Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded

Department of Computer Science and Engineering

B.Tech. Projects

Implementation Of Convolutional Neural Network using MATLAB

Authors- U.V. Kulkarni, Shivani Degloorkar, Prachi Haldekar, Manisha Yedke

A step-by-step guide using MATLAB

Image classification is the task of classifying an image into one of the given categories based on visual content of an image. Neural networks are able to make predictions by learning the relationship between features of an image and some observed responses. In recent years, Convolutional neural networks (CNN) have achieved unprecedented performance in the field of image classification.
If you are a CNN rookie, it is advisable to go through the part of understanding CNN first and then continue on to know how to implement CNN using MATLAB. Else, you can skip to: Training CNN from scratch.

Understanding Convolutional neural network

So to start with CNN, let us first understand how computer sees an image. When an image is provided as input to a computer, it sees image as an array of pixel values. The size of array being m x n x r. Here, m, n represents height and width of the image respectively and r represents number of color channels. For instance, r value for rgb image is 3 (Figure 1) and that for gray is 1.

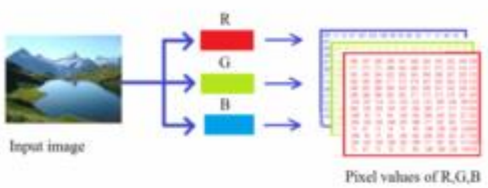


Figure 1: RGB image as seen by computer

Coming back, To build CNN, we use four main types of layers : Convolutional layer, Activation Layer, Pooling Layer and Fully Connected layer. The architecture of CNN may vary depending on the types and number of layers included. The types and number of layers included depend on application or data. For example, a smaller network with only one or two convolutional layers might be sufficient to learn small number of gray scale images. However, more complicated network with multiple convolutional and fully connected layers might be

needed for large number of colored images.
We will now discuss all these layers with their connectivity and parameters individually.

Convolutional Layer

The convolutional layer is the core building block of CNN. Input to convolutional layer is m x n x r dimensional array of pixel values.
In typical neural network, each neuron in previous layer is connected to every other neuron in hidden layer (Figure 2). When dealing with high-dimensional inputs such as images, it is impractical to connect hidden layer neurons to all neurons in the input layer. However, in CNN, only small region of neurons in input layer connect to neurons in hidden layer. These regions are referred to as local receptive fields (Figure 3).

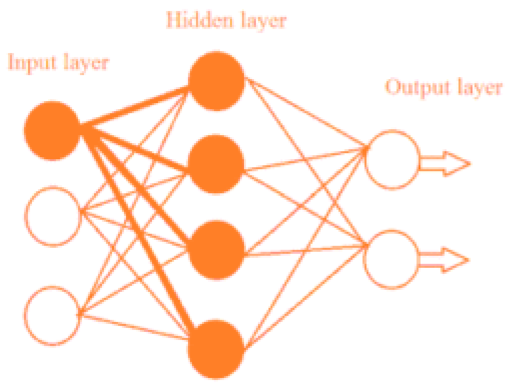


Figure 2: Typical neural network

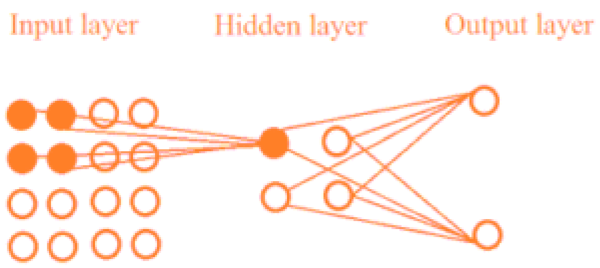


Figure 3: Convolutional neural network

These receptive local fields also know as kernels or filters, are the parameters of this layer. Every kernel is small along width and height as compared to input image size but is similar in depth to that of input. For example, given rgb input image of dimension 28 x 28 x 3, kernel might be of size 5 x 5 x 3 and that for gray image of same dimension, it might be of size 5 x 5 x 1.
So, what happens when an image is passed through convolutional layer ?
While passing an image through convolutional layer, we slide each kernel across the width and height of the input image. We compute element wise dot products between the entries of the kernel and the input image and add a bias term to it. This same computation is repeated across entire image i.e. convolving the input. The step size with which the kernel moves through a image is called a stride. After we slide the filter over the width and height of the input image, we form a 2-dimensional feature map. We have a set of these kernels and bias terms in a convolutional layer. Each feature map has a different set of kernel and a bias. Therefore, the

number of kernels determine the number of feature maps in the output of a convolutional layer. For eg, 6 different kernels convolved over an input image would produce 6 different feature maps.

