EEE 543: Neural Networks -Final Project Report

Oytun Gunes, ID: 20901170, Mucahit Gumus ID: 20900664

July 20, 2019

Abstract

In this report 10 class classification task is done on commonly used CIFAR-10 data. A data analysis is performed before the design, cross correlation of the images are examined. Two different methods are implemented: conjugate gradient cost function and line search and single layer convolutional neural network (CNN). Many hyperparameters are optimized to obtain accuracies. In the results part cross entropy cost, validation accuracy and confusion matrices are illustrated. 30 percent test accuracy is obtained with the first method. CNN increased the test accuracy up to 40 percent.

1 Introduction

In the last five years CNN (Convolutional Neural Networks) has achieved outperforming results compared to standard classification algorithms.This has been started with the breaktrough of ImageNet classification results [1]. Soon after, Donahue et al. [2] have display the results of this network on several image dataset. They have found that CNN is an effective feature extractor. kaggle.com, a popular website in machine learning, has organized a competition on CIFAR-10 data. It is very interesting to work on a popular image data and perform 10 class classification. Our expectation on the task is to obtain a high accuracy of course. However, most people in the literature have constructed a complex and at least 3-5 layers on libraries on CNN structure to obtain a higher accuracy, whereas in our task the challenge is to code a complex architecture without using libraries and toolboxes and optimize parameters on a big data.

2 Methods

2.1 Analysis on Data

In this project the aim is to classify objects and animals correctly given training data. The data given is a popular data in the literature known as CIFAR-10. It consists of 50000 training and 10000 test data. There are 10 classes in the data. The sample images from each class is shown below.

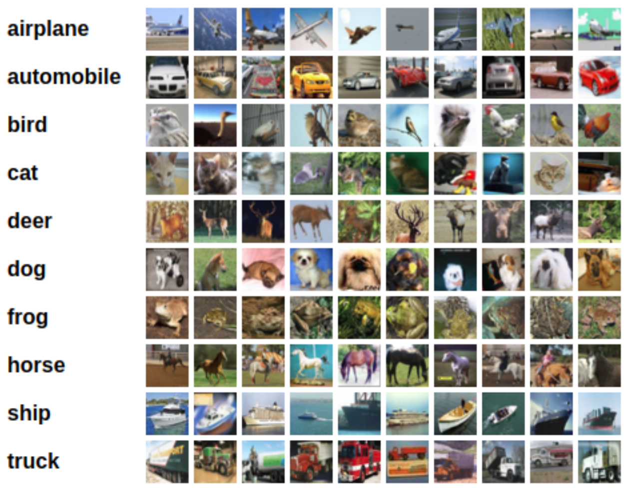

Figure 1: Sample Images from the Data

The correlation between the image is shown below.

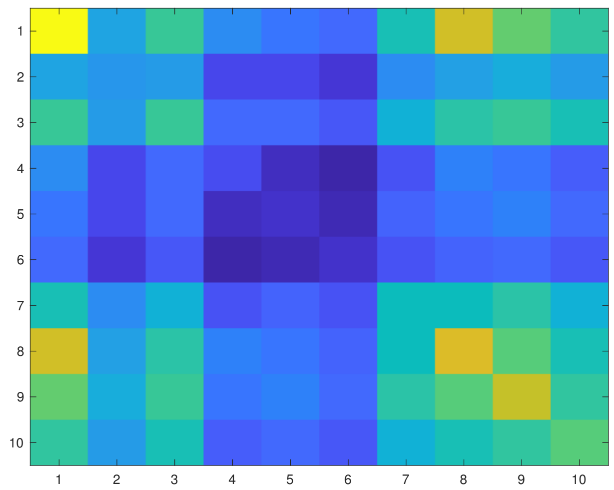

Figure 2: Visualization of Correlation Matrix of different classes

The correlation matrix above explains the cross correlation between the images. In the diagonals it explains the autocorrelation of the images which are high as expected. Off diagonal entries of the correlation matrix is also high which means there is high correlation among different classes. Therefore, classification over the dataset becomes challenging.

2.2 Training with conjugate gradient cost function and line search

We firstly tried a fully connected multi-layer neural network structure in order for classifying the data of 10 kind of objects. In this multi-layer structure, we have 40 neurons in the first hidden layer, 20 neurons in the second hidden layer and 10 neurons are used in the output layer in order to classify 10 objects. At the output, we implement softmax decision and calculate the cross entropy as an error function and using backpropogation we found the update rules for the weights and biases. The error is defined as:

$$E = -\sum_{i=1}^{10} (d_i) log(o_i)$$

where d is the desired vector, the correct object's index is 1 and others' are 0. The derivative of softmax function is found and as $(o_i - d_i)$. The gradient of the output layer is found as:

$$\frac{\partial E}{\partial W_{ji}} = \frac{\partial E}{\partial z_i} \frac{\partial z_i}{\partial W_{ji}}$$

$$\frac{\partial E}{\partial W_{ji}} = (o_i - d_i) h_j$$

where h is the output of the hidden layer.

The gradient of the second hidden layer is found again by the chain rule as:

$$\frac{\partial E}{\tilde{W}_{aj}} = z_a h_j (1 - h_j) \lambda \sum_{i=1}^{10} (o_i - d_i) W_{ji}$$

Continuing in the same manner, gradient for the first layer is also found.

Then, those gradient values and the current value of the error is used in the fmincg function to calculate the updates. In fmincg, 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.

Before training, we firstly preprocess data in order to protect from saturation and decreasing complexity. Hence, we modify the pictures to gray scale and subtract the mean. Lastly, we zip the data to [-3std, 3 std] interval.

