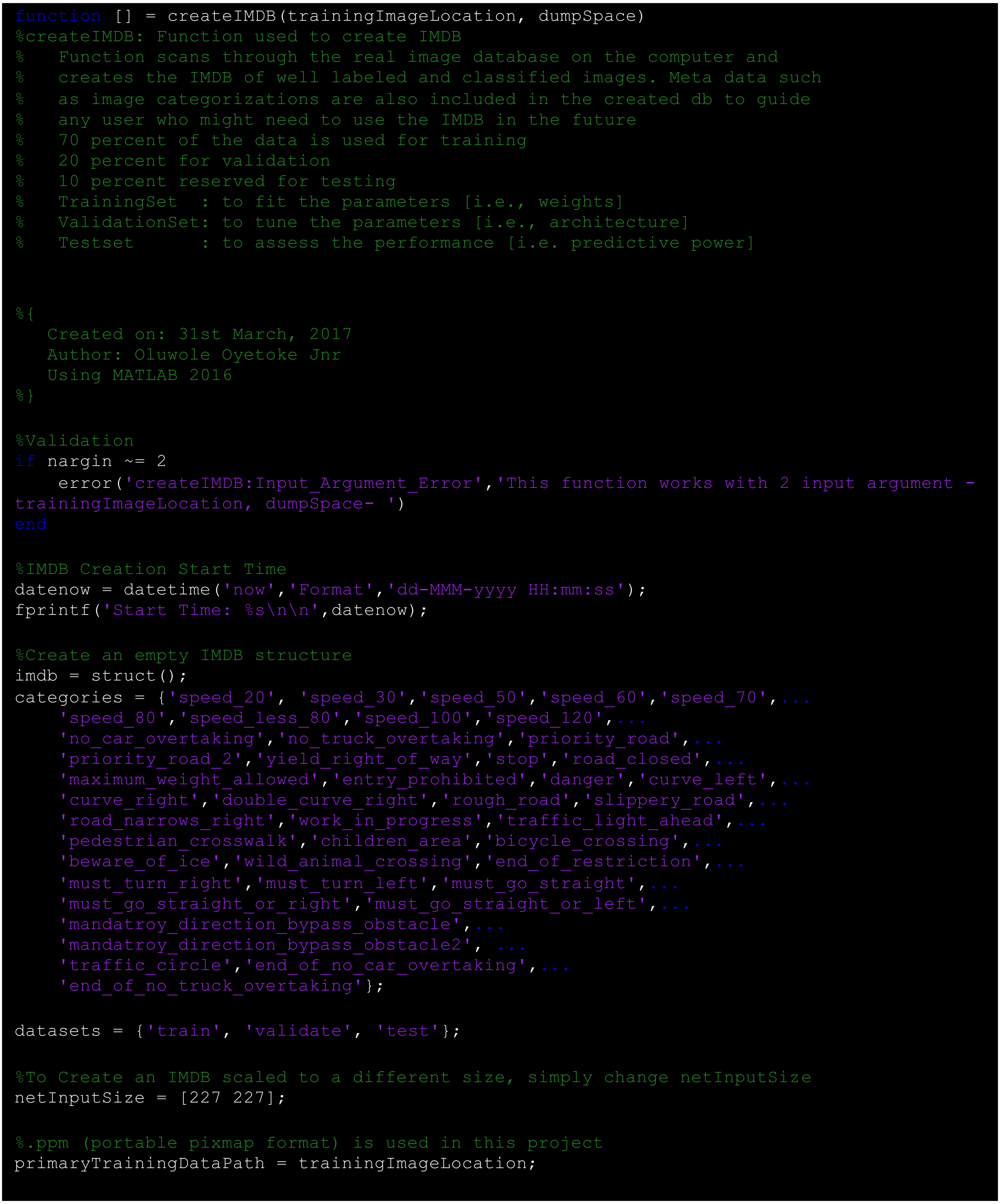
CREATE TRAINING DATA

Training Set Details

- No. of Images: 39000 images containing a series of 43 traffic sign categories
- 70% was used for training
- 30% for validation
- 10% for testing

Code 0. Code Used to Create the IMDB 'Struct' from the GTSRB Dataset



```
function [] = createIMDB(trainingImageLocation, dumpSpace)
%createIMDB: Function used to create IMDB
%   Function scans through the real image database on the computer and
%   creates the IMDB of well labeled and classified images. Meta data such
%   as image categorizations are also included in the created db to guide
%   any user who might need to use the IMDB in the future
%   70 percent of the data is used for training
%   20 percent for validation
%   10 percent reserved for testing
%   TrainingSet   : to fit the parameters [i.e., weights]
%   ValidationSet: to tune the parameters [i.e., architecture]
%   Testset       : to assess the performance [i.e. predictive power]



%{
   Created on: 31st March, 2017
   Author: Oluwole Oyetoke Jnr
   Using MATLAB 2016
%}

%Validation
if nargin ~= 2
    error('createIMDB:Input_Argument_Error','This function works with 2 input argument -
trainingImageLocation, dumpSpace- ')
end

%IMDB Creation Start Time
datenow = datetime('now','Format','dd-MMM-yyyy HH:mm:ss');
fprintf('Start Time: %s\n\n',datenow);

%Create an empty IMDB structure
imdb = struct();
categories = {'speed_20', 'speed_30','speed_50','speed_60','speed_70',...
    'speed_80','speed_less_80','speed_100','speed_120',...
    'no_car_overtaking','no_truck_overtaking','priority_road',...
    'priority_road_2','yield_right_of_way','stop','road_closed',...
    'maximum_weight_allowed','entry_prohibited','danger','curve_left',...
    'curve_right','double_curve_right','rough_road','slippery_road',...
    'road_narrows_right','work_in_progress','traffic_light_ahead',...
    'pedestrian_crosswalk','children_area','bicycle_crossing',...
    'beware_of_ice','wild_animal_crossing','end_of_restriction',...
    'must_turn_right','must_turn_left','must_go_straight',...
    'must_go_straight_or_right','must_go_straight_or_left',...
    'mandatroy_direction_bypass_obstacle',...
    'mandatroy_direction_bypass_obstacle2', ...
    'traffic_circle','end_of_no_car_overtaking',...
    'end_of_no_truck_overtaking'};

datasets = {'train', 'validate', 'test'};

%To Create an IMDB scaled to a different size, simply change netInputSize
netInputSize = [227 227];

%.ppm (portable pixmap format) is used in this project
primaryTrainingDataPath = trainingImageLocation;
```

```
try
    %Loads all the content of the  training folder
    trainingFolderStruct = dir([primaryTrainingDataPath]);
catch
    error('createIMDB2:Traing_Image_Location_Error','Error Encounterd When Loading Image
Data From Folder')
end
[noOfTrainingFolders d] = size(trainingFolderStruct);
dirFlags = [trainingFolderStruct.isdir];
subFolderList = trainingFolderStruct(dirFlags);


%Loop through the training image folder to get total number of images in DB
%Main folder contains subfolders of images for each training class
imageCount=0;
for mainLoopCount = 3:noOfTrainingFolders
    secondaryTrainingDataPath = fullfile(primaryTrainingDataPath,...
        subFolderList(mainLoopCount).name, '*.ppm');
    subFolderStruct = dir([secondaryTrainingDataPath]);
    noOfContents2= numel(subFolderStruct);
    imageCount =imageCount+noOfContents2;
    % fprintf('Number of Images in Training Class %s = %d\n', ...
    %subFolderList(mainLoopCount).name, noOfContents2);
end
fprintf('Number of Training Images in Total: %d\n', imageCount);


%Initialize part of the imdb structure
imdb.meta.sets = {'train', 'validate', 'test'};

%Possible Image Categories
imdb.meta.categories = categories;

%AlexNet Uses 227 by 227 by 3 images
imdb.images.data = ones(netInputSize(1), netInputSize(2), 3, imageCount, 'single');
imdb.images.labels = ones(1,imageCount, 'single'); %Image label
% vector indicating to which set an image belong,
%i.e., % training, validation, test etc.
imdb.images.set = ones(1, imageCount, 'uint8');

fprintf('Each image will be resized to %d by %d by 3 \n', netInputSize(1),
netInputSize(2));

%Loop through Dataset, appropriately dimension all contents, label,
%classify and place in sets
imageCounter=1;
for mainLoopCount = 3:noOfTrainingFolders

    actualPos = mainLoopCount-2;
    toWorkOn = char (categories(actualPos));
    fprintf('%d. Loading and working on training, validation and test images for '' %s ''
traffic sign\n',actualPos, toWorkOn);
    secondaryTrainingDataPath = fullfile(primaryTrainingDataPath,...
        subFolderList(mainLoopCount).name, '*.ppm');

    %Get only .ppm contnets of the folder
    subFolderStruct = dir([secondaryTrainingDataPath]);



    %Get no. of contents in folder
    noOfContents2= numel(subFolderStruct);

    if(noOfContents2<10)
        error('createIMDB2:Image_Class_Error',...
            'Every Class in the dataset should contain at least 10 images')
    end

    %Get Number of images to be used for training, validation and testing
    trainNo = floor(0.7* noOfContents2);
    valNo = floor(0.2* noOfContents2);
    testNo = floor(0.1* noOfContents2);
    total =trainNo+valNo+testNo;
    if(total<noOfContents2)
        difference = noOfContents2 - total;
        trainNo = trainNo+difference;
    elseif (total>noOfContents2)
        difference = total-noOfContents2;
        trainNo = trainNo-difference;
    end
    fprintf('Out of a total of %d images in this class %d will be used for training, %d
for validation and %d for testing\n', total, trainNo, valNo, testNo);
    setBank = getSetPositions(trainNo, valNo, testNo);

    for innerLoopCount = 1:noOfContents2

        pathToImage= fullfile(primaryTrainingDataPath, subFolderList(mainLoopCount).name,
subFolderStruct(innerLoopCount).name);

        imageRead = imread(pathToImage);

        %Check to make sure image contains 3 channels. AlexNet works with 3
        %channels
        [xDim yDim zDim] = size(imageRead);
        threeDImage = imageRead;
        if (zDim==1)
            threeDImage = cat(3, imageRead, imageRead, imageRead);
        end



        %Resize Image to acceptable AlexNet Input size [227 227 3]
        properSizedImageData = threeDImage;
        if(xDim~=netInputSize(1) || yDim~=netInputSize(2))
            properSizedImageData = imresize(threeDImage, [netInputSize(1)
netInputSize(2)], 'bilinear');
        end

        %Set image back into DB & apply all related meta information
        %AlexNet Uses 227 by 227 by 3 images
        %Load in Image Data. Stack of 3 channels
        imdb.images.data(:,:,1,imageCounter) = properSizedImageData(:,:,1);
        imdb.images.data(:,:,2,imageCounter) = properSizedImageData(:,:,2);
        imdb.images.data(:,:,3,imageCounter) = properSizedImageData(:,:,3);

        %Assign Label to image
        %categories(mainLoopCount-2);
        imdb.images.labels(1,imageCounter) = mainLoopCount-2;

        %datasets(1) = Training, datasets(2) = Validate datasets(3)= Test
        imdb.images.set(1, imageCounter) = setBank(innerLoopCount);

        imageCounter = imageCounter+1;

    end %Inner Loop
    percentageCompleted = uint8 ((imageCounter*100)/imageCount);
    fprintf('%d%% completed so far\n\n', percentageCompleted);
end %Main Loop



%Save IMDB
```

```
fprintf('Saving IMDB file\n');
filename = fullfile(dumpSpace, 'Traffic_Sign_IMDB(GSTBR)_All_32by32.mat');
save(filename, 'imdb');

datenow2 = datetime('now','Format','dd-MMM-yyyy HH:mm:ss');
fprintf('End Time: %s\n\n',datenow2);


d1=datenum(datenow);     % convert to number
d2=datenum(datenow2);    % convert to number
difference=d2-d1;        % difference between the two
days = floor(difference);
hrs = datestr(difference, 'HH');
mins = datestr(difference, 'MM');
seconds = datestr(difference, 'SS');
% difference in days:hr:min:sec

%Escape random number generator legacy mode
rng('default');

fprintf('Overall Time Taken: %d day(s), %s hour(s), %s minute(s), %s second(s)
\n\n',days, hrs, mins, seconds);

end

%Function used to pick random images for test, validate and train
function allocationSpots = getSetPositions(trainNo, valNo, testNo);
total = trainNo+valNo+testNo;
perms = randperm(total);

trainPositions = perms(1:trainNo);
valPositions = perms(trainNo+1:trainNo+valNo);
testPositions = perms(trainNo+valNo+1:trainNo+valNo+testNo);

allocationSpots = zeros(total,1);

for i=1:total
    if(ismember(i,trainPositions))
        allocationSpots(i) = 1;
    elseif(ismember(i,valPositions))
        allocationSpots(i) = 2;
    elseif(ismember(i,testPositions))
        allocationSpots(i) = 3;
    end
end



end
```