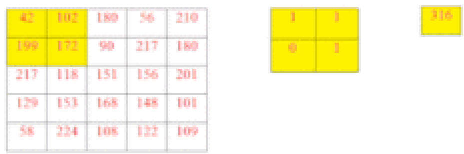


Figure 4: Sliding kernel 1 over input image to obtain feature map 1

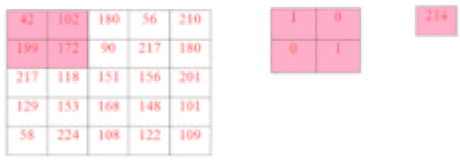


Figure 5: Sliding kernel 2 over input image to obtain feature map 2

The kernels consists of a set of learn-able weights which are randomly initialized with some small values at first. These weight matrices in form of kernel when slid over input image extracts some features from image. When we have multiple convolutional layers, these features at initial layers maybe some types of edge orientations or patches of colors and eventually at higher levels it consists of more complex or entire pattern itself. Feature maps are the output from convolutional layer. The size and number of feature maps produced depends on size of kernels, stride rate and number of kernels. For instance, consider a simple example where input is 2 dimensional 7 x 7 image. Now lets see how above mentioned parameters affect the size of output feature maps.

Size of kernels :

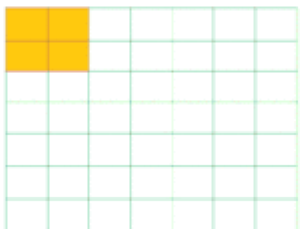


Figure 6

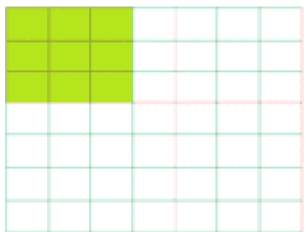
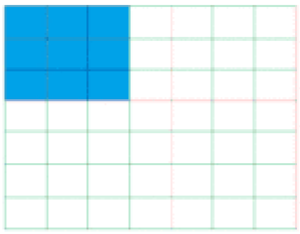


Figure 7

Stride rate :



7 x 7 image with kernel size 3 x 3 and stride rate as 2

Figure 8

7 x 7 image with kernel size 3 x 3 and stride rate as 3

Figure 9

Number of kernels :

Number of kernels decide number of feature maps produced. For example, 6 kernels produce 6 feature maps.

The problem seen in figure 9 can be solved by zero padding. Zero padding is basically adding rows or columns of zeros to the borders of an image input. It helps us control the output size of feature map.

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Figure 10: 9 x 9 image obtained after padding 7 x 7 image with zeros along the borders

Now, to sum up how these parameters affect output of convolutional layer i.e. feature maps, consider N x N image, K x K kernel, stride rate S and zero padding P. The size of output feature map can be given as:
Output size = ((N - K + 2 * P) / S) + 1

Activation Layer

In CNN it is convention to apply activation layer (non linear layer) after every convolutional layer. This is done in order to bring non linearity to the architecture after performing linear operations in convolutional layer. There are many types of nonlinear activation function such as a rectified linear unit (ReLU), tanh and sigmoid.

Pooling Layer

Pooling layers too are introduced between subsequent convolutional layers. These layers donot perform any learning tasks. It is a way of down-sampling i.e. reducing the dimension of the input to reduce amount of computation and parameters needed. Input to pooling layer are the series of features maps generated by convolutional layer. Basically what pooling layer does is, it groups a fixed number of units of a region and get a single value for that group. The region is selected using a window which in general is of size 2 x 2. This window slides with fixed stride which is most of the times fixed to two. It is worth noting that there are only two common variations of the pooling layer in practice: A pooling layer more commonly with window size = 2 and stride = 2 and window size = 3 and stride = 2. The pooling layer operates independently on every feature map and resizes it spatially. Therefore, number of pooled maps is equal to number of feature maps from previous convolutional layer.
Output size of pooling layer with n number of F x F dimensional feature maps as input, W as window size, S as stride rate can be given as n number of pooled maps with dimension P x P where,
P = ((F - W) / S) + 1
Note that it is uncommon to use zero padding in pooling layer.
Max- and average-pooling are two of the types of pooling. Max-pooling returns the maximum values whereas average-pooling outputs the average values of the fixed regions of its input.