2.3 Convolutional Neural Networks (CNN) Explanation [3]

Convolutional Network as mentioned above is a highly effective method in classification. It solves some of the problems in feed-forward neural network approach. One of them is fully connected structure has an increasing number of parameters since each node in layer L is connected to a node in layer L-1. It becomes unmanagable to work on with a huge number of parameters. Secondly, computing the linear activations of the hidden units would be computationally costly. Local connectivity of CNN solves these problems.

Local Connectivity:

Each hidden unit is connected only to a subregion(patch) of the image. CNN process patches independently.

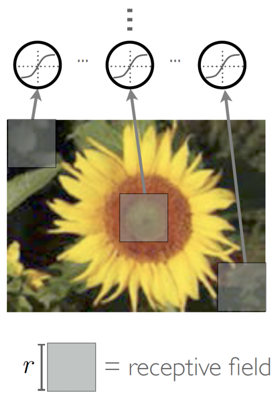

Figure 3: Patches in the Image

In addition, units are connected to all channels. 1 channel if grayscale image, 3 channels (R, G, B) if color image as shown below.

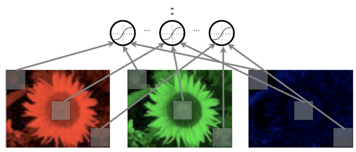

Figure 4: Channels

Parameter Sharing:

In CNN it shares a matrix of parameters across certain units. Units organized into the same feature map share parameters. Hidden units within a feature map cover different positions in the image. This reduces the number of parameters and it will help to extract features at every position. Feature maps of the image is shown below.

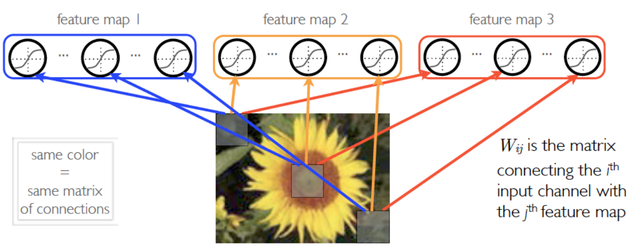

Figure 5: Feature Maps in the Image

Convolution Layer:

Feature maps described above is computed with a discrete convolution ($*$) of a kernel matrix k_{ij} which is the hidden weights matrix W_{ij} with its rows and columns flipped.

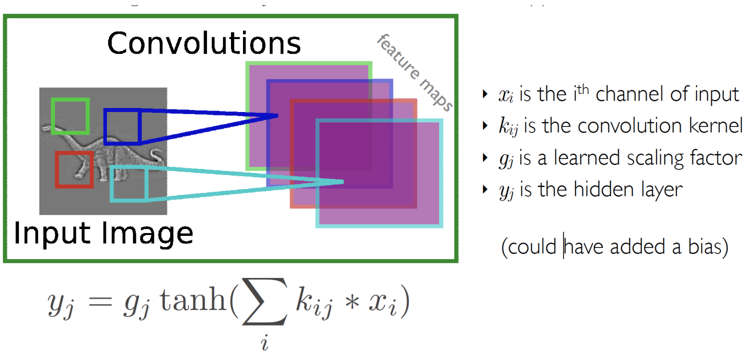

Figure 6: Obtaining Feature maps

The convolution operation of an image x with a kernel k is as below:

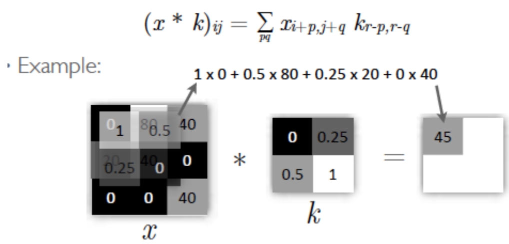

Figure 7: Convolution Operation on the Image

Convolution operation in the image can be considered as the weighted sum between two signals. In image processing the convolution of the location (x,y) is done by extracting a patch of kxk dimension small image. Then filter this patch with a filter which has dimensions also kxk by multiplying elementwise and adding the values and obtain the result. Similarly this operation should be done for all kxk pairs in the image by sliding one by one. After the convolution layer it passes through an activation function such as sigmoid or commonly Relu.
Max Pooling Layer: Max pooling layer is done after the convolution layer in order to reduce further the number of parameters and introduces invariance to local translations. As an illustration 2x2 max pooling is done on the image below. The maximum element is selected.

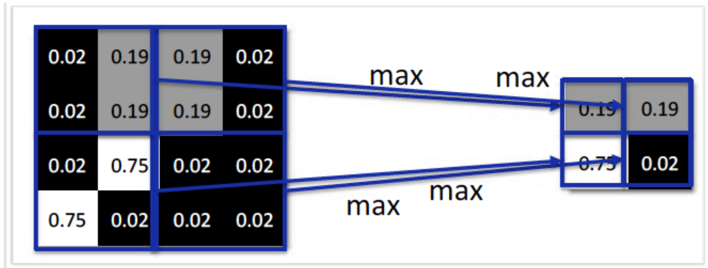

Figure 8: Max pooling Illustration

Output Layer: Output layer is a regular, fully connected layer with softmax non-linearity output provides an estimate of the conditional probability of each class. The network is trained by stochastic gradient descent.
Gradient of Convolution Layer:
We have used an error function of cross-entropy. Using the backpropogation we have updated the kernels and output weights. Let l be the loss function. For convolution operation $y_j = x_i * k_{ij}$ the gradient for x_i is :

$$\nabla_{x_i} l = \sum_j (\nabla_{y_j} l) \underline{*} (W_{ij})$$

where * is the convolution with zero padding and x_i is the row/column flipped version of x_i .