TRAIN NEURAL NETWORK

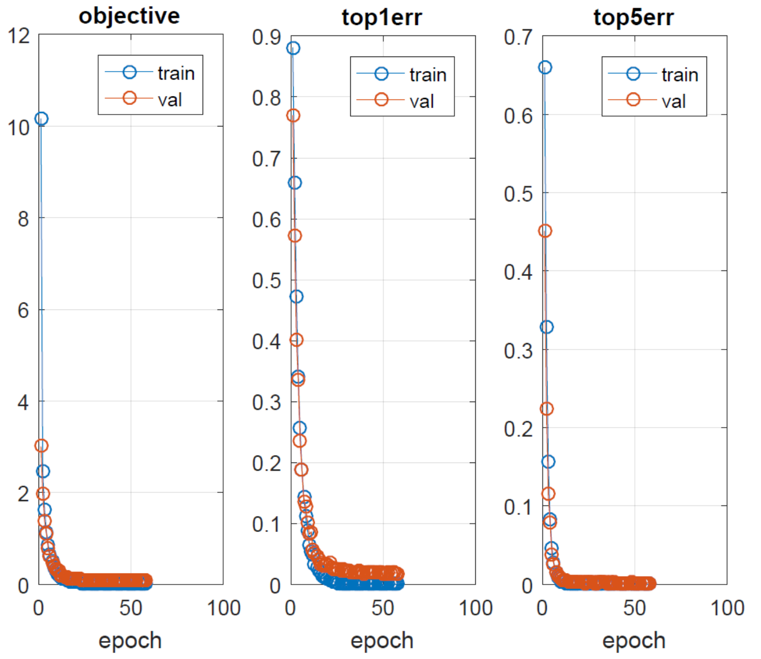


Figure 0. Figure Showing Decreasing Error Rate as training happened (47 hours for 58 rounds [Epoch])

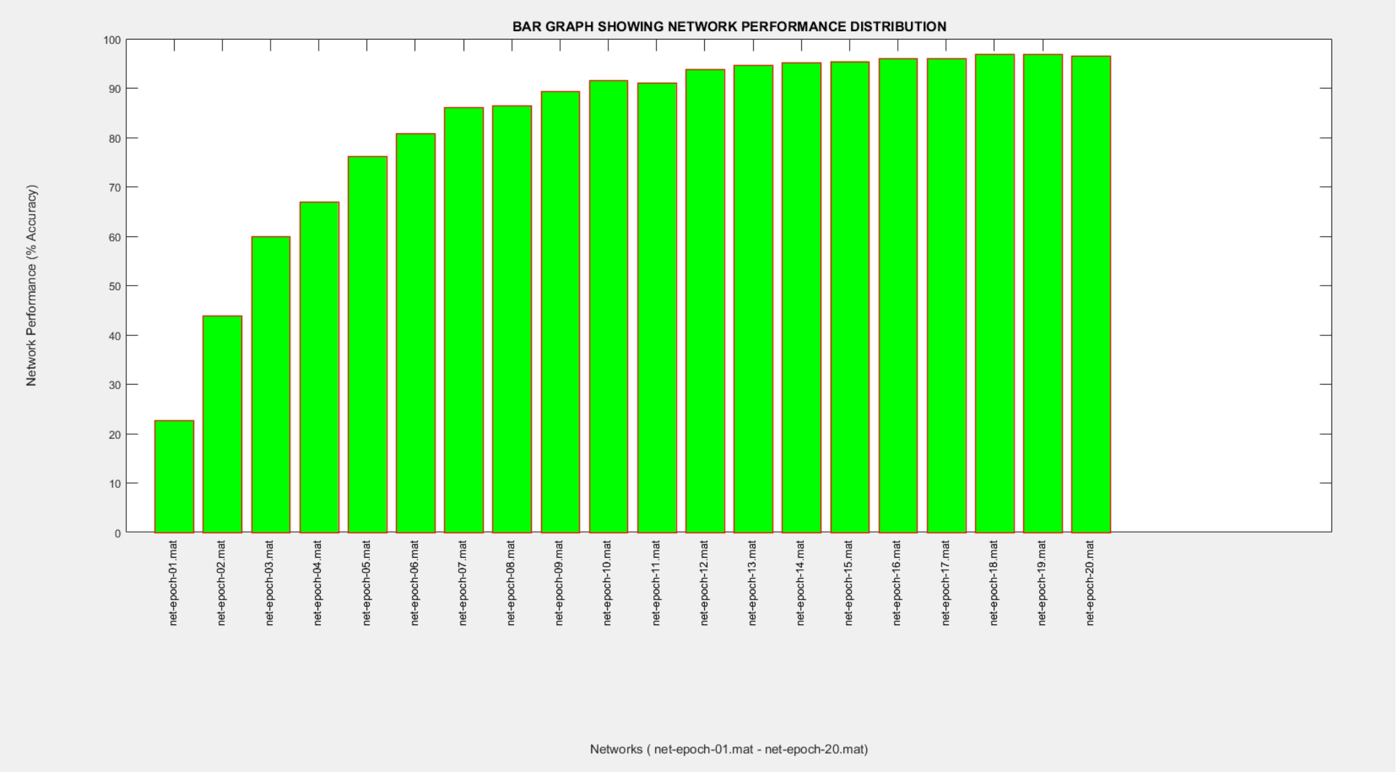


Figure 1. Figure Showing Improvement in Classification Accuracy of System Between 1st and 20th Training Epoch