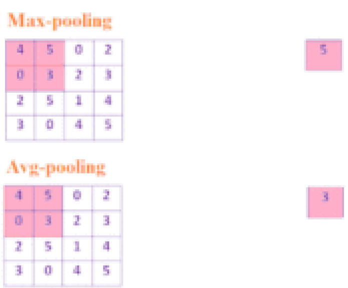


Figure 11: Pooling with window size 2 x 2 and stride 2

The main use of pooling is to make feature detection location independent. For example, assume we have two images on very large white background. In first image the letter is written in middle of image and in second image it is present at bottom right corner. Now, after we pass these two images through pooling layer we get reduced images which are nearly similar with letters somewhere in middle. This controls over-fitting. When we have over-fitting, our network is great with training set but is not good with testing set i.e. it is bad at generalization.

Fully Connected Layer

The convolutional and pooling layers are followed by one or more fully connected layers. All neurons in a fully connected layer connect to all the neurons in the previous layer. This layer combines all of the features learned by the previous layers across the network to identify the images. The way this fully connected layer works is that it looks at the output of the previous layer (which are the activation maps of high level features) and determines which features most correlate to a particular class. It then outputs the highest probability for that class. The output size of the fully connected layer of the network is equal to the number of classes of the data set.

Summary

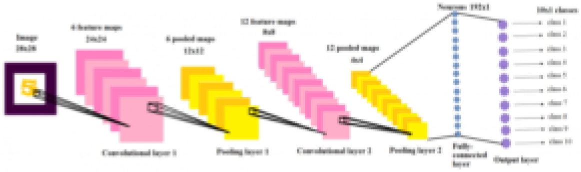


Figure 12: Complete CNN architecture

Now lets sum up how our network transforms the original image layer by layer from the original pixel values to the final class scores.
Input holds the pixel values of the image. For example 28x28x3 image.
Convolutional layer computes the output by computing dot product between kernels and a small region they are connected to in the input volume. This may result in output such as 24x24x6 if we decided to use 6 kernels of size 5x5x3.
Activation layer applies an element-wise activation function. This leaves the size of the output unchanged to 24x24x6.
Pooling layer performs a down-sampling operation along the width and height resulting in output such as 12x12x6.
Fully-connected layer computes the class scores resulting in output of size 10x1, where each of the 10 numbers correspond to a class score.

Back-propagation (Training CNN)

Our goal with back-propagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole. When training the network, there is additional layer called loss layer. This layer provides feedback to the neural network on whether it identified inputs correctly, and if not, how far off its guesses were. Here we define a loss function which quantifies our unhappiness with the scores across the training data. The function takes in desired output from user and the output produced by network and computes its badness. Loss over dataset is sum of loss over all inputs. This helps to guide the neural network to reinforce the right concepts at the time of train.
To learn more about how back-propagation in CNN updates weights throughout the network, you can refer: "Derivation of Back-propagation in Convolutional Neural Network (CNN)".

Training CNN from scratch

The first step of creating and training a new convolutional neural network is to define the network architecture. For this purpose we have used architecture as depicted in Figure 13. This is refereed from paper: "Derivation of Back-propagation in Convolutional Neural Network (CNN)". It consists of two convolutional and pooling layer and activation layers with uni polar sigmoid function. Also refer this paper for back-propagation algorithm further used in this guide for training the network.

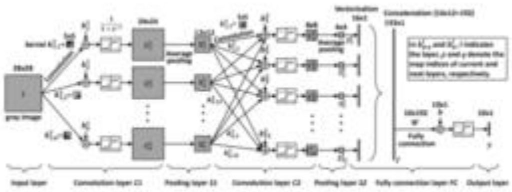


Figure 13: CNN Architecture

In this guide we will train our CNN model to identify Disguised faces for demo purpose. However, below implementation can be used to train network on any dataset.