Gradient for W_{ij}:

$$\nabla_{W_{ij}} l = (\nabla_y l) * \tilde{x}_i$$

Gradient of Pooling Layer:
Let l be the loss function:
For max pooling operation $y_{ijk} = \max_{p,q} x_{i,j+p,k+q}$ the gradient for x_{ijk}.

$$\nabla_{x_{ijk}} l = 0 \text{ except for } \nabla_{x_{i,j+p',k+q'}} l = \nabla_{y_{ijk}} l$$

The overall structure of CNN is to alternate between the convolution and pooling layers as shown below:

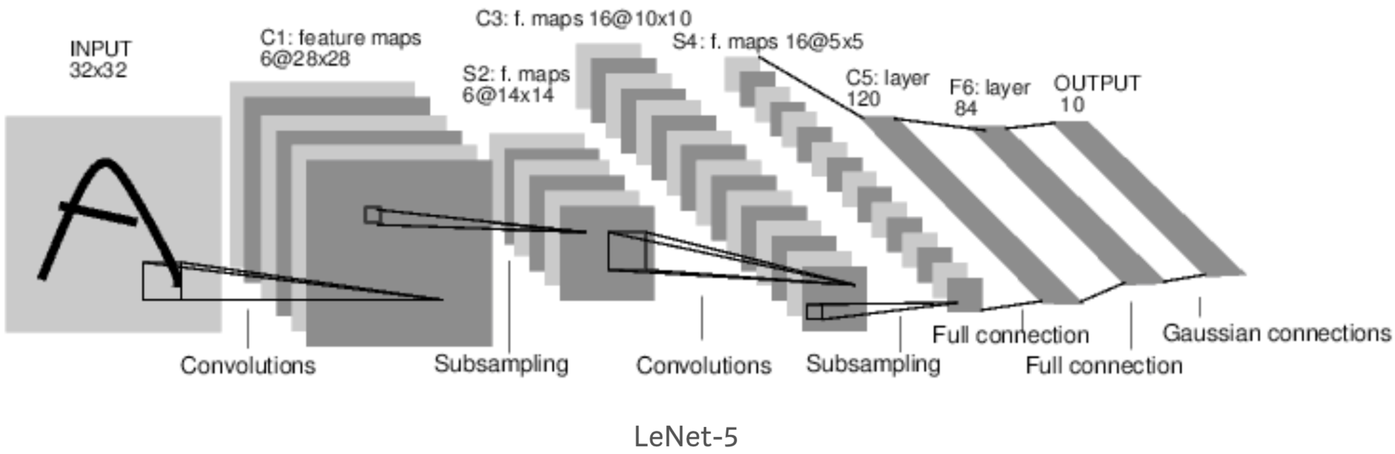

Figure 9: Overall CNN architecture

Simulation Setup:
Our simulation setup for CNN is as follows:
Single convolutional layer with 16 kernels with a size of 5x5.
Sigmoid layer with $\lambda = 1$
Max pooling layer with 2x2
Fully connected layer 3136x10
Output Layer 10x1
Softmax Layer
Parameter Selection:
Parameters are selected using the toolbox of MATLAB since it is computationally efficient.

3 Results

3.1 Results of conjugate gradient cost function and line search

In figure 10, we have visualized the first hidden layer weights as images. We have observed that the weights are distributed differently, however we could not observe specific features. That is because training with the fully connected structure and line seaerch methedology is not successful enough to extract hidden features of the objects although these neural structure achieves other classification problems such as letter recognition well.

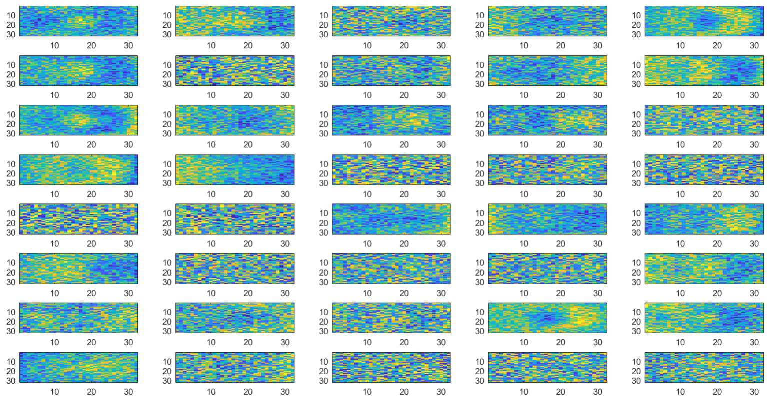

Figure 10: Visualization of hidden weights after training.

In figure 11 and 12, we have visualized confusion matrix for train and test data. We have observed that confusion matrices are similar for both of the dataset which shows that there is no overfitting with the learning. It also demonstrates that decision of cat is always less than other classes hence algorithm decided on dog mostly for cat images. Accuracy in classifying dog and ship images is higher than other 8 classes.

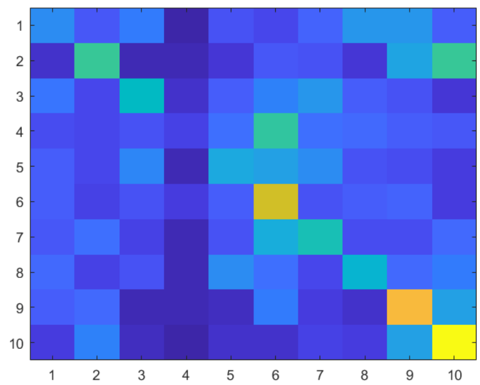

Figure 11: Confusion Matrix after training.

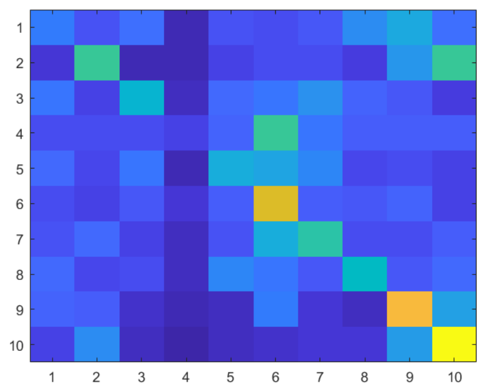

Figure 12: Confusion Matrix after testing.