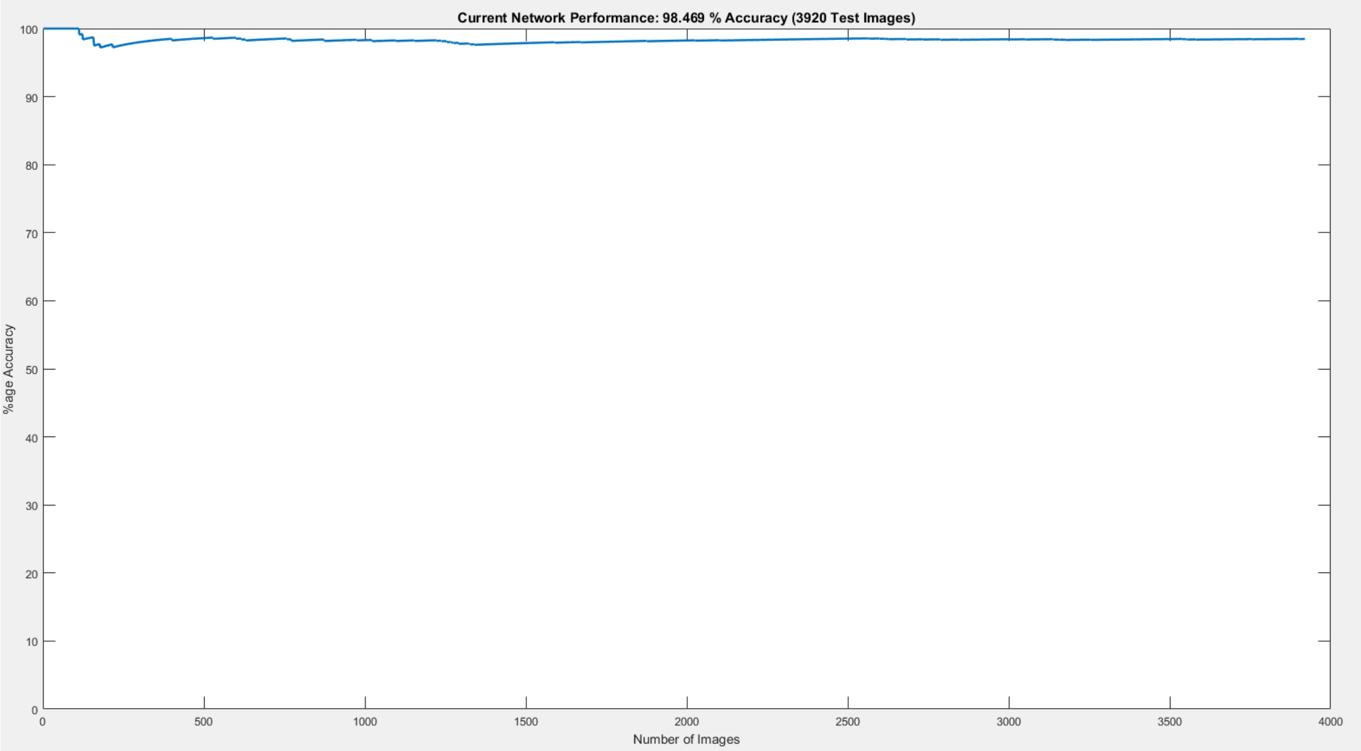


Figure 2. Figure Showing Network Performance After 58 Rounds of Training

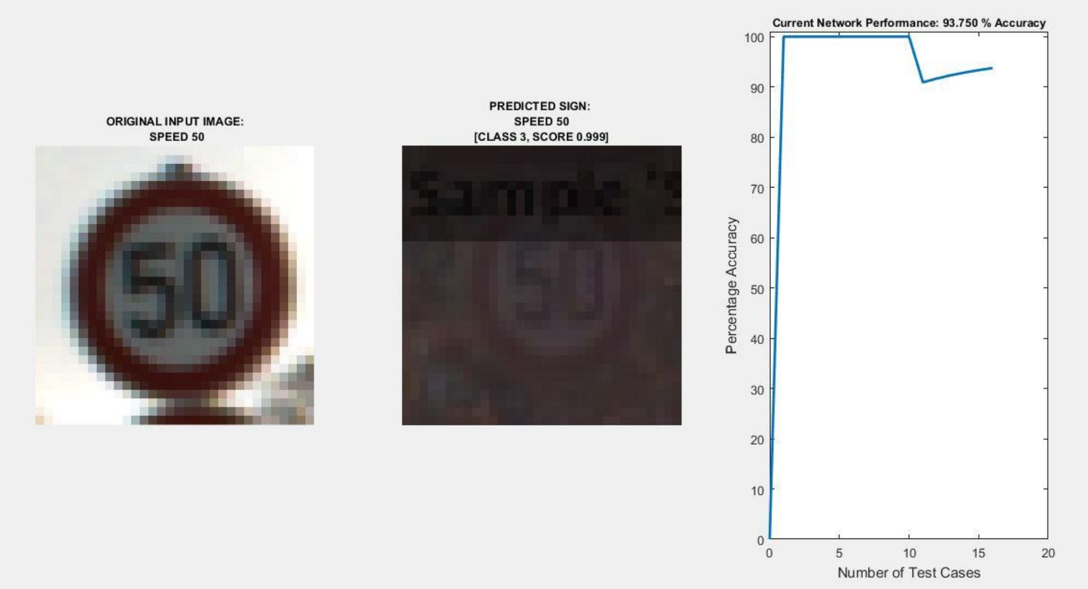


Figure 3. Figure Showing Random Test Ongoing

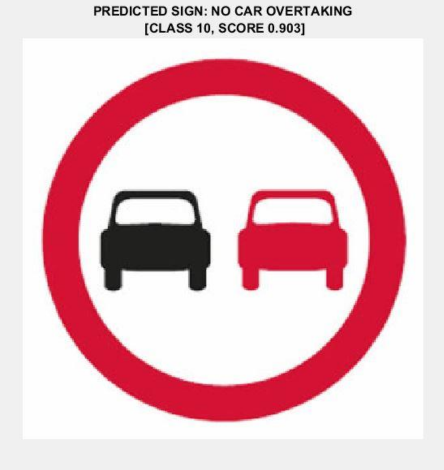
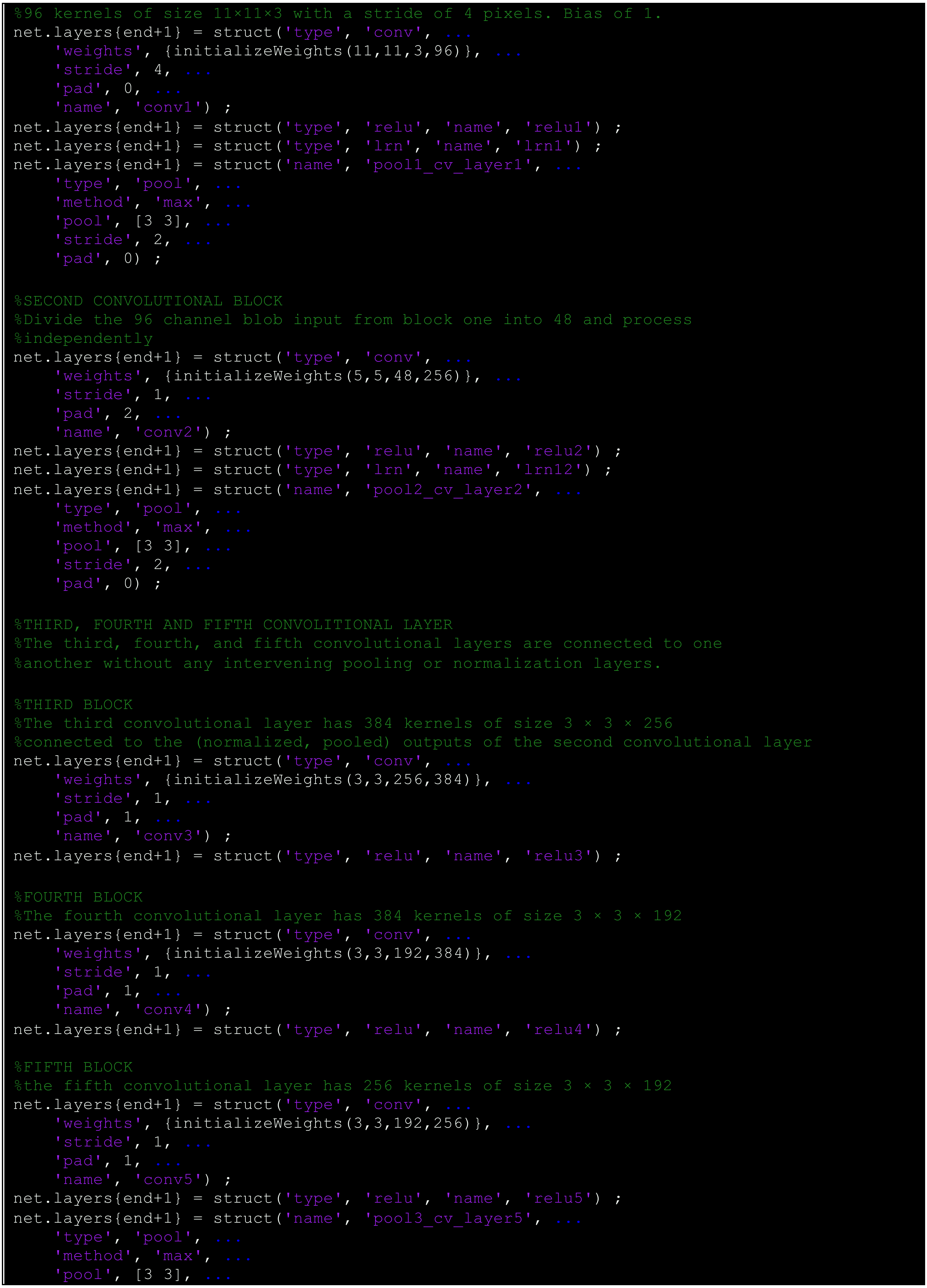