Step 1: Data and Preprocessing

The dataset we used in this guide is cropped version of the IIIT-Delhi Disguise Version 1 face database (ID V1).
Note : This database can be cited in -
T. I. Dhamecha, R. Singh, M. Vatsa, and A. Kumar, Recognizing Disguised Faces: Human and Machine Evaluation, PLoS ONE, 9(7): e99212, 2014.
T. I. Dhamecha, A. Nigam, R. Singh, and M. Vatsa Disguise Detection and Face Recognition in Visible and Thermal Spectrums, In proceedings of International Conference on Biometrics, 2013 (Poster))
We manually split the entire dataset into two parts: disguised and undisguised. Moreover, the dataset doesn't come with an official train and test split, so we simply use 10% of the both disguised and undisguised data as a train set. Now, we have four data folders: Train_disguised, Train_Undisguised, Test_disguised, Test_Undisguised.
These are the examples of some of the images in dataset.

Disguised :

Undisguised :

Data reprocessing for this data-set will involve loading train data, resizing all images to same size, labeling images with desired output (for undisguised: 1,0 and for disguised: 0,1 since we would have two classes in output layer for undisguised and disguised) and then storing it in an array.

```
% Loading dataset images from train folder
disguised_src_file = dir('C:\Users\SHREE\Documents\MATLAB\train_disguised\*.jpg');
undisguised_src_file = dir('C:\Users\SHREE\Documents\MATLAB\train_undisguised\*.jpg');

% Initialising number of patterns
number_of_disguised_images = length(disguised_src_file);
number_of_undisguised_images = length(undisguised_src_file);

number_of_patterns = number_of_disguised_images + number_of_undisguised_images;

image_size = 28;

number_of_classes = 2;

% Initialising dataset and desired output matrix
dataset = zeros(image_size, image_size , number_of_patterns);
desired_output = zeros(number_of_classes , number_of_patterns);

pattern = 1;

% Reading image one by one from undisguised train folder
for i = 1 : number_of_undisguised_images
    filename = strcat('C:\Users\SHREE\Documents\MATLAB\train_undisguised\',undisguised_src_file(i).name);
    image = imread(filename);

    % Converting RGB image to black and white image
    black_white_image = im2bw(image);

    % Resizing obtained black and white image to required size
    black_white_resizeimage = imresize(black_white_image, [image_size image_size]);

    % Storing resized image to dataset array
    dataset(:,:,pattern)= black_white_resizeimage;
```

```
    % Setting desired output of first neuron to 1
    desired_output(1,pattern)=1;

    pattern = pattern + 1;
end

% Reading image one by one from disguised train folder
for j = 1 : number_of_disguised_images
    filename =
strcat('C:\Users\SHREE\Documents\MATLAB\train_disguised\',disguised_src_f
ile(j).name);
    image = imread(filename);

    % Converting RGB image to black and white image
    black_white_image = im2bw(image);

    % Resizing obtained black and white image to required size
    black_white_resizeimage = imresize(black_white_image, [image_size
image_size]);

    % Storing resized image to dataset array
    dataset(:,:,pattern)= black_white_resizeimage;

    % Setting desired output of second neuron to 1
    desired_output(2,pattern)=1;

    pattern = pattern + 1;
end
```

Step 2: Defining hyperparameters

In this example we use two convolutional and pooling layers. Therefore, we define two set of hyperparameters for two convolutional and pooling layers. Here, we also define other hyperparameters like number of training cycles, learning rate and max tolerable error.

```
number_of_training_cycles=1000000;
learning_rate = 0.1;
% Max tolerable error
emax = 0.01;

% Defining hyperparameters for convolutional layer 1
number_of_feature_maps_for_conv_layer1 = 12;
kernel_size_for_conv_layer1 = 5;

% Defining hyperparameters for pooling layer 1
window_size_for_pooling_layer1 = 2;

% Defining hyperparameters for convolutional layer 2
number_of_feature_maps_for_conv_layer2 = 12;
kernel_size_for_conv_layer2 = 5;

% Defining hyperparameters for pooling layer 2
window_size_for_pooling_layer2 = 2;
```