As it is also obvious from the confusion matrix, the classification with this method does not achieve a good rate hence accuracy is around 30 %.

3.2 Results of CNN

In this section we present different types of results to demonstrate training and validation performance of CNN algorithm described in methods section. Training 5 epochs lasted around 50 minutes without using GPU.

Firstly, cross entropy cost over 5 epochs are shown below. It can be observed from the figure that training provides with decrease in cross entropy cost.

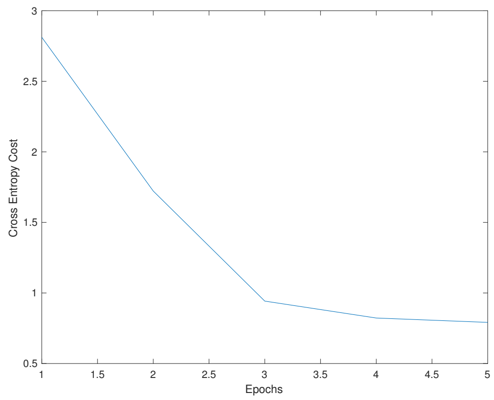

Figure 13: Cross Entropy Cost over epochs

Secondly, validation accuracy over epochs are shown below. It can be seen that validation accuracy increases up to 40%. It is obvious from figure 13 and ?? that validation accuracy and training cost are positively related.

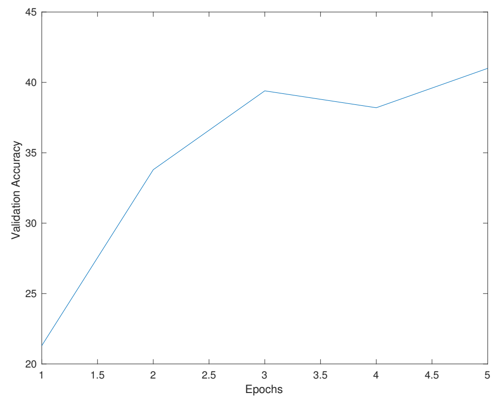

Figure 14: Validation Accuracy over epochs

Thirdly the kernels of the convolution layers after training are shown below. Each kernel has extracted a different feature as, there are bright points at different pixels in the figure.

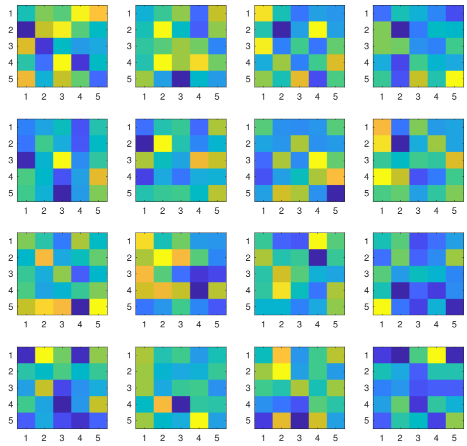

Figure 15: Final kernels of the convolution layer

Finally confusion matrix of the test data is shown below. As expected diagonals are higher than the off-diagonals which means that training provides to decide on true class for all of the 10 classes.

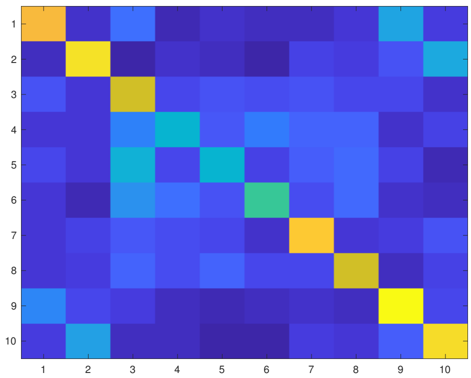

Figure 16: Confusion Matrix of test data

Implemented CNN structure provides 41 percent test accuracy.

4 Discussion

CIFAR-10 dataset is a challenging data as we have seen from the literature since it has objects in the image. Some people have obtained accuracies above 90 with a very complex structure of CNN with the help of phyton libraries. They have mostly used 3 layer convolution with dropout and batch normalization layers. Our design of single layer CNN reachs up to 40 percent which is comparably reasonable result for a single layer CNN. Optimizing hyper-parameters was a really tough challenge since we had around total of 10 parameters such as initializing weights, kernel size, max pooling size, learning rate, λ, momentum constant $alpha$ and activation function type. 50 minutes of training time did not allow us to try many parameters to obtain a higher accuracy. We have also tried 2 layer CNN since it takes around 1.5 hours to train all the data with CPU we could not optimize the parameters enough to obtain a higher accuracy compared to single layer CNN.

The first method we have used has a fully connected neural network structure and use the given fmincg function before, gave around 30 percent accuracy. Training time of this design is around 30 minutes for 5 epochs.

Of course, there are many ways to increase the test accuracy. One of them is to add data augmentation before the input layer and add more data to training data. Another one is to add rectification layer which performs well in object recognition which makes the training translation invariant.

References

[1] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.

[2] Donahue, Jeff, et al. "Decaf: A deep convolutional activation feature for generic visual recognition." International conference on machine learning. 2014.

[3] Lecture Notes of Neural Network Bilkent University 2017-2018 Fall.

