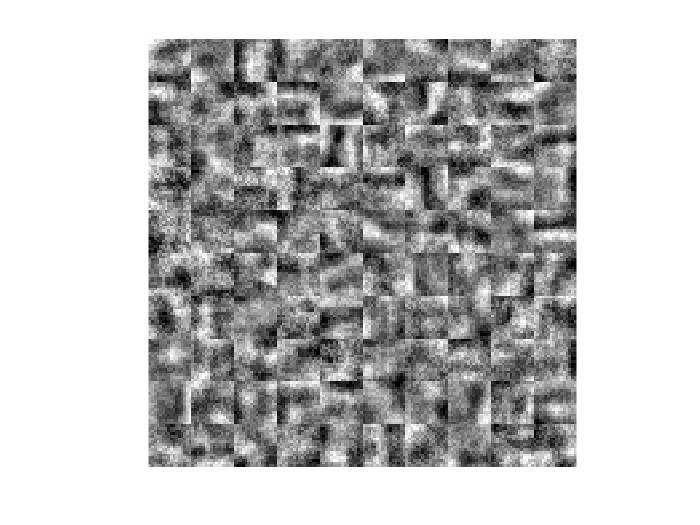


Figure 6: and L\_Hidden = 100

Analyzing the configuration with three different number of hidden layer neurons, denoted as L\_Hidden, which are 10, 64 and 100 respectively to the figures, with a constant parameter we can see the effect of number of hidden layer neurons. As hidden layer neurons learn the features of the original input image data, as the number of hidden layer neurons increase, they well learn the features the original image data, for instance in our case, as the number of hidden layer neurons increase, the network can better detect the edges and orientations in the original images.

Considering the effect of , as this parameter is related to the regularization term in the cost function, we can see that happens when is 0. Regularization term smoothens out the transitions in the grayscale, so no distinct transitions between white and black, in an grayscale image, if we have an optimal regularization term. For the case of absence of , then we do not have any regularization term so we observe sudden, not smooth, transitionts between white and black. This situation can be seen on the figures above.

We increase the and train the network again for the same three different number of hidden layer neurons.

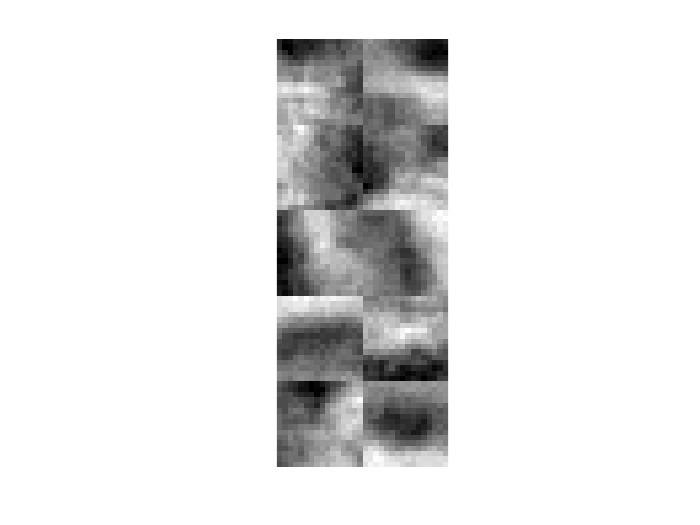
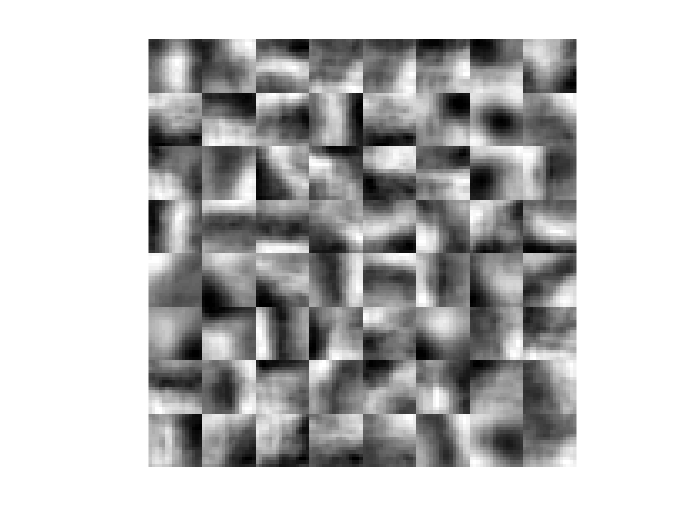