Step 3: Initialization of parameters and sizes of outputs of all layers

We initialize all biases with zeros, kernels and weights with random uniform distribution. We also define output sizes of each layer by assuming stride rate as one and no zero padding.

```
% Initialization of parameters and defining sizes of output layers

% Convolutional layer 1:

    % Initialization of kernels and biases with all zeros
    bias_weight_for_convolutional_layer1 =
zeros(number_of_feature_maps_for_conv_layer1, 1);
    kernel_for_convolutional_layer1 = zeros(kernel_size_for_conv_layer1,
kernel_size_for_conv_layer1, number_of_feature_maps_for_conv_layer1);

    % Initialising kernels with random uniform distribution
    kernel_initialisation_value_for_conv_layer1 =
sqrt(number_of_feature_maps_for_conv_layer1 /( (1 +
number_of_feature_maps_for_conv_layer1) *
kernel_size_for_conv_layer1^2));
    kernel_initialisation_range_for_conv_layer1 =
kernel_initialisation_value_for_conv_layer1 * 2;

    for i=1:number_of_feature_maps_for_conv_layer1
    kernel_for_convolutional_layer1(:,:,i) =
rand(kernel_size_for_conv_layer1 , kernel_size_for_conv_layer1) *
kernel_initialisation_range_for_conv_layer1 -
kernel_initialisation_value_for_conv_layer1;
    end

    % Initialising output feature maps of convolutional layer 1 with
zeros
    % Assuming stride rate as one and no zero padding
    size_of_conv_output1_image = image_size - kernel_size_for_conv_layer1
+ 1;
    output_of_conv_layer1 = zeros(size_of_conv_output1_image,
size_of_conv_output1_image, number_of_feature_maps_for_conv_layer1);


% Pooling layer 1:

    % Initialising output matrices with all zeros
    % Assuming stride rate as one and no zero padding
    size_of_pooling1_output_image = size_of_conv_output1_image /
window_size_for_pooling_layer1 ;
    pooling1_output=zeros(size_of_pooling1_output_image,
size_of_pooling1_output_image, number_of_feature_maps_for_conv_layer1);

% Convolutional layer 2:

    % Initialization of kernels and biases with all zeros
    kernel_for_conv_layer2 = zeros( kernel_size_for_conv_layer2 ,
kernel_size_for_conv_layer2 , number_of_feature_maps_for_conv_layer1 ,
number_of_feature_maps_for_conv_layer2 );
    bias_weight_for_conv_layer2 = zeros(
number_of_feature_maps_for_conv_layer2, 1 );

    % Convolutional layer 2 -- Initialising kernels with random uniform
distribution
```

```
    kernel_initialisation_value_for_conv_layer2 = 
sqrt(number_of_feature_maps_for_conv_layer2 /( 
(number_of_feature_maps_for_conv_layer1 + 
number_of_feature_maps_for_conv_layer2) * (kernel_size_for_conv_layer2 * 
kernel_size_for_conv_layer2)));
    kernel_initialisation_range_for_conv_layer2 = 
kernel_initialisation_value_for_conv_layer2 * 2;

    for i = 1 : number_of_feature_maps_for_conv_layer2
    kernel_for_conv_layer2(:,:,:,i) = rand(kernel_size_for_conv_layer2 , 
kernel_size_for_conv_layer2 , number_of_feature_maps_for_conv_layer1) * 
kernel_initialisation_range_for_conv_layer2 - 
kernel_initialisation_value_for_conv_layer2;
    end

    % Initialising output feature maps of convolutional layer 2 with 
zeros
    size_of_conv2_output = size_of_pooling1_output_image - 
kernel_size_for_conv_layer2 + 1;
    conv2_output = zeros( size_of_conv2_output, size_of_conv2_output, 
number_of_feature_maps_for_conv_layer2 );

% Pooling layer 2

    % Initialising output matrices with all zeros
    size_of_pooling2_output_image = size_of_conv2_output / 
window_size_for_pooling_layer2 ;
    pooling2_output = zeros(size_of_pooling2_output_image, 
size_of_pooling2_output_image, number_of_feature_maps_for_conv_layer2);

% Vectorization layer

    % Initialising vectorization output matrix with zeros
    vectorization_output_size = size_of_pooling2_output_image * 
size_of_pooling2_output_image;
    vectorization_output = zeros(vectorization_output_size, 1, 
number_of_feature_maps_for_conv_layer2);

% Concatenation layer

    % Initialising concatenation output matrix with zeros
    concatenation_output_size = vectorization_output_size * 
number_of_feature_maps_for_conv_layer2 ;
    concatenation_output = zeros(concatenation_output_size , 1);

% Fully Connected Layer

    % Weight matrix initialized with zeros and then with random uniform 
distribution
    weight_matrix_for_fully_connected_layer = zeros(number_of_classes, 
concatenation_output_size);

    weight_initialisation_value_for_fully_connected_layer = 
sqrt(number_of_classes /(concatenation_output_size + number_of_classes));
    weight_initialisation_range_for_fully_connected_layer = 
weight_initialisation_value_for_fully_connected_layer * 2;

    weight_matrix_for_fully_connected_layer(:,:)=rand(number_of_classes, 
concatenation_output_size).*
```

```
weight_initialisation_range_for_fully_connected_layer -
weight_initialisation_value_for_fully_connected_layer;

% Output Layer

    % Bias vector and output vector initialization
    bias_for_output_of_cnn = zeros(number_of_classes, 1);
    output_of_cnn = zeros (number_of_classes, 1);
```