Figure 4. Figure Showing a Classification Test

Code 1. Code Used to Create and Train the AlexNet Model (SimpleNN)

```
function [net  trainingInfo] = AlexNetNN( imageDB, dumpLocation );
%AlexNetNN: Used to Create AlexNet and Train it based on specified %imageDB
%   Creates A SimpleNN AlexNet Network, initializes its weight values and
%   trians it based on the suplied ImageMDB over several Epochs
%{
            Created on: 31st March, 2017
             Author: Oluwole Oyetoke Jnr
             Using MATLAB 2016

               AlexNet NETWORK OVERVIEW
AlexNet Structure: 60 million Parameters
8 layers in total: 5 Convolutional and 3 Fully Connected Layers
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 43 neurons (class scores)

•   Every fully-connected Layer has 4096 neurons
•   Max-pooling layers follow first, second, and fifth convolutional layers
•   Response-normalization layers follow the first and second convolutional layers
•   The ReLU non-linearity is applied to the output of every convolutional and fully-
connected layer
%}

% Install and compile MatConvNet Library (needed only once).
untar('http://www.vlfeat.org/matconvnet/download/matconvnet-1.0-beta23.tar.gz') ;
cd matconvnet-1.0-beta23
run matlab/vl_compilenn;

% Setup MatConvnet.
run matlab/vl_setupnn;

%Choose SimpleNN network type. There also exists the Directed Acyclic Graph
%(DagNN). An object-oriented wrapper for CNN with complex topologies
opts.networkType = 'simplenn' ;

%Switch away from MATLab lagacy random number generator method (caution)
rng('default');
rng(0);

%create empty layer struct where the AlexNet network will be modeled into
net.layers = {} ;
networkInputSize= [227 227 3];

%Validation
if (nargin ~= 2)
 error('AlexNetNN:Input_Argument_Error','This function works with 2 input arguments -
imageDB, dumpLocation- ');
end

%BEGINING OF NETWORK DEFINITION

%FIRST CONVOLUTIONAL BLOCK
%The first convolutional layer filters the 227×227×3 input image with
```



```
%96 kernels of size 11×11×3 with a stride of 4 pixels. Bias of 1.
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(11,11,3,96)}, ...
    'stride', 4, ...
    'pad', 0, ...
    'name', 'conv1') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu1') ;
net.layers{end+1} = struct('type', 'lrn', 'name', 'lrn1') ;
net.layers{end+1} = struct('name', 'pool1_cv_layer1', ...
    'type', 'pool', ...
    'method', 'max', ...
    'pool', [3 3], ...
    'stride', 2, ...
    'pad', 0) ;

%SECOND CONVOLUTIONAL BLOCK
%Divide the 96 channel blob input from block one into 48 and process
%independently
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(5,5,48,256)}, ...
    'stride', 1, ...
    'pad', 2, ...
    'name', 'conv2') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu2') ;
net.layers{end+1} = struct('type', 'lrn', 'name', 'lrn12') ;
net.layers{end+1} = struct('name', 'pool2_cv_layer2', ...
    'type', 'pool', ...
    'method', 'max', ...
    'pool', [3 3], ...
    'stride', 2, ...
    'pad', 0) ;

%THIRD, FOURTH AND FIFTH CONVOLITIONAL LAYER
%The third, fourth, and fifth convolutional layers are connected to one
%another without any intervening pooling or normalization layers.

%THIRD BLOCK
%The third convolutional layer has 384 kernels of size 3 × 3 × 256
%connected to the (normalized, pooled) outputs of the second convolutional layer
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,256,384)}, ...
    'stride', 1, ...
    'pad', 1, ...
    'name', 'conv3') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu3') ;

%FOURTH BLOCK
%The fourth convolutional layer has 384 kernels of size 3 × 3 × 192
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,192,384)}, ...
    'stride', 1, ...
    'pad', 1, ...
    'name', 'conv4') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu4') ;

%FIFTH BLOCK
%the fifth convolutional layer has 256 kernels of size 3 × 3 × 192
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(3,3,192,256)}, ...
    'stride', 1, ...
    'pad', 1, ...
    'name', 'conv5') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu5') ;
net.layers{end+1} = struct('name', 'pool3_cv_layer5', ...
    'type', 'pool', ...
    'method', 'max', ...
    'pool', [3 3], ...
    'stride', 2, ...
    'pad', 0) ;


%FULLY CONNECTED LAYER 1
%The fully-connected layers have 4096 neurons each
% with a network input of 227 by 227 by 3, and zero padding at layer 1,
%the input to this layer should be changed to [6,6,256,4096]since the
%output from above layer will be [6x6x256]
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(1,1,256,4096)}, ...
    'stride', 1, ...
    'pad', 0, ...
    'name', 'fc1') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu6') ;

%FULLY CONNECTED LAYER 2
% since the output from above is [1x1x4096]
net.layers{end+1} = struct('type', 'conv', ...
    'weights', {initializeWeights(1,1,4096,4096)}, ...
    'stride', 1, ...
    'pad', 0, ...
    'name', 'fc2') ;
net.layers{end+1} = struct('type', 'relu', 'name', 'relu7') ;

%FULLY CONNECTED LAYER 3
% since the output from above is [1x1x4096] & We need 43 output 1by1
% classfier neurons
net.layers{end+1} = struct('type', 'conv', ...
    'weights',{initializeWeights(1,1,4096,43)}, ...
    'stride', 1, ...
    'pad', 0, ...
    'name', 'fc3') ;

%Classifier layer to softmax function
net.layers{end+1} = struct('type', 'softmaxloss') ;

% Add meta parameters to the network structure
net.meta.inputSize = [networkInputSize(1) networkInputSize(2) networkInputSize(3)] ;
net.meta.trainOpts.learningRate =  0.00005;
net.meta.trainOpts.weightDecay = 0.00005 ;
net.meta.trainOpts.batchSize = 100 ;
net.meta.trainOpts.numEpochs = 80;

net.meta.categories = imageDB.meta.categories;
net.meta.sets = imageDB.meta.sets;

%END OF NETWORK DEFINITION
% Fill in default values & Initialze the LRN with the following parameters
%PARAM = [N KAPPA ALPHA BETA] = [5 2 10E-4 0.75] OR [5 1 0.0001/5 0.75] ;
net = vl_simplenn_tidy(net) ;



%Initialize Parameter Needed For Training
opts.train.batchSize = 100; %Select Image Batch Size of 10
opts.train.numEpochs = 80; % 80 passes through the network during training
opts.train.continue = true ; %Continue from last Epoch if interrupted
opts.train.gpus = [] ; %Do not use GPU. Change to '[1]' to use
%Small LR, network might take time to converge but will be more accurate
opts.train.learningRate = 0.00005;
%Combined with the local weight decay in the current training layer
opts.train.weightDecay = 0.0005;
opts.train.momentum = 0.9;
opts.train.expDir = dumpLocation; %Where training output will be dumped
opts.train.numSubBatches = 1;
```

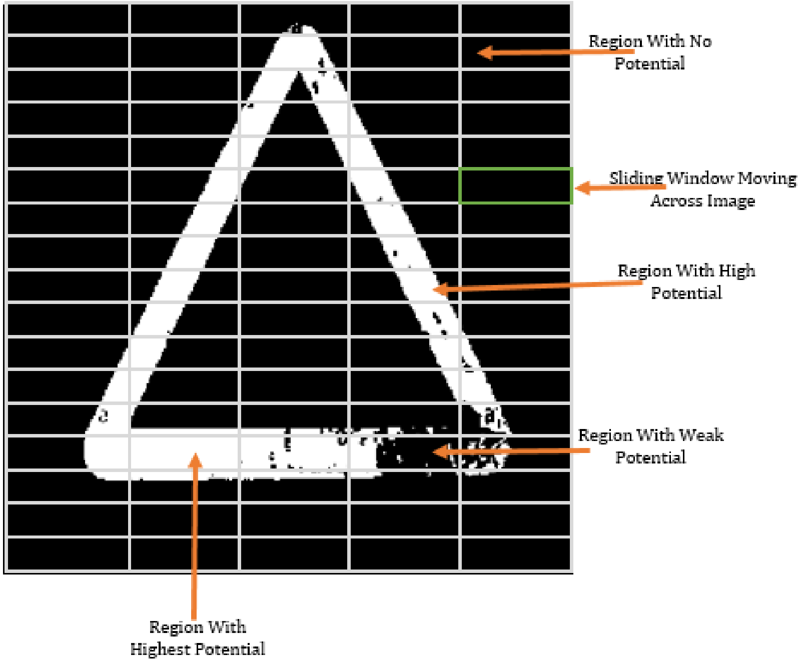
```
% getBatch options
bopts.useGpu = numel(opts.train.gpus) >  0 ;


%Before training starts take the average image out from all images in DB
%to improve speed of convergence of network and performance
imageMean = mean(imageDB.images.data(:)) ;
imageDB.images.data = imageDB.images.data - imageMean ;
net.meta.ImageMean = imageMean;


% Switch to the Requied Network type and begin Training
switch lower(opts.networkType)
    case 'simplenn'
        % Model already created above, therefore, begin Training (%set  are 1
        %the images to beused for training & set =2 are for testing)
        trainingInfo = cnn_train(net, imageDB,  @(i,b)
getBatch(bopts,i,b,networkInputSize), ...
            opts.train, 'val', find(imageDB.images.set == 2)) ;

    case 'dagnn'
        net = dagnn.DagNN.fromSimpleNN(net, 'canonicalNames', true) ;
        net.addLayer('error', dagnn.Loss('loss', 'classerror'), ...
            {'prediction','label'}, 'error') ;
        %Add training code here if making use of DagNN
    otherwise
        assert(false) ;
end
end

%Function used to initialize filter wieghts and bias for the network
function weights = initializeWeights(width,height,channels, depth)
%Using He's Initialization Method
var = sqrt(2/(width*height*channels));

%Filter Weights
weights{1} = randn(width,height,channels,depth,'single') * var;

%Biases Weights
weights{2} = zeros(depth,1,'single') ;
end

% Create batch of images and labels
function [im, lb] = getBatch(opts,imageDB, batch, networkInputSize)
images = imageDB.images.data(:,:,:,batch);
labels = imageDB.images.labels(1,batch);

%Check to make sure image contains 3 channels
[xDim yDim zDim] = size(images);
threeDImage = images;
if (zDim==1)
    threeDImage = cat(3, images, images, images);
end

%Resize Image to acceptable AlexNet Input size [227 227 3] || [32 32 3]
properSizedImageData = threeDImage;

if((xDim~=networkInputSize(1)) || (yDim~=networkInputSize(2)))
    %Bilinear interpolation; the output pixel value is a weighted average
    %of pixels in the nearest 2-by-2 neighborhood
    properSizedImageData =  imresize(threeDImage, ...
        [networkInputSize(1) networkInputSize(2)], 'bilinear');
end

if opts.useGpu > 0
```

```
    images = gpuArray(properSizedImageData) ;
end
im = properSizedImageData;
lb = labels;
end
```