Figure 7: and L\_Hidden = 10

Figure 8: and L\_Hidden = 64

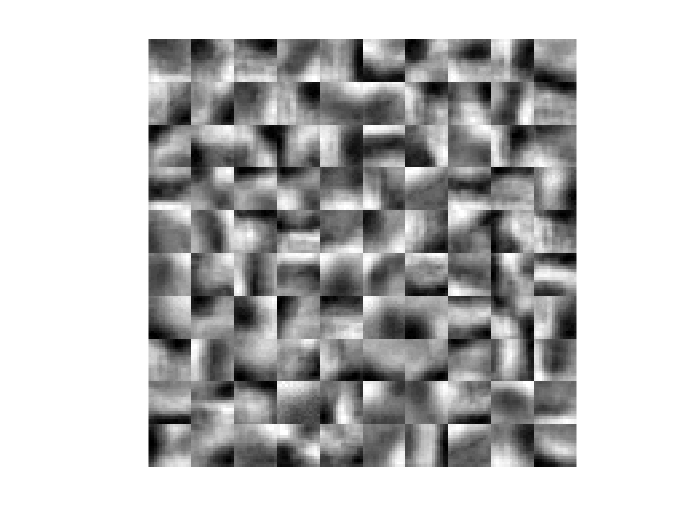


Figure 9: and L\_Hidden = 100

The point of regularization term and can be seen on the figures above, for the case .

The difference between the three figures for the and is that the edges and orientations on the hidden weight representations for the are much more apparent and transitions are smoother, comparing to the case. Again, the effect of number of hidden layer neurons is as it is explained above, for the case.

For the next case, we increase to and train the network again for the same three different number of hidden layer neurons.

Figure 12: and L\_Hidden = 100

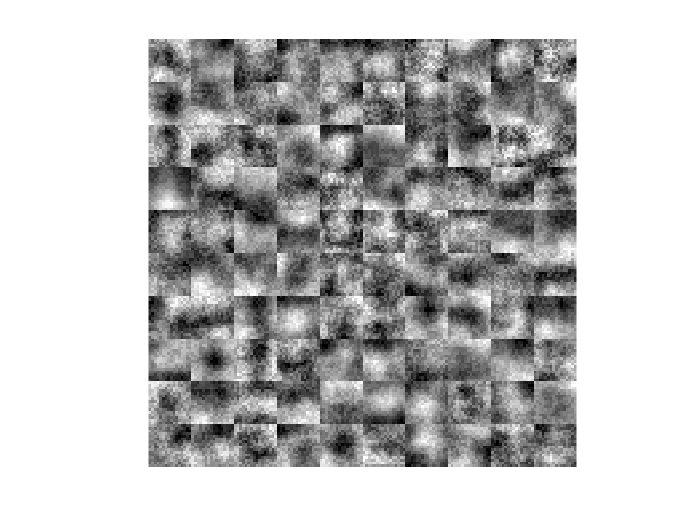
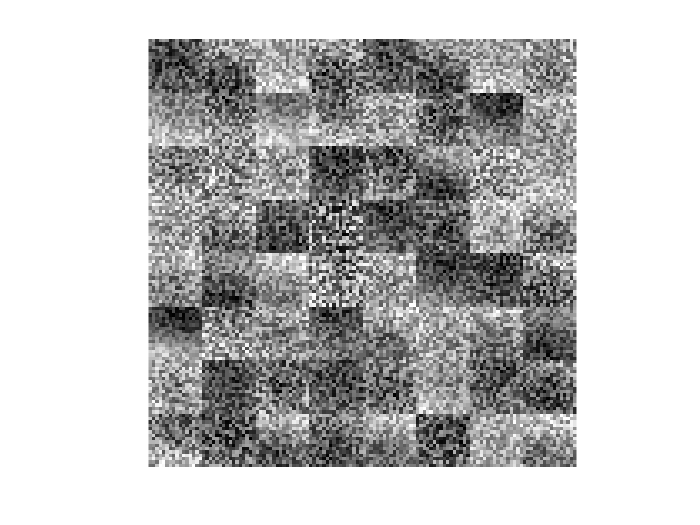
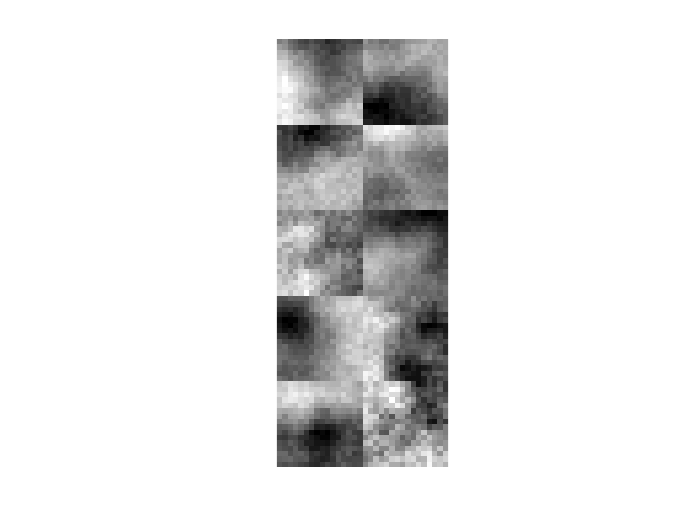