Step 4: Defining adjustment vectors

This is a part of back-propagation. Here, we define adjustment vectors for each layer which tune parameters of each layer while training the network.

```
% Initialisation of adjustment vectors with zeros

    % Adjustment vector for weight
    delta_W_ij = zeros(number_of_classes, concatenation_output_size);

    % Adjustment vector for output of cnn
    Y_i = zeros(number_of_classes, 1);

    % Adjustment vector for bias at output layer
    delta_bias_i = zeros(number_of_classes, 1);

    % Adjustment vector for concatenation output
    delta_F = zeros(concatenation_output_size, 1);

    % Adjustment vector for pooling layer 2
    delta_S2_q = zeros(size_of_pooling2_output_image,
size_of_pooling2_output_image, number_of_feature_maps_for_conv_layer2);

    % Adjustment vector for convolutional layer 2
    delta_C2_q = zeros( size_of_conv2_output, size_of_conv2_output,
number_of_feature_maps_for_conv_layer2 );

    % Adjustment vector for convolutional layer 2 before sigmoid
function(activation function)
    delta_c2_q_sigmoid = zeros( size_of_conv2_output,
size_of_conv2_output, number_of_feature_maps_for_conv_layer2 );

    % Adjustment vector for rotated pooling layer 1
    delta_S1_rotate_p = zeros(size_of_pooling1_output_image,
size_of_pooling1_output_image, number_of_feature_maps_for_conv_layer1);

    % Adjustment vector for kernel of convolutional layer 2
    delta_k2_pq = zeros( kernel_size_for_conv_layer2,
kernel_size_for_conv_layer2, number_of_feature_maps_for_conv_layer1,
number_of_feature_maps_for_conv_layer2 );

    % Adjustment vector for bias of convolutional layer 2
    delta_b2_q = zeros( number_of_feature_maps_for_conv_layer2, 1 );

    % Adjustment vector for pooling layer 1
    delta_s1_p = zeros(size_of_pooling1_output_image,
size_of_pooling1_output_image,
number_of_feature_maps_for_conv_layer1);

    % Adjustment vector for convolutioal layer 1
    delta_c1_p = zeros( size_of_conv_output1_image,
size_of_conv_output1_image, number_of_feature_maps_for_conv_layer1 );
```

```
    % Adjustment vector for convolutional layer 1 before sigmoid
function(activation function)
    delta_c1_p_sigmoid = zeros( size_of_conv_output1_image,
size_of_conv_output1_image, number_of_feature_maps_for_conv_layer1 );

    % Adjustment vector for kernel of convolutional layer 1
    delta_k1_p = zeros( kernel_size_for_conv_layer1,
kernel_size_for_conv_layer1, number_of_feature_maps_for_conv_layer1
);

    % Adjustment vector for bias of convolutional layer 1
    delta_b1_p = zeros(number_of_feature_maps_for_conv_layer1, 1);
```

Step 5: Convolutional layer 1

This part of the program takes in input image matrix and one kernel at a time, convolves kernel over input and returns the output by applying activation on each element of output. We make a call to this function using a for loop. We send input image matrix, expected size of output (as calculated in step 3 so that there is no need of function to calculate it), kernel size, kernel and bias as parameters. The function returns a feature map with activation applied on it. We store each of this output along depth of 3D array.