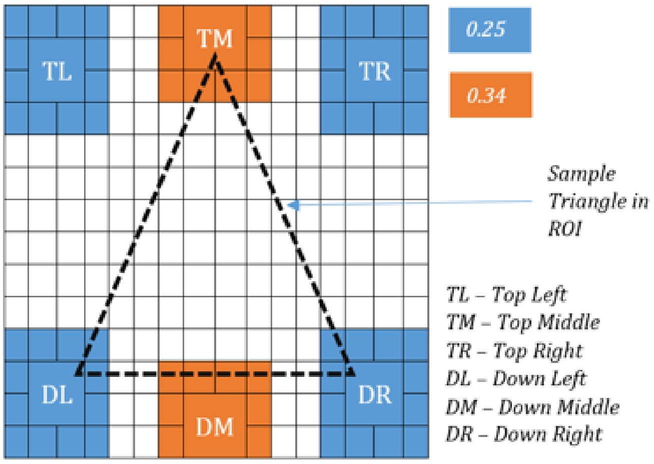
Other Codes

- callAlex.m
- classifyImg.m
- createIMDB.m
- sceneClassifier.m

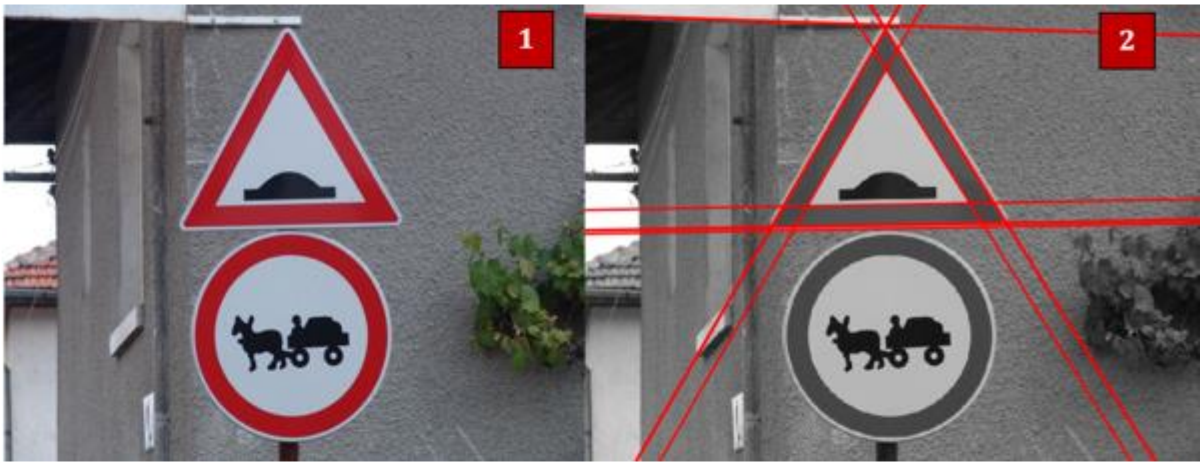
DETECTION MECHANISM


Region With No Potential
Sliding Window Moving Across Image
Region With High Potential
Region With Weak Potential
Region With Highest Potential

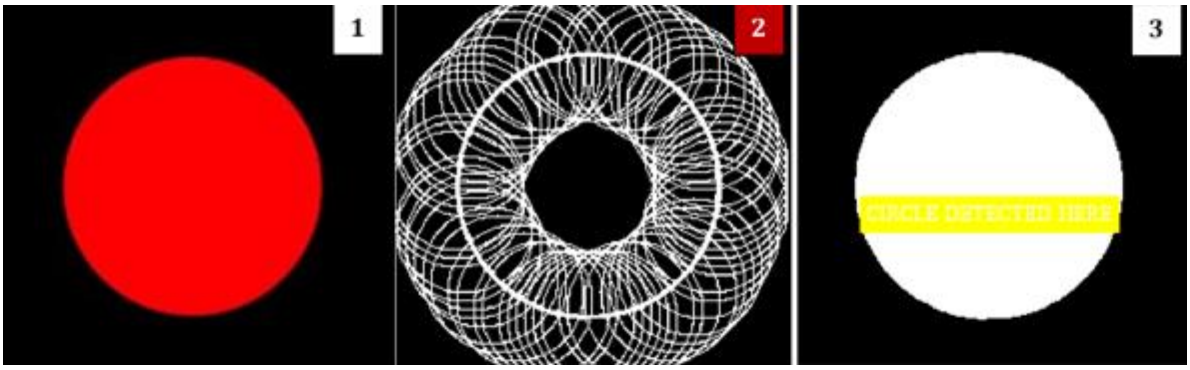
CONNECTED COMPONENT ANALYSIS



SUB-REGION ANALYSIS



LINE DETECTION USING HOUGH TRANSFORM



CIRCLE DETECTION USING CIRCULAR HOUGH TRANSFORM

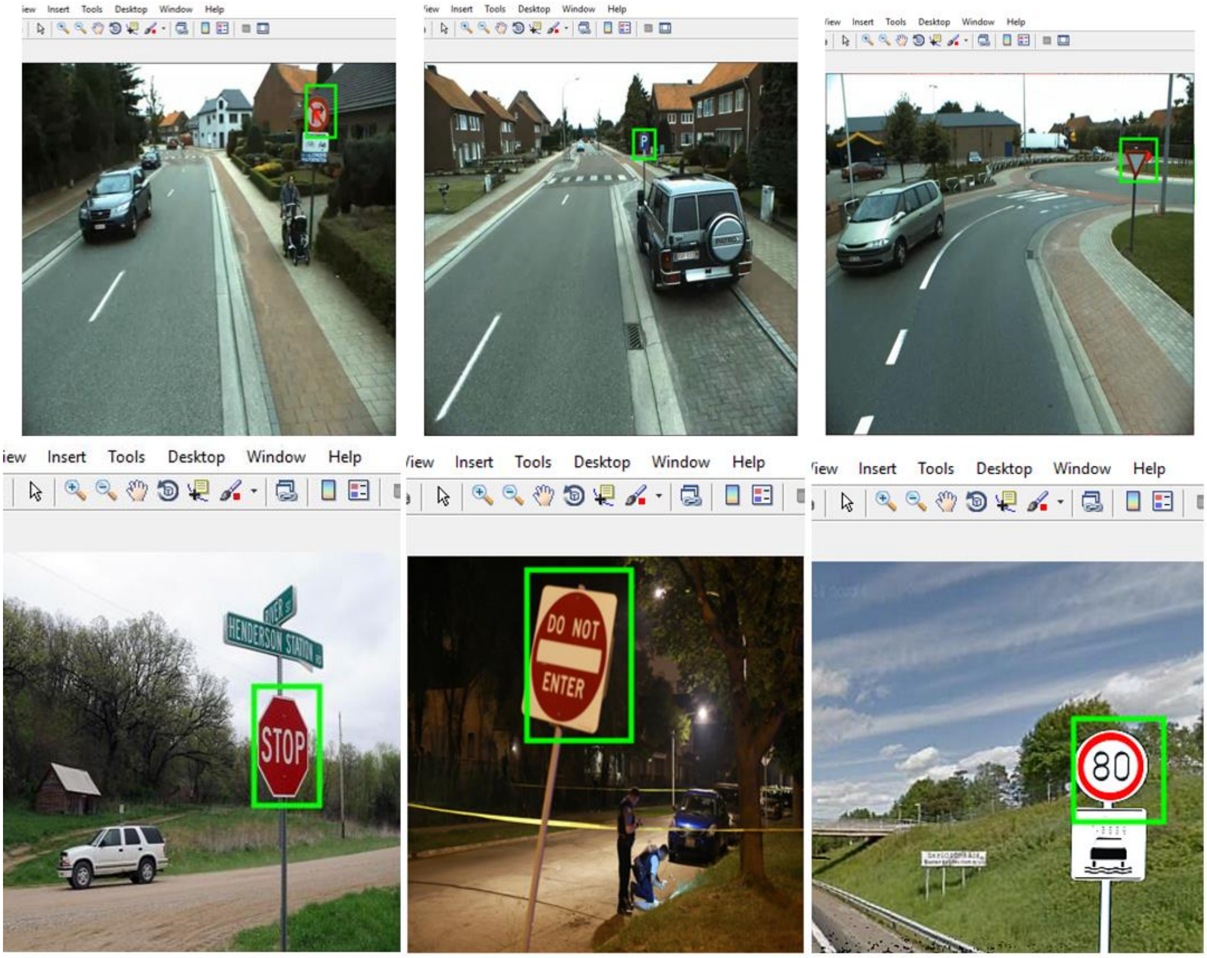

iew Insert Tools Desktop Window Help
iew Insert Tools Desktop Window Help
iew Insert Tools Desktop Window Help
iew Insert Tools Desktop Window Help
iew Insert Tools Desktop Window Help
iew Insert Tools Desktop Window Help
HENDERSON STATION
STOP
DO NOT ENTER
80