Figure 11: and L\_Hidden = 64

Figure 10: and L\_Hidden = 10



As it can be seen on the three figures above, the results are very noisy and transitions are sharp, not smooth. Bigger value means that the value of regularization term increases, which results in penaltying the network more harshly. This situation prevents network to learn the features efficiently.

To conclude the analysis on effect of and number of hidden layer neurons, as an overall point, as number of hidden layer neurons increase, the more features the network learns of the original image data but increasing the number of hidden layer neurons more than optimal results in overfitting the data, therefore the network becomes worse in predicting the original dataset. For an opposite situation, if the number of hidden layer neurons is small, then the network cannot learn well the features and performs worse. Increasing the results in smoother transitions, but if it exceeds the optimal value, the network suffers from noise, which can be seen on the Figure 10, 11 and 12.

**Question 2)**

**Important Note about Comments:**

The comments about the outputs and results in this question can be seen just below the every output section in the Jupyter notebook. The results are attached as a PDF to this report.

APPENDIX

function oguz\_altan\_21600966\_hw3(question)

clc

close all

switch question

case '1'

disp('1')

%% question 1 code goes here

clear;

load assign3\_data1;

%part a

gray\_data = 0.2126\*data(:,:,1,:) + 0.7152\*data(:,:,2,:) + 0.0722\*data(:,:,3,:);

flat\_gray\_data = reshape(gray\_data,[256, 10240]);

remove\_pix = flat\_gray\_data - mean(flat\_gray\_data);

std\_mean = std(remove\_pix(:)); %finding std across all pixel in the data

meanpix = max(min(remove\_pix, 3\*std\_mean), - 3\*std\_mean) / 3\*std\_mean;

normalized\_ims = (meanpix + 1) \* 0.4 + 0.1;

norm\_in\_gray = reshape(normalized\_ims,[16,16,10240]); %reshape to matrix 16\*16\*10240

rand\_patch = randperm(10240);

figure;

for i = 1:200

subplot(10,20,i);

imshow(data(:,:,:,rand\_patch(i)));

end

figure;

for i = 1:200

subplot(10,20,i);

imshow(norm\_in\_gray(:,:,rand\_patch(i)))

end

%part b

num\_images = 10240; %number of images in the dataset

params.L\_in = 256; %input is 256 pixel image vector

params.L\_hid = 64; %by architecture, number of neurons for output and input are same

params.lambda = 5e-4;

params.rho = 0.01;

params.beta = 1.5;

options = optimset('MaxIter',250);

W\_interval = sqrt(6/(params.L\_in + params.L\_hid));