Appendix: MATLAB CODES

```
clear;
close all;
clc;
%loading data and normalization
tic;
load('final_project_dataset.mat');
pixelNum=32;
for i=1:size(testdat,1)
    aa=testdat(i,:);
    testdatShaped(:,:,:,i)=reshape(aa, [pixelNum pixelNum 3]);
end
for i=1:size(traindat,1)
    aa=traindat(i,:);
    traindatShaped(:,:,:,i)=reshape(aa, [pixelNum pixelNum 3]);
end
% preprocess test data for w-b and normalize
nbWTrainData= zeros(pixelNum,pixelNum,size(traindatShaped,4));
for i=1:size(traindatShaped,4)
    nbWTrainData(:,:,i)=rgb2gray(traindatShaped(:,:,:,i));
end
% preprocess train data for w-b and normalize
nbWTestData= zeros(pixelNum,pixelNum,size(testdatShaped,4));
for i=1:size(testdatShaped,4)
    nbWTestData(:,:,i)=rgb2gray(testdatShaped(:,:,:,i));
end
nbWTrainData=double(nbWTrainData);
nbWTestData=double(nbWTestData);
%modify train and test labels by increasing 1
trainlbl = trainlbl +1;
testlbl = testlbl + 1;

%initialize weights and kernel
ker1L=5;
ker1Num=16;
pl1=2;
numOfHid= ker1Num*((pixelNum-ker1L+1)/pl1)^2;
numOfOut=10;
kerCons1= sqrt(6/(ker1L*ker1L+900)); % how to get 4 is ker1L^2*1/pl1^2
ker1= -1*kerCons1+ (kerCons1*2)*rand(ker1L,ker1L,ker1Num);
ker1=0.05*randn(ker1L,ker1L,ker1Num);
biasKer1=-1*kerCons1+ (kerCons1*2)*rand(pixelNum-ker1L+1,pixelNum-ker1L+1,
    ker1Num);
biasKer1=0.05*randn(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
weightConsOut= sqrt(6/(numOfHid+900)); %how to get 200
```

```
weightOut= -1*weightConsOut+ (weightConsOut*2)*rand(numOfHid,numOfOut);
biasOut=-1*weightConsOut+ (weightConsOut*2)*rand(1,numOfOut);
lambda=1;
eta=0.0001;
alpha=0;
toc;
epochNum=5;
miniBatchNum=128;


confusiontest=zeros(numOfOut,numOfOut);

accEpoch=zeros(1,epochNum);
CostEpoch=zeros(1,epochNum);

for epochIter=1:epochNum
    eta=0.0001*(0.5^(epochIter-1));% decrease learning rate over epochs
    tic;
    picSequence = randperm(length(trainlbl));
    prevWODel= zeros(numOfHid,numOfOut);
    prevBODel=zeros(1,numOfOut);
    prevK1Del=zeros(ker1L,ker1L,ker1Num);
    prevBiasKer1Del=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
    % mini-batch
    for batchIter=1:floor(length(picSequence)/miniBatchNum)
        storeWODel= zeros(numOfHid,numOfOut);
        storeBODel=zeros(1,numOfOut);
        storeK1Del=zeros(ker1L,ker1L,ker1Num);
        storeBiasKer1Del=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
        for mBIter=1:miniBatchNum
            X=nbWTrainData(:,:,picSequence((batchIter-1)*miniBatchNum+mBIter));

            %convolutional layer
            conv1Out=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
            for ker1Iter=1:ker1Num
                conv1Out(:,:,ker1Iter) = conv2(X,ker1(:,:,ker1Iter),'valid');
            end
            conv1OutAct=1./(1+exp(-lambda*(conv1Out-biasKer1)));

            % max pooling layer
            pl1Out= zeros((pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,ker1Num);
            pl1OutIndex=zeros(2,(pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,ker1Num);
            for ker1Iter=1:ker1Num
                for i=1:(pixelNum-ker1L+1)/pl1
                    for k=1:(pixelNum-ker1L+1)/pl1
                        dumy= conv1OutAct((i-1)*pl1+1:i*pl1 ,(k-1)*pl1+1:k*pl1 ,ker1Iter);
                        [maxVal, maxInd] = max(dumy(:));
                        [Irow, Icol] = ind2sub(size(dumy),maxInd);
                        pl1Out(i,k,ker1Iter) = maxVal;
                        pl1OutIndex(1,i,k,ker1Iter)=Irow;
                        pl1OutIndex(2,i,k,ker1Iter)=Icol;
                    end
                end
            end
            %fully connected layer
            hiddenOut=pl1Out(:);
            neuralOut=transpose(hiddenOut)*weightOut-biasOut;
            % output layer
```

```
            sumOut=sum(exp(neuralOut));
            neuralOutAct= exp(neuralOut)*1/(sumOut);
            %backpropogation
            d=zeros(1,numOfOut);
            d(1,trainlbl(picSequence((batchIter-1)*miniBatchNum+mBIter)))=1;
            gradOut= neuralOutAct - d;
            gradHidden= weightOut*(transpose(gradOut));
            gradpl1=reshape(gradHidden, [(pixelNum-ker1L+1)/pl1 (pixelNum-ker1L+1)/pl1 ker1Num]);

            gradconv1=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
            for ker1Iter=1:ker1Num
                for i=1:(pixelNum-ker1L+1)/pl1
                    for k=1:(pixelNum-ker1L+1)/pl1
                        indRow=(i-1)*pl1+pl1OutIndex(1,i,k,ker1Iter);