Figure 5. Figure Showing Detection in Action

Code 2. Connected Component Analysis (CCA) C/C++ Code

```
#include "mex.h"
#include <stdlib.h>
#include <cstdlib>
#include <Math.h>
#include <stdio.h>

#define xDim 500
#define yDim 500
#define zDim 3
#define windowSize 20
#define stepSize 10
#define beacon  ceil((windowSize*windowSize) / 20)

//using namespace std;

struct Region {

    int region; //region unattended to
    int value; //default value
    int xCoord; //default value
    int yCoord; //default value

    struct Region* left; //empty MapRegion object
    struct Region* right; //empty MapRegion object
    struct Region* top; //empty MapRegion object
    struct Region* bottom; //empty MapRegion object


public:
    //initialize variables
    Region() : region (-1), value(-1), xCoord(-1), yCoord(-1) {}
};



class MapRegion {

public:

    MapRegion() {};
    ~MapRegion() {};
    //int CCA(int **regionShedd);
    //int getLocation(int x, int y, int width);
};

void CCAmex(int* inputArray, int* outputArray);
int getLocation(int x, int y, int width);



void CCAmex(int* inputArray, int* outputArray) {

    //Initialize aprameters and variables
    int startX = 0; int startY = 0; int stopX = windowSize - 1; int stopY = windowSize -
1;
    int xCounter = 0;
    int yCounter = 0;
    int regionSum = 0; //Sum of the number of Pixels which are potentialy members of the
ROI

    const int xBlocks = ((xDim - windowSize) / stepSize) + 1;
    const int yBlocks = ((yDim - windowSize) / stepSize) + 1;
    const int totalBlocks = xBlocks * yBlocks;
    struct Region* Map[totalBlocks];
    int **regionShedd = new int*[xDim];

    int* regionSizeTracker = (int*)calloc(totalBlocks, sizeof(int));
    int* regionMap = (int*)calloc(totalBlocks, sizeof(int));
    int regionCounter = 0;
    int location = 0; int left = 0; int right = 0; int top = 0; int bottom = 0;

    int* myLeftNeighRegion = NULL;
    int* myRightNeighRegion = NULL;
    int* myTopNeighRegion = NULL;
    int* myBottomNeighRegion = NULL;
    int cnt=0;

    int xCount = 0; int yCount = 0;
    int* oldestRegion = NULL;

    for (int m=0; m<totalBlocks; m++) {
        Map[m] = new Region(); //Initialize stuct
    }

         //Get RegionShed in 2D
        for (int i = 0; i < xDim; i++) {
        regionShedd[i] = new int[xDim];
        for (int j = 0; j < xDim; j++) {
            regionShedd[i][j] = inputArray[cnt];
            cnt++;
            }
        }



    //Analysis loop
    for ( int BlockLoop=0; BlockLoop<totalBlocks; BlockLoop++ ) {
        regionSum = 0;
        xCount = xCounter;
        yCount = yCounter;
        location = getLocation(xCount, yCount, xBlocks);
        left = getLocation(xCount - 1, yCount, xBlocks);
        right = getLocation(xCount + 1, yCount, xBlocks);
        bottom = getLocation(xCount, yCount + 1, xBlocks);
        top = getLocation(xCount, yCount - 1, xBlocks);

        //Pixel Sum Pooling happens here
        for (int j = startX; j<=stopX; j++) {
            for (int k = startY; k<=stopY; k++) {
                regionSum = regionSum + regionShedd[j][k];
            }
        }

        //Save region's potential & Link Regions Together
        regionMap[BlockLoop] = regionSum;


        Map[location]->value = regionSum;
        Map[location]->xCoord = xCount;
        Map[location]->yCoord = yCount;

        if (xCount == 0 && yCount == 0) {
            Map[location]->left = Map[location];
            Map[location]->right = Map[right];
            Map[location]->top = Map[location];
            Map[location]->bottom = Map[bottom];
        }
        else if (xCount == xBlocks - 1 && yCount == 0) {
            Map[location]->left = Map[left];
            Map[location]->right = Map[location];
            Map[location]->top = Map[location];
            Map[location]->bottom = Map[bottom];
        }
        else if (xCount == 0 && yCount == yBlocks - 1) {
            Map[location]->left = Map[location];
            Map[location]->right = Map[right];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[location];
        }
        else if (xCount == xBlocks - 1 && yCount == yBlocks - 1) {
            Map[location]->left = Map[left];
            Map[location]->right = Map[location];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[location];
        }
        else if (xCount == 0) {
            Map[location]->left = Map[location];
            Map[location]->right = Map[right];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[bottom];
        }
        else if (xCount == xBlocks - 1) {
            Map[location]->left = Map[left];
            Map[location]->right = Map[location];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[bottom];
        }
        else if (yCount == 0) {
            Map[location]->left = Map[left];
            Map[location]->right = Map[right];
            Map[location]->top = Map[location];
            Map[location]->bottom = Map[bottom];
        }
        else if (yCount == yBlocks - 1) {
            Map[location]->left = Map[left];
            Map[location]->right = Map[right];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[location];
        }
        else {
            Map[location]->left = Map[left];
            Map[location]->right = Map[right];
            Map[location]->top = Map[top];
            Map[location]->bottom = Map[bottom];
        }

        //Check to see if current region is bright enough
        if (Map[location]->value>beacon) { //i.e more than 1/4th of the region is white
            //Check the regions its neighbours belong to
            myLeftNeighRegion = &Map[location]->left->region;
            myRightNeighRegion = &Map[location]->right->region;
            myTopNeighRegion = &Map[location]->top->region;
            myBottomNeighRegion = &Map[location]->bottom->region;

            //If any of its neighbours already belong to a region, select the region of
the neigbour with the lowest value and force this one to join it
            if (*myLeftNeighRegion>-1 || *myRightNeighRegion>-1 || *myTopNeighRegion>-1 ||
*myBottomNeighRegion>-1) {
                //oldestRegion = NULL;
                if (*myLeftNeighRegion != -1) { *oldestRegion = *myLeftNeighRegion;}
                else if (*myRightNeighRegion != -1) { *oldestRegion = *myRightNeighRegion;
}
                else if (*myTopNeighRegion != -1) { oldestRegion = myTopNeighRegion; }
                else if (*myBottomNeighRegion != -1) { *oldestRegion =
*myBottomNeighRegion; }

                if (*myLeftNeighRegion != -1 && *myLeftNeighRegion<*oldestRegion) {
*oldestRegion = *myLeftNeighRegion; }
                if (*myRightNeighRegion != -1 && *myRightNeighRegion<*oldestRegion) {
*oldestRegion = *myRightNeighRegion; }
                if (*myTopNeighRegion != -1 &&  *myTopNeighRegion<*oldestRegion) {
*oldestRegion = *myTopNeighRegion; }
                if (*myBottomNeighRegion != -1 && *myBottomNeighRegion<*oldestRegion) {
*oldestRegion = *myBottomNeighRegion; }

                Map[location]->region = *oldestRegion;

                regionSizeTracker[*oldestRegion] = regionSizeTracker[*oldestRegion] + 1;

                //Force other neigbours of this region that may be already part of another
region to join the lower region
                if (*myLeftNeighRegion >-1 && *myLeftNeighRegion != *oldestRegion) {
                    regionSizeTracker[*myLeftNeighRegion] =
regionSizeTracker[*myLeftNeighRegion] - 1; //Update region Size tracker
                    Map[location]->left->region = *oldestRegion;
                    regionSizeTracker[*oldestRegion] = regionSizeTracker[*oldestRegion] +
1; //Update region Size tracker
                }
                if (*myRightNeighRegion >-1 && *myRightNeighRegion != *oldestRegion) {
                    regionSizeTracker[*myRightNeighRegion] =
regionSizeTracker[*myRightNeighRegion] - 1; //Update region Size tracker
                    Map[location]->right->region = *oldestRegion;
                    regionSizeTracker[*oldestRegion] = regionSizeTracker[*oldestRegion] +
1;  //Update region Size tracker
                }
                if (*myTopNeighRegion >-1 && *myTopNeighRegion != *oldestRegion) {
                    regionSizeTracker[*myTopNeighRegion] =
regionSizeTracker[*myTopNeighRegion] - 1; //Update region Size tracker
                    Map[location]->top->region = *oldestRegion;
                    regionSizeTracker[*oldestRegion] = regionSizeTracker[*oldestRegion] +
1;  //Update region Size tracker
                }
                if (*myBottomNeighRegion >-1 && *myBottomNeighRegion != *oldestRegion) {
                    regionSizeTracker[*myBottomNeighRegion] =
regionSizeTracker[*myBottomNeighRegion] - 1; //Update region Size tracker
                    Map[location]->bottom->region = *oldestRegion;
                    regionSizeTracker[*oldestRegion] = regionSizeTracker[*oldestRegion] +
1;  //Update region Size tracker
                }

            }
            //If non of its neigbour already belongs to a region, then assign it a region
            else {
                Map[location]->region = regionCounter;
                regionSizeTracker[regionCounter] = regionSizeTracker[regionCounter] + 1;
                regionCounter = regionCounter + 1;
            }//End of 'if any of its neighbours already....'

        }

        else {
            //Do nothing
        }//End of 'check to see if current region is bright enough.....'


        //Loop management
        if (yCounter == yBlocks - 1) {
            //reset xStart, xStop positions and xCounter
            startY = 0;
            stopY = windowSize - 1;
            yCounter = -1;
            xCounter = xCounter + 1;

            startX = startX + stepSize;
            stopX = startX + windowSize -1   ;
        }
        else {

            startY = startY + stepSize;
            stopY = startY + windowSize - 1;
        }

        yCounter = yCounter + 1;
    } //end of xBlockLoop

//memcpy(outputArray, regionSizeTracker, sizeof(int)*totalBlocks); //Copy region tracker
to output

//Get the region with the highest potential
int max=regionSizeTracker[0];
int maxPos=0;
int temp=0;
for (int ii=1; ii<totalBlocks; ii++){
    temp  = regionSizeTracker[ii];
    if(temp>=max){
     max = temp;
     maxPos=ii;
    }
}


 //Get positions of edge indexs
 int currentHighestX=-1; int currentLowestX=xDim+1; int currentHighestY=-1; int
currentLowestY=yDim+1; int thisX=0; int thisY=0;
 int xCnt=0; int yCnt=0;
  for (int iii=0; iii<totalBlocks; iii++){
```

```
 location = getLocation(xCnt, yCnt, xBlocks);
    if (Map[location]->region==maxPos){
       thisX = Map[location]->xCoord;
       thisY = Map[location]->yCoord;

       if (thisX>currentHighestX){currentHighestX = thisX;}
       else if (thisX<currentLowestX){currentLowestX = thisX;}
       else{}

       if (thisY>currentHighestY){currentHighestY = thisY;}
       else if (thisY<currentLowestY){currentLowestY = thisY;}
       else{}
    }
    if (yCnt==yBlocks-1){
       //reset xStart, xStop positions and xCounter
        yCnt=-1;
        xCnt = xCnt+1;
    }
          yCnt=yCnt+1;
}

  //Due to matlab array indexing, add 1 to all indexes
   currentLowestX = currentLowestX + 1;
   currentLowestY = currentLowestY + 1;
   currentHighestX = currentHighestX + 1;
   currentHighestY = currentHighestY + 1;
   int ex = (currentLowestX*stepSize) - (stepSize - 1);
   int wy = (currentLowestY*stepSize) - (stepSize - 1);
   int height = (currentHighestX*stepSize) - (stepSize - 1) + (windowSize - 1) - ex;
   int width = (currentHighestY*stepSize) - (stepSize - 1) + (windowSize - 1) - wy;


   //x,y,width,height
   //int* outputArray = (int*)calloc(4, sizeof(int));
   outputArray[0] = ex;
   outputArray[1] = wy;
   outputArray[2] = height;
   outputArray[3] = width;

   //Free created pointers
  /*
   //FOR--->>> struct Region* Map[totalBlocks];
   for( int indx = 0; indx < totalBlocks; ++indx )
   {
   delete Map[indx];
   }
   delete [] Map;
   //Only free() a pointer that
   //1. is NULL or
   //2. you obtained via a call to malloc(), calloc() or realloc()
   */
  //FOR --->>> int **regionShedd = new int*[500];
   for( int indx = 0; indx < xDim; ++indx )
   {
   free(regionShedd[indx]);
   }
   delete [] regionShedd;

   //FOR --->> int* regionSizeTracker = (int*)calloc(totalBlocks, sizeof(int));
   free(regionSizeTracker);

   //FOR --->>> int* regionMap = (int*)calloc(totalBlocks, sizeof(int));
   free(regionMap);

   //FOR OTHERS
   free(myLeftNeighRegion);
   free(myRightNeighRegion);
```

```
    free(myTopNeighRegion);
    free(myBottomNeighRegion);
    free(oldestRegion);

return;
} //End of function

int getLocation(int x, int y, int width) {
    return  (y + (width*x));
}



/* The gateway function */
/*
 * nlhs  - Number of output (left-side) arguments, or the size of the plhs array.
 * plhs  - Array of output arguments.
 * nrhs  - Number of input (right-side) arguments, or the size of the prhs array.
 * prhs  - Array of input arguments.

 *CALL MEX FUNCTION THIS WAY IN MATLAB CODE
transpose = img';
img1D = transpose(:)'; //TURNING IMAGE TO 1D ARRAY
[output] = CCAmex(uint32(img1D))
 *GET OUTPUT THIS WAY
x = output(1);
y = output(2);
width = output(3);
height = output(4);


 */

void mexFunction( int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[])
{

    int *inMatrix;               /* 1xN input matrix */
    int *outMatrix;              /* output matrix */
    int outputSize=4;

    const int xBlocks = ((xDim - windowSize) / stepSize) + 1;
    const int yBlocks = ((yDim - windowSize) / stepSize) + 1;
    const int totalBlocks = xBlocks * yBlocks;

    /* To check that there is only one input argument i.e inMatrix */
    if(nrhs!=1) {
        mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","One inputs required.");
    }
    /* To check that there is only one output argument i.e outMatrix */
    if(nlhs!=1) {
        mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","One output required.");
    }

    /* make sure the first input argument is not a scalar i.e it is a matrix */
    if(mxGetNumberOfElements(prhs[0])<=1) {
        mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar","Input multiplier must be a
1D matrix.");
    }

    /* make sure the first input argument is type double or uint8 */
    if( !mxIsDouble(prhs[0]) && !mxIsUint8(prhs[0]) && !mxIsUint32) {
        mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notDouble","Input matrix must be type
double or uint8.");
    }

    /* check that number of rows in first input argument is 1 */
    if(mxGetM(prhs[0])!=1) {
        mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector","Input must be a row 
vector.");
    }

    /* create a pointer to the real data in the input matrix  */
    inMatrix = (int32_T *) mxGetPr(prhs[0]);

    /* create the output matrix */
    plhs[0] = mxCreateNumericMatrix(1,(mwSize)outputSize,mxINT32_CLASS,mxREAL);

    /* get a pointer to the real data in the output matrix */
    outMatrix = (int32_T *) mxGetPr(plhs[0]);

    /* call the computational routine */
    CCAmex(inMatrix,outMatrix);
    return;
}
```