W\_hid = - W\_interval + (2 \* W\_interval) .\* rand(params.L\_in, params.L\_hid);

W\_out = - W\_interval + (2 \* W\_interval) .\* rand(params.L\_hid, params.L\_in);

b\_hid = rand(1,params.L\_hid)\*2\*W\_interval-W\_interval;

b\_out = rand(1,params.L\_in)\*2\*W\_interval-W\_interval;

We = [W\_hid(:) ; W\_out(:) ; b\_hid(:) ; b\_out(:)];

costFunction = @(We) aeCost(We,normalized\_ims,params);

[we\_opt, cost, ep] = fmincg(costFunction,We,options);

W1 = reshape(we\_opt (1:params.L\_hid\*params.L\_in), params.L\_in, params.L\_hid);

show\_w(W1);

end

end

function [J, Jgrad] = aeCost(We,data,params)

[~,num\_images] = size(data);

%packing up weight and bias matrix

W\_hid = reshape(We(1 : params.L\_in\*params.L\_hid),params.L\_in,params.L\_hid);

W\_out = reshape(We(params.L\_in\*params.L\_hid+1 : params.L\_in\*params.L\_hid\*2),params.L\_hid,params.L\_in);

b\_hid = reshape(We(params.L\_in\*params.L\_hid\*2+1 : params.L\_in\*params.L\_hid\*2 + params.L\_hid),1,params.L\_hid);

b\_out = reshape(We(params.L\_in\*params.L\_hid\*2 + params.L\_hid + 1 : size(We)),1,params.L\_in);

%feedforwarding network

hid\_act = sigmoid(W\_hid'\*data - b\_hid');

out\_nn = sigmoid(W\_out'\*hid\_act - b\_out');

rho\_hat = mean(hid\_act,2);

%calculating errors separated into three terms in the cost function

mse = (1/(2\*num\_images)).\*sum(sum((data-out\_nn).^2,2));

regul = (params.lambda/2)\*(sum(W\_hid.^2,'all') + sum(W\_out.^2,'all'));

KL\_div = params.beta\*sum((params.rho\*log2(params.rho./rho\_hat) + (1-params.rho)\*log2((1-params.rho)./(1-rho\_hat))));

J = mse + regul + KL\_div; %calculating total cost

Jgrad\_W\_hidden = (-(1/num\_images)\*(((W\_out\*((data-out\_nn).\*((1-out\_nn).\*out\_nn))).\*((1-hid\_act).\*hid\_act))\*data')+ ...

params.lambda\*W\_hid' + params.beta\*(1/log(2)).\*(-params.rho./rho\_hat + (1-params.rho)./(1-rho\_hat)).\*(1/num\_images).\*(((1-hid\_act).\*hid\_act)\*data'))';

Jgrad\_W\_out = (-(1/num\_images)\*(data-out\_nn).\*((1-out\_nn).\*out\_nn)\*hid\_act' + params.lambda\*W\_out')';

Jgrad\_b\_hid = (-(1/num\_images)\*((W\_out\*((data-out\_nn).\*((1-out\_nn).\*out\_nn))).\*((1-hid\_act).\*hid\_act)\*(-1\*ones(1,num\_images))') + ...

params.beta\*(1/log(2)).\*(-params.rho./rho\_hat + (1-params.rho)./(1-rho\_hat)).\*(1/num\_images).\*(((1-hid\_act).\*hid\_act)\*(-1\*ones(1,num\_images))'))';

Jgrad\_b\_out = (-(1/num\_images)\*(data-out\_nn).\*((1-out\_nn).\*out\_nn)\*(-1\*ones(1,num\_images))')';

Jgrad = [Jgrad\_W\_hidden(:) ; Jgrad\_W\_out(:) ; Jgrad\_b\_hid(:) ; Jgrad\_b\_out(:)]; %rolling the matrices to a single vector

end

%this function takes the weights, adjust the contrast and dimensions of the

%images that the weights represent in the corresponding layer neurons.

function [h, array] = show\_w (we)

we = we - mean(we(:));

[r,c] = size(we);

weight\_height = sqrt(r);

div = divisors(c);