                        indCol=(k-1)*pl1+pl1OutIndex(2,i,k,ker1Iter);
                        gradconv1(indRow,indCol,ker1Iter)=gradpl1(i,k,ker1Iter)*conv1OutAct(indRow,indCol,ker1Iter)*(1-conv1OutAct(indRow,indCol,ker1Iter));
                    end
                end
            end
            storeWODel=storeWODel -1*eta*(hiddenOut)*gradOut;
            weightOut =weightOut+  -1*eta*(hiddenOut)*gradOut + alpha*prevWODel;
            prevWODel=alpha*prevWODel+-1*eta*(hiddenOut)*gradOut;
            storeBODel=storeBODel -1*eta*(-1)*gradOut;
            biasOut=biasOut+ -1*eta*(-1)*gradOut;
            prevBODel=alpha*prevBODel -1*eta*(-1)*gradOut;

            ker1Update=zeros(ker1L,ker1L,ker1Num);
            for ker1Iter=1:ker1Num
                for i=1:ker1L
                    for k=1: ker1L
                        ind1=ker1L-i+1;
                        ind2=pixelNum-i+1;
                        ind3=k;
                        ind4=k+size(conv1Out,1)-1;
                        ker1Update(:,:,ker1Iter)=ker1Update(:,:,ker1Iter)-1*(eta)*sum(sum(X(ind1:ind2,ind3:ind4).*gradconv1(:,:,ker1Iter))); %*1/((pixelNum-ker1L+1)^2)
                    end
                end
            end
            storeK1Del=storeK1Del+ker1Update;
            storeBiasKer1Del=storeBiasKer1Del -1*eta*(-1)*gradconv1;
            aaaaa=0;
        end
        % update the kernels and weights
        ker1=ker1+(1/miniBatchNum)*storeK1Del+alpha*prevK1Del;
        prevK1Del=alpha*prevK1Del+(1/miniBatchNum)*storeK1Del;
        biasKer1=biasKer1+(1/miniBatchNum)*storeBiasKer1Del + alpha*prevBiasKer1Del;
        prevBiasKer1Del=prevBiasKer1Del++(1/miniBatchNum)*storeBiasKer1Del;
        weightOut =weightOut+  (1/miniBatchNum)*storeWODel + alpha*prevWODel;
        prevWODel=alpha*prevWODel+(1/miniBatchNum)*storeWODel;
        biasOut=biasOut+ (1/miniBatchNum)*storeBODel+ alpha*prevBODel;
        prevBODel=alpha*prevBODel + (1/miniBatchNum)*storeBODel;

    end
    toc;
```

```matlab
    %test validation data
    tic;
    testError=0;
    testShuf=randperm(size(nbWTrainData,3));
    for testIter=1:1000
        X=nbWTrainData(:,:,testShuf(testIter));
        conv1Out=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
        for ker1Iter=1:ker1Num
            conv1Out(:,:,ker1Iter) = conv2(X,ker1(:,:,ker1Iter),'valid');
        end
        conv1OutAct=1./(1+exp(-lambda*(conv1Out-biasKer1)));
        pl1Out= zeros((pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,ker1Num);
        pl1OutIndex=zeros(2,(pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,
            ker1Num);
        for ker1Iter=1:ker1Num
            for i=1:(pixelNum-ker1L+1)/pl1
                for k=1:(pixelNum-ker1L+1)/pl1
                    dumy= conv1OutAct((i-1)*pl1+1:i*pl1 ,(k-1)*pl1+1:k*pl1 ,
                        ker1Iter);
                    [maxVal, maxInd] = max(dumy(:));
                    [Irow, Icol] = ind2sub(size(dumy),maxInd);
                    pl1Out(i,k,ker1Iter) = maxVal;
                    pl1OutIndex(1,i,k,ker1Iter)=Irow;
                    pl1OutIndex(2,i,k,ker1Iter)=Icol;
                end
            end
        end
        hiddenOut=pl1Out(:);
        neuralOut=transpose(hiddenOut)*weightOut-biasOut;
        sumOut=sum(exp(neuralOut));
        neuralOutAct= exp(neuralOut)*1/(sumOut);
        d=trainlbl(testShuf(testIter));
        [mV, mI]= max(neuralOutAct);
        if (mI ~= d)
            testError=testError+1;
        end
        dvec=zeros(1,numOfOut);
        dvec(1,d)=1;
        CostEpoch(1,epochIter)=-dvec*transpose(log(neuralOutAct));

    end
    testError*(100/1000)
    accEpoch(1,epochIter)=testError*(100/1000);

    toc;
end

% Calculating test accuracy

testError=0;
for testIter=1:size(nbWTestData,3)
    X=nbWTestData(:,:,testIter);
    conv1Out=zeros(pixelNum-ker1L+1,pixelNum-ker1L+1,ker1Num);
    for ker1Iter=1:ker1Num
        conv1Out(:,:,ker1Iter) = conv2(X,ker1(:,:,ker1Iter),'valid');
    end
    conv1OutAct=1./(1+exp(-lambda*(conv1Out-biasKer1)));
    pl1Out= zeros((pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,ker1Num);
    pl1OutIndex=zeros(2,(pixelNum-ker1L+1)/pl1,(pixelNum-ker1L+1)/pl1,ker1Num);
    for ker1Iter=1:ker1Num
        for i=1:(pixelNum-ker1L+1)/pl1
```

```
            for k=1:(pixelNum-ker1L+1)/pl1
                dumy= conv1OutAct((i-1)*pl1+1:i*pl1 ,(k-1)*pl1+1:k*pl1 ,ker1Iter);
                [maxVal, maxInd] = max(dumy(:));