Other Codes

- analyseCriticalAreas.m
- CCA.m
- detectCircle.m
- detectCorner.m
- detectEdge.m
- getROI.m
- myDetectCircle.m

OPTIMIZATION

Vectorization:

```
for x=1:xDimension
       for
y=1:yDimension

img(x,y) = value;
       end
end
```

```
img(:) =
```

Parallel Computing:

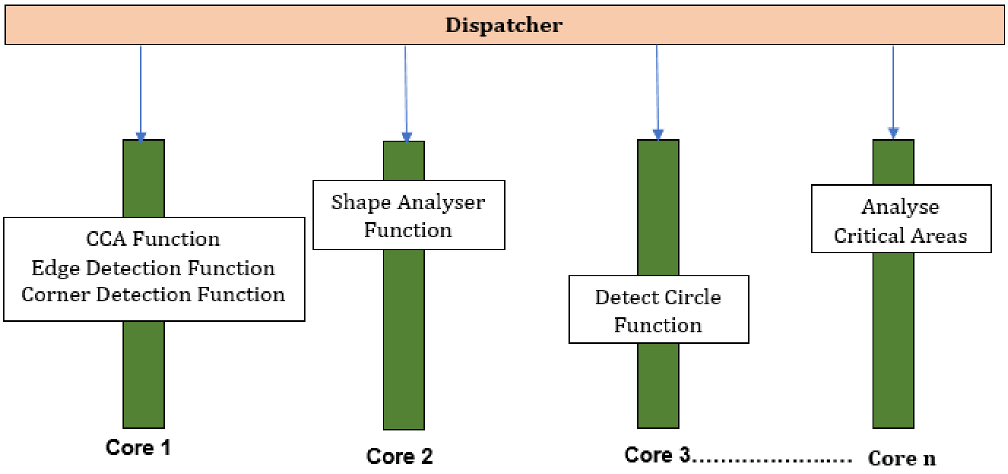


Figure 6. Figure Showing Parallel Dispatch of Functions

Legacy Coding:

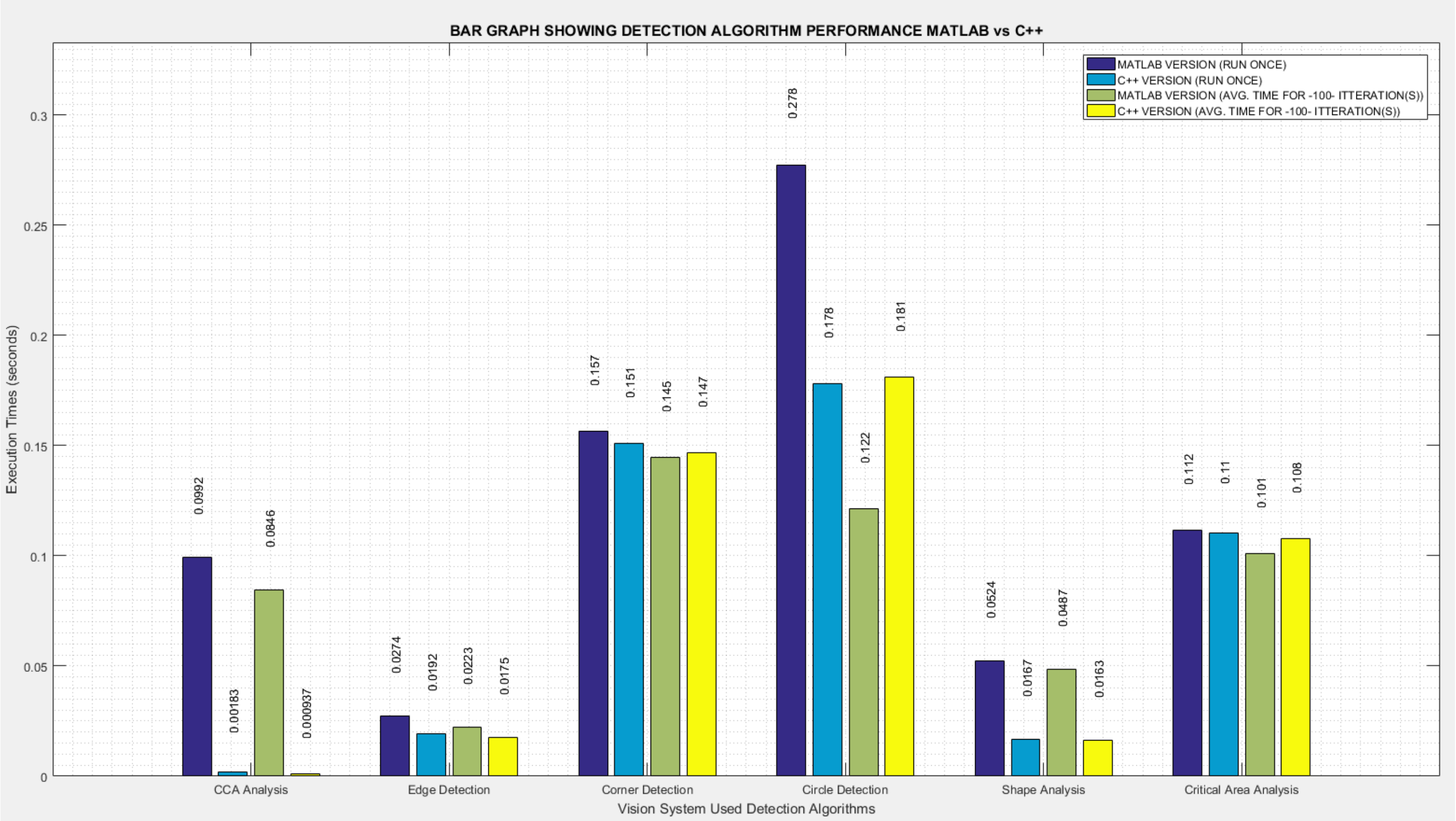


Figure 7. Figure Showing Improved Performance With Some Detection Function When Tested in C/C++