[~, no\_of\_div] = size(div);

im\_height = div(round(no\_of\_div/2));

im\_width = c/im\_height;

patch = zeros(im\_height\*(weight\_height), im\_height\*(weight\_height));

weight\_counter = 1;

for i = 1 : im\_height

for j = 1 : im\_width

if weight\_counter > c

continue;

end

patch((j-1)\*(weight\_height)+(1:weight\_height), (i-1)\*(weight\_height)+(1:weight\_height)) = reshape(we(:, weight\_counter), weight\_height,weight\_height) / max(abs(we(:, weight\_counter)));

weight\_counter = weight\_counter + 1;

end

end

figure;

we = imagesc(patch,[-1 1]);

colormap(gray);

axis image off

drawnow;

end

function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)

% Minimize a continuous differentialble multivariate function. Starting point

% is given by "X" (D by 1), and the function named in the string "f", must

% return a function value and a vector of partial derivatives. The Polack-

% Ribiere flavour of conjugate gradients is used to compute search directions,

% and a line search using quadratic and cubic polynomial approximations and the

% Wolfe-Powell stopping criteria is used together with the slope ratio method

% for guessing initial step sizes. Additionally a bunch of checks are made to

% make sure that exploration is taking place and that extrapolation will not

% be unboundedly large. The "length" gives the length of the run: if it is

% positive, it gives the maximum number of line searches, if negative its

% absolute gives the maximum allowed number of function evaluations. You can

% (optionally) give "length" a second component, which will indicate the

% reduction in function value to be expected in the first line-search (defaults

% to 1.0). The function returns when either its length is up, or if no further

% progress can be made (ie, we are at a minimum, or so close that due to

% numerical problems, we cannot get any closer). If the function terminates

% within a few iterations, it could be an indication that the function value

% and derivatives are not consistent (ie, there may be a bug in the

% implementation of your "f" function). The function returns the found

% solution "X", a vector of function values "fX" indicating the progress made

% and "i" the number of iterations (line searches or function evaluations,

% depending on the sign of "length") used.

%

% Usage: [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)

%

% See also: checkgrad

%

% Copyright (C) 2001 and 2002 by Carl Edward Rasmussen. Date 2002-02-13

%

%

% (C) Copyright 1999, 2000 & 2001, Carl Edward Rasmussen

%

% Permission is granted for anyone to copy, use, or modify these

% programs and accompanying documents for purposes of research or

% education, provided this copyright notice is retained, and note is

% made of any changes that have been made.

%

% These programs and documents are distributed without any warranty,

% express or implied. As the programs were written for research

% purposes only, they have not been tested to the degree that would be

% advisable in any important application. All use of these programs is

% entirely at the user's own risk.

%

% [ml-class] Changes Made:

% 1) Function name and argument specifications

% 2) Output display

%

% Read options

if exist('options', 'var') && ~isempty(options) && isfield(options, 'MaxIter')

length = options.MaxIter;

else

length = 100;

end

RHO = 0.01; % a bunch of constants for line searches

SIG = 0.5; % RHO and SIG are the constants in the Wolfe-Powell conditions

INT = 0.1; % don't reevaluate within 0.1 of the limit of the current bracket

EXT = 3.0; % extrapolate maximum 3 times the current bracket

MAX = 20; % max 20 function evaluations per line search

RATIO = 100; % maximum allowed slope ratio

argstr = ['feval(f, X']; % compose string used to call function

for i = 1:(nargin - 3)

argstr = [argstr, ',P', int2str(i)];

end

argstr = [argstr, ')'];

if max(size(length)) == 2, red=length(2); length=length(1); else red=1; end

S=['Iteration '];

i = 0; % zero the run length counter

ls\_failed = 0; % no previous line search has failed

fX = [];

[f1 df1] = eval(argstr); % get function value and gradient

i = i + (length<0); % count epochs?!

s = -df1; % search direction is steepest

d1 = -s'\*s; % this is the slope

z1 = red/(1-d1); % initial step is red/(|s|+1)

while i < abs(length) % while not finished

i = i + (length>0); % count iterations?!