                [Irow, Icol] = ind2sub(size(dumy),maxInd);
                pl1Out(i,k,ker1Iter) = maxVal;
                pl1OutIndex(1,i,k,ker1Iter)=Irow;
                pl1OutIndex(2,i,k,ker1Iter)=Icol;
            end
        end
    end
    hiddenOut=pl1Out(:);
    neuralOut=transpose(hiddenOut)*weightOut-biasOut;
    sumOut=sum(exp(neuralOut));
    neuralOutAct= exp(neuralOut)*1/(sumOut);
    d=testlbl(testIter);
    [mV, mI]= max(neuralOutAct);
    %confusion matrix
    confusiontest(d,mI)=confusiontest(d,mI)+1;
    if (mI ~= d)
        testError=testError+1;
    end

end
testerrorfinal = testError*(100/size(nbWTestData,3));
% plotting  curves
plot(CostEpoch)
xlabel 'Epochs';
ylabel 'Cross Entropy Cost';

figure
accuracyepochs=100-accEpoch;
plot(accuracyepochs)
xlabel 'Epochs';
ylabel 'Validation Accuracy ';

figure
for i=1:size(ker1,3)
    subplot(4,4,i)
    imagesc(ker1(:,:,i));
end



imagesc(confusiontest)
```

```
clear;
close all;
clc;
%loading data and normalization
load('final_project_dataset.mat');
pixelNum=32;
for i=1:size(testdat,1)
    aa=testdat(i,:);
    testdatShaped(:,:,:,i)=reshape(aa, [pixelNum pixelNum 3]);
end
for i=1:size(traindat,1)
    aa=traindat(i,:);
    traindatShaped(:,:,:,i)=reshape(aa, [pixelNum pixelNum 3]);
end
% preprocess test data for w-b and normalize
```

```
bWTestData= zeros(pixelNum,pixelNum,size(testdatShaped,4));
nbWTestData= zeros(pixelNum,pixelNum,size(testdatShaped,4));
for i=1:size(testdatShaped,4)
    bWTestData(:,:,i)= 0.2126*testdatShaped(:,:,1,i) + 0.7152*testdatShaped(:,:,2,i) + 0.0722*testdatShaped(:,:,3,i);
    nbWTestData(:,:,i)=bWTestData(:,:,i)-ones(pixelNum,pixelNum)*sum(sum(bWTestData(:,:,i)))*1/(pixelNum*pixelNum);
end
stdData=std(nbWTestData(:));
nbWTestData((nbWTestData > 3*stdData))=3*stdData;
nbWTestData((nbWTestData < -3*stdData))=-3*stdData;
nbWTestData=nbWTestData*(4/(30*stdData))+0.5;
% preprocess train data for w-b and normalize
bWTrainData= zeros(pixelNum,pixelNum,size(traindatShaped,4));
nbWTrainData= zeros(pixelNum,pixelNum,size(traindatShaped,4));
for i=1:size(traindatShaped,4)
    bWTrainData(:,:,i)= 0.2126*traindatShaped(:,:,1,i) + 0.7152*traindatShaped(:,:,2,i) + 0.0722*traindatShaped(:,:,3,i);
    nbWTrainData(:,:,i)=bWTrainData(:,:,i)-ones(pixelNum,pixelNum)*sum(sum(bWTrainData(:,:,i)))*1/(pixelNum*pixelNum);
end
stdData2=std(nbWTrainData(:));
nbWTrainData((nbWTrainData > 3*stdData2))=3*stdData2;
nbWTrainData((nbWTrainData < -3*stdData2))=-3*stdData2;
nbWTrainData=nbWTrainData*(4/(30*stdData2))+0.5;
%modify train and test labels by increasing 1
 trainlbl = trainlbl +1;
 testlbl = testlbl + 1;
%save processed data to call in CEcost
save('dataProcessed.mat','nbWTestData','nbWTrainData', 'testlbl','trainlbl');
%initialization of some parameters
numOfHid=40;
numOfOut=10;
lamda=1;
%uniform random initial of weights
 weightConsHid= sqrt(6/(pixelNum*pixelNum+numOfOut));
 weightHid = -1*weightConsHid+ (weightConsHid*2)*rand(pixelNum,pixelNum,numOfHid);
 biasHid=-1*weightConsHid+ (weightConsHid*2)*rand(numOfHid,1);
 weightConsOut= sqrt(6/(numOfHid));
 weightOut= -1*weightConsOut+ (weightConsOut*2)*rand(numOfOut,numOfHid);
 biasOut=-1*weightConsOut+ (weightConsOut*2)*rand(numOfOut,1); %reshaping all weights for function
W=[reshape(weightHid,[pixelNum*pixelNum*numOfHid 1]);biasHid;reshape(weightOut,[numOfHid*numOfOut 1]); biasOut];
 conjIter=1;
 for i=1:conjIter
    %call of fmingrad
     [W, fX, iterations] = fmincg('CEcost',W,optimset('MaxIter',50));
end
%reshaping weights
 weightHid= reshape(W(1:pixelNum*pixelNum*numOfHid), [pixelNum pixelNum numOfHid]);
 biasHid=W(pixelNum*pixelNum*numOfHid+1:pixelNum*pixelNum*numOfHid+numOfHid);
 weightOut= reshape(W(pixelNum*pixelNum*numOfHid+numOfHid+1:pixelNum*pixelNum*numOfHid +numOfHid +numOfHid*numOfOut), [numOfOut numOfHid]);
 biasOut=W(pixelNum*pixelNum*numOfHid +numOfHid +numOfHid*numOfOut+1:length(W));
%ploting weights as image for all hidden neurons
 figure(1);
 for hiddenIter=1:numOfHid
    subplot(8,5,hiddenIter);
```

```
        imagesc(weightHid(:,:,hiddenIter));
    end
    %cost and confusion matrix calculation for train and test data
    Jtrain=0;
    Ctrain=zeros(10,10);
    Ctest=zeros(10,10);
    for trainIter=1:length(trainlbl)
        X=nbWTrainData(:,:,trainIter);
        hiddenOut=zeros(numOfHid,1);
        for hiddenIter=1:numOfHid
            hiddenOut(hiddenIter,1)=sum(sum(X.*weightHid(:,:,hiddenIter)))-biasHid(hiddenIter,1);
        end
        hiddenOutAct=1./(1+exp(-1*hiddenOut));