Heterogeneous Computing:

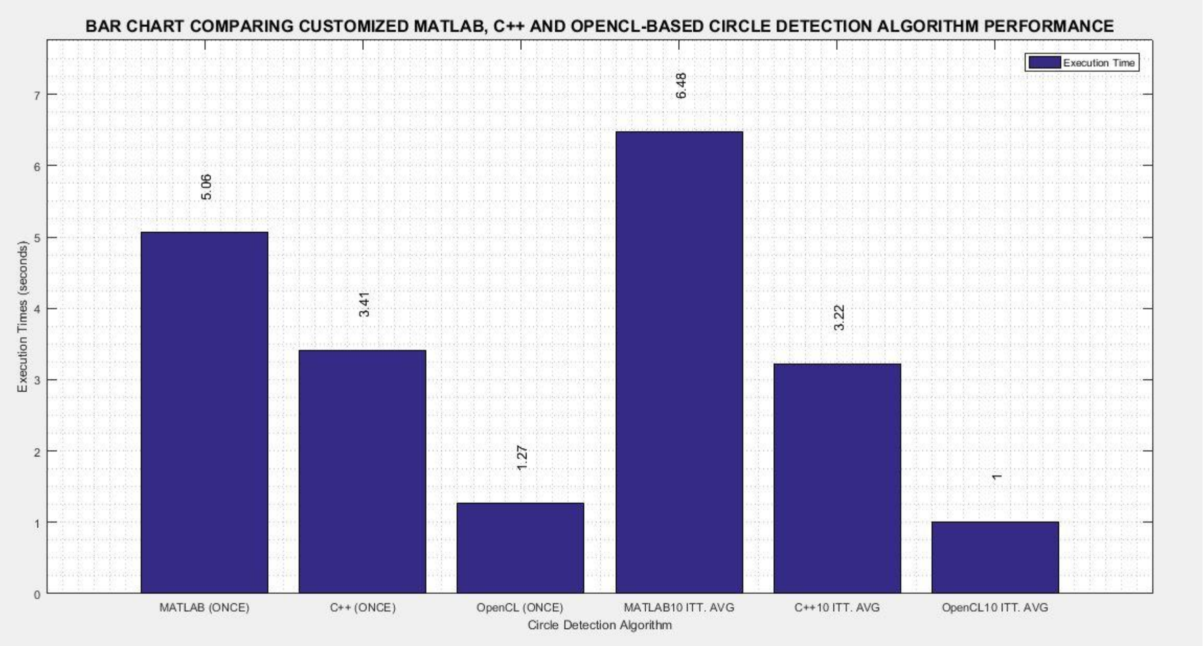


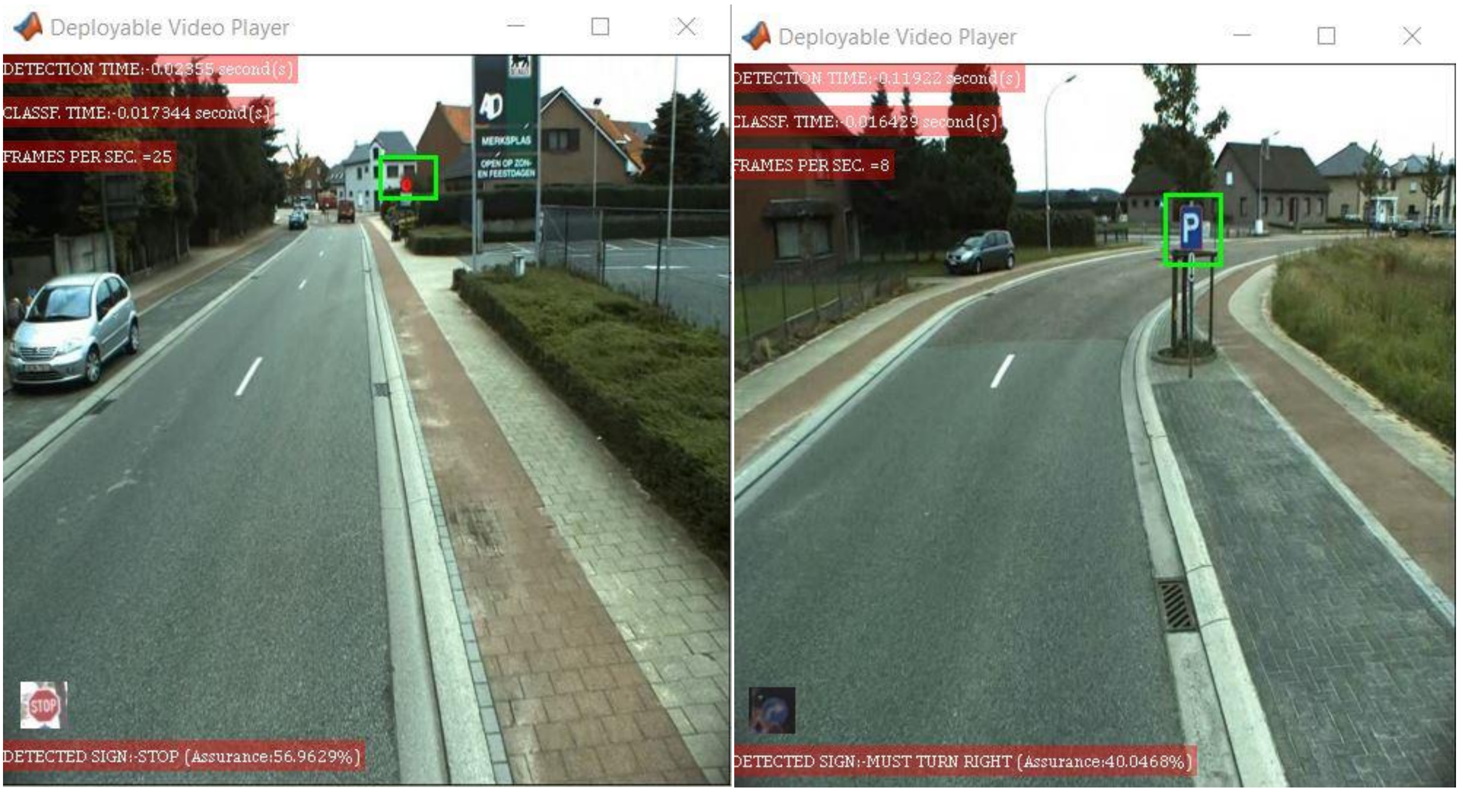
Figure 8. Figure Showing Improved Performance in Circle Detection Through the Use of a GPU

Code 3 OpenCL Kernel Code for Circle Detection Using Circular Hough Transform

```
//OpenCL kernel for circle detection using Circular Hough Transform

//pragmas to enable floating point operations
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable

//Already created Sine and Cosine Look-up table for improved speed
__constant double cosineTableGlobal[360] = { 1, 0.540302, -0.416147, -0.989992, -
0.653644…}

__constant double sineTableGlobal[360] = { 0, 0.841471, 0.909297, 0.14112, -0.756802, -
0.958924…}

__kernel void circularHough(
__global float *img,
__global int *accumulator,
__global int *xDimension,
__global int *yDimension,
__global int *range,
__global float *maxPixVal)
{
   //Define Variables
   int id = get_global_id(0); //Get Current Global ID
   int minRadius = (int) ceil(((double)((*yDimension)*0.1)));
   int radius=0;
   int a=0;
   int b=0;
   int pos = 0;

  //Get x,y,z index
   int idx = id;
   int zed = floor(((double)id) / (((double)(*xDimension)) *
((double)(*yDimension))));
   idx = id-(zed * (*xDimension) * (*yDimension));
   int wy = floor(((double)idx) / ((double)(*xDimension)));
   int ex = idx % (*xDimension);
   int val1 = (int) (wy + ((*xDimension)*ex));
   float val2= (float) (0.7*((float)(*maxPixVal)));
```

```
 if ((img[val1] > 0) && (img[val1] >= val2)) {
   for (int rad = 0; rad < (*range); rad++) { //Loop through radius range
     radius = minRadius + rad - 1;
     for (int tetha = 0; tetha < 360; tetha++) { //Every possible angle
        a = (int) ( ceil( ((double)ex) - (((double)radius) *
        ((double)cosineTableGlobal[tetha])) ));
        b = (int) ( ceil( ((double)wy) - (((double)radius) *
        ((double)sineTableGlobal[tetha])) ));
if (a > 0 && b > 0 && a <= (*xDimension) && b <= (*yDimension)) {
   pos = (int) ((rad * (*xDimension) * (*yDimension)) + (b *
     (*xDimension)) + a);
     atom_inc(&accumulator[pos]); //Increment accumulator content
}
}//end of theta loop
}//end of radius loop
}//end of if
}
```

FINAL RESULT:



```
DETECTION TIME:-0.02355 second(s)
CLASSF. TIME:-0.017344 second(s)
FRAMES PER SEC. =25
DETECTED SIGN:-STOP (Assurance:56.9629%)
```

```
DETECTION TIME:-0.11922 second(s)
CLASSF. TIME:-0.016429 second(s)
FRAMES PER SEC. =8
DETECTED SIGN:-MUST TURN RIGHT (Assurance:40.0468%)
```

Figure 6. Figure Showing Vision System in Operation MATLAB (right), C/C++ (left)