X0 = X; f0 = f1; df0 = df1; % make a copy of current values

X = X + z1\*s; % begin line search

[f2 df2] = eval(argstr);

i = i + (length<0); % count epochs?!

d2 = df2'\*s;

f3 = f1; d3 = d1; z3 = -z1; % initialize point 3 equal to point 1

if length>0, M = MAX; else M = min(MAX, -length-i); end

success = 0; limit = -1; % initialize quanteties

while 1

while ((f2 > f1+z1\*RHO\*d1) | (d2 > -SIG\*d1)) & (M > 0)

limit = z1; % tighten the bracket

if f2 > f1

z2 = z3 - (0.5\*d3\*z3\*z3)/(d3\*z3+f2-f3); % quadratic fit

else

A = 6\*(f2-f3)/z3+3\*(d2+d3); % cubic fit

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = (sqrt(B\*B-A\*d2\*z3\*z3)-B)/A; % numerical error possible - ok!

end

if isnan(z2) | isinf(z2)

z2 = z3/2; % if we had a numerical problem then bisect

end

z2 = max(min(z2, INT\*z3),(1-INT)\*z3); % don't accept too close to limits

z1 = z1 + z2; % update the step

X = X + z2\*s;

[f2 df2] = eval(argstr);

M = M - 1; i = i + (length<0); % count epochs?!

d2 = df2'\*s;

z3 = z3-z2; % z3 is now relative to the location of z2

end

if f2 > f1+z1\*RHO\*d1 | d2 > -SIG\*d1

break; % this is a failure

elseif d2 > SIG\*d1

success = 1; break; % success

elseif M == 0

break; % failure

end

A = 6\*(f2-f3)/z3+3\*(d2+d3); % make cubic extrapolation

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = -d2\*z3\*z3/(B+sqrt(B\*B-A\*d2\*z3\*z3)); % num. error possible - ok!

if ~isreal(z2) | isnan(z2) | isinf(z2) | z2 < 0 % num prob or wrong sign?

if limit < -0.5 % if we have no upper limit

z2 = z1 \* (EXT-1); % the extrapolate the maximum amount

else

z2 = (limit-z1)/2; % otherwise bisect

end

elseif (limit > -0.5) & (z2+z1 > limit) % extraplation beyond max?

z2 = (limit-z1)/2; % bisect

elseif (limit < -0.5) & (z2+z1 > z1\*EXT) % extrapolation beyond limit

z2 = z1\*(EXT-1.0); % set to extrapolation limit

elseif z2 < -z3\*INT

z2 = -z3\*INT;

elseif (limit > -0.5) & (z2 < (limit-z1)\*(1.0-INT)) % too close to limit?

z2 = (limit-z1)\*(1.0-INT);

end

f3 = f2; d3 = d2; z3 = -z2; % set point 3 equal to point 2

z1 = z1 + z2; X = X + z2\*s; % update current estimates

[f2 df2] = eval(argstr);

M = M - 1; i = i + (length<0); % count epochs?!

d2 = df2'\*s;

end % end of line search

if success % if line search succeeded

f1 = f2; fX = [fX' f1]';

% fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);

s = (df2'\*df2-df1'\*df2)/(df1'\*df1)\*s - df2; % Polack-Ribiere direction

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

d2 = df1'\*s;

if d2 > 0 % new slope must be negative

s = -df1; % otherwise use steepest direction

d2 = -s'\*s;

end

z1 = z1 \* min(RATIO, d1/(d2-realmin)); % slope ratio but max RATIO

d1 = d2;

ls\_failed = 0; % this line search did not fail

else

X = X0; f1 = f0; df1 = df0; % restore point from before failed line search

if ls\_failed | i > abs(length) % line search failed twice in a row

break; % or we ran out of time, so we give up

end

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

s = -df1; % try steepest

d1 = -s'\*s;

z1 = 1/(1-d1);

ls\_failed = 1; % this line search failed

end

% if exist('OCTAVE\_VERSION')

% fflush(stdout);

% end

disp("Iteration " + i);

end

%plot(1:length(fX),fX);

% fprintf('\n');

end

function sig = sigmoid(x)

sig = 1 ./ (1+exp(-x));

end