        neuralOut=weightOut*hiddenOutAct-biasOut;
        neuralOutAct=1./(1+exp(-1*neuralOut));
        [m ind] = max(neuralOutAct);
        Ctrain(trainlbl(trainIter),ind)=Ctrain(trainlbl(trainIter),ind)+1;
        d=zeros(numOfOut,1);
        d(trainlbl(trainIter))=1;
        Jtrain=Jtrain+transpose(-1*d)*log(neuralOutAct)-transpose(1-d)*(log(1-neuralOutAct));
    end
    Jtest=0;
    for testIter=1:length(testlbl)
        X=nbWTestData(:,:,testIter);
        hiddenOut=zeros(numOfHid,1);
        for hiddenIter=1:numOfHid
            hiddenOut(hiddenIter,1)=sum(sum(X.*weightHid(:,:,hiddenIter)))-biasHid(hiddenIter,1);
        end
        hiddenOutAct=1./(1+exp(-1*hiddenOut));
        neuralOut=weightOut*hiddenOutAct-biasOut;
        neuralOutAct=1./(1+exp(-1*neuralOut));
        [m ind] = max(neuralOutAct);
        Ctest(testlbl(testIter),ind)=Ctest(testlbl(testIter),ind)+1;
        d=zeros(numOfOut,1);
        d(testlbl(testIter))=1;
        Jtest=Jtest+transpose(-1*d)*log(neuralOutAct)-transpose(1-d)*(log(1-neuralOutAct));
    end
    %plotting confusion matrix and normalizing cost results
    Jtrain=Jtrain/length(trainlbl);
    Jtest=Jtest/length(testlbl);
    figure(2);
    imagesc(Ctest, [0 max(max(Ctest))]);
    figure(3);
    imagesc(Ctrain, [0 max(max(Ctrain))]);
    disp ('Cross entropy  cost for test data: '); disp(num2str(Jtest));
    disp ('Cross entropy  cost for train data: '); disp(num2str(Jtrain));
```

```
function [JOut,JGradOut] = CEcost(Weights)
%initialize some params
    aa=Weights;
    pixelNum=32;
    numOfHid=40;
    numOfOut=10;
    lamda=1;
    load('dataProcessed.mat');
    %taking weights for use in iteration
    weightHid= reshape(Weights(1:pixelNum*pixelNum*numOfHid), [pixelNum pixelNum
```

```
numOfHid]);
    biasHid=Weights(pixelNum*pixelNum*numOfHid+1:pixelNum*pixelNum*numOfHid+numOfHid);
    weightOut= reshape(Weights(pixelNum*pixelNum*numOfHid+numOfHid+1:pixelNum*pixelNum*numOfHid +numOfHid +numOfHid*numOfOut), [numOfOut numOfHid]);
    biasOut=Weights(pixelNum*pixelNum*numOfHid +numOfHid +numOfHid*numOfOut+1:length(Weights));
    picSequence = randperm(length(trainlbl));
    J=0; %initial cost before iteration
    storeHiddenDer= zeros(pixelNum,pixelNum,numOfHid);
    storeHidBiasDer=zeros(numOfHid,1);
    storeOutDer=zeros(numOfOut,numOfHid);
    storeOutBiasDer=zeros(numOfOut,1);
    for batchIter=1:length(picSequence) %batch iter
        X=nbWTrainData(:,:,picSequence(batchIter)); %shuffling data
        %output and gradient calculations
        hiddenOut=zeros(numOfHid,1);
        for hiddenIter=1:numOfHid
            hiddenOut(hiddenIter,1)=sum(sum(X.*weightHid(:,:,hiddenIter)))-biasHid(hiddenIter,1);
        end
        hiddenOutAct=1./(1+exp(-1*hiddenOut));
        neuralOut=weightOut*hiddenOutAct-biasOut;
        neuralOutAct=1./(1+exp(-1*neuralOut));
        d=zeros(numOfOut,1);
        d(trainlbl(picSequence(batchIter)))=1;

        derivativeFOut = neuralOutAct.*(1-neuralOutAct);
        gradOut=((-1*d).*(1./neuralOutAct)+(1-d).*(1./(1-neuralOutAct))).*derivativeFOut;
        %gradOut= neuralOutAct - d;
        derivativeHidden= hiddenOutAct.*(1-hiddenOutAct);
        gradHidden= derivativeHidden.*transpose((transpose(gradOut)*weightOut));
        %summing cost and derivatives
        J=J+transpose(-1*d)*log(neuralOutAct)-transpose(1-d)*(log(1-neuralOutAct));
        %J=J+transpose(-1*d)*log(neuralOutAct);
        storeOutDer= storeOutDer+ gradOut*transpose(hiddenOutAct);
        for hiddenIter=1:numOfHid
            storeHiddenDer(:,:,hiddenIter)=storeHiddenDer(:,:,hiddenIter)+gradHidden(hiddenIter,1)*X;
        end
        storeHidBiasDer= storeHidBiasDer-1*gradHidden;
        storeOutBiasDer=storeOutBiasDer+-1*gradOut;
    end
    %normalizing
    storeOutDer=storeOutDer*(1/length(picSequence));
    storeHiddenDer=storeHiddenDer*(1/length(picSequence));
    storeHidBiasDer=storeHidBiasDer*(1/length(picSequence));
    storeOutBiasDer=storeOutBiasDer*(1/length(picSequence));
    JOut=J*(1/length(picSequence));
    JGradOut=[reshape(storeHiddenDer,[pixelNum*pixelNum*numOfHid 1]);storeHidBiasDer;reshape(storeOutDer,[numOfHid*numOfOut 1]);storeOutBiasDer];
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.
%
% 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)
%
% (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.

% 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
  end
  fprintf('\n');
end
```