Original image as input:

```
% Processing image through Convolutional layer 1

    for i = 1 : number_of_feature_maps_for_conv_layer1
        % Function call to convolutional layer
        output_of_conv_layer1(:,:,i) = convolutional_layer(image,
size_of_conv_output1_image, kernel_size_for_conv_layer1,
kernel_for_convolutional_layer1(:,:,i),
bias_weight_for_convolutional_layer1(i,1));
    end
```

```
% Function for Convolutional layer 1

function conv_output = convolutional_layer(input_image ,
size_of_output_image , kernel_size , kernel , bias_weight)

    conv_output = zeros(size_of_output_image , size_of_output_image);

    for rows = 1 : size_of_output_image
        for cols = 1 : size_of_output_image
            temp = 0;
            for kernelrows = 0 : (kernel_size - 1)
                for kernelcols = 0 : (kernel_size - 1)
                    temp = temp + input_image( rows + kernelrows , cols +
kernelcols ) * kernel( 1 + kernelrows , 1 + kernelcols);
                end
```

```
            end
            net = bias_weight + temp;
            conv_output(rows,cols) = activation(net);
        end
    end
end

function result = activation(net)
    result = 1/(1+exp(-net));
end
```

Convolved image as output:

In above image since all the edges are highlighted, we can roughly infer that first convolutional layer acts as edge detector.

Step 6: Pooling layer 1

This part of the program takes in output of convolutional layer 1 one by one and window size, does average pooling with stride rate as 2 and returns the pooled output. We pass size of convolutional layer output, expected size of pooled output, window size for pooling layer 1, convolutional layer output.

Convolved image as input:

```
% Function for pooling layer

function pooling_output = pooling_layer(size_of_conv_output_image ,
size_of_output_image , window_size_for_pooling_layer , conv_layer_output)

    pooling_output = zeros(size_of_output_image,size_of_output_image);

    pooling_output_rows=1;
    pooling_output_cols=1;

    for rows = 1 : 2 : size_of_conv_output_image
        for cols = 1 : 2 : size_of_conv_output_image
            temp = 0;
```

```
            for windowrows = 0 : (window_size_for_pooling_layer - 1)
                for windowcols = 0 :(window_size_for_pooling_layer - 1)
                    temp = temp +
conv_layer_output(rows+windowrows,cols+windowcols);
                end
            end
            average=temp/(window_size_for_pooling_layer *
window_size_for_pooling_layer);
            pooling_output(pooling_output_rows , pooling_output_cols) =
average;
            pooling_output_cols = pooling_output_cols + 1 ;
        end
        pooling_output_cols=1;
        pooling_output_rows = pooling_output_rows + 1 ;
    end
end
```

Pooled image as output:

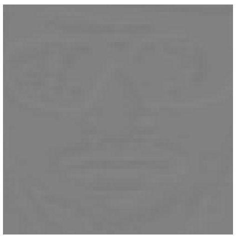
Pooling layer does not participate in feature detection. We can see that information in retained in above image. Only the dimensions change.

Step 7: Convolutional layer 2
In this layer, set of kernels operate over pooled maps. Each pooled map has its own set of kernels. Here, number of set of kernels = number of pooled maps from previous pooling layer. A set of kernel consists of kernels = number of feature maps for convolutional layer 2. Activation applied on summation of values after convolving ith kernel of each set over its pooled map at each position gives value of ith feature map at that position. To understand more precisely refer Figure 13.
Pooled map as input:

```
% Processing image through Convolutional layer 2

    for i = 1 : number_of_feature_maps_for_conv_layer2
        conv2_output(:,:,i) =
convolutional_layer2(bias_weight_for_conv_layer2(i,1),
size_of_conv2_output, number_of_feature_maps_for_conv_layer1,
```

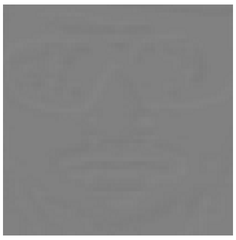
```
kernel_size_for_conv_layer2, kernel_for_conv_layer2(:,:,:,i) ,
pooling1_output);
    end
% Function for convolutional layer 2

function conv_output2 = convolutional_layer2(bias_weight_for_conv_layer2
, size_of_conv2_output , number_of_feature_maps_for_conv_layer1
,kernel_size_for_conv_layer2, kernel_for_conv_layer, pooling1_output)

    conv_output2 = zeros(size_of_conv2_output , size_of_conv2_output);

    for rows = 1 : size_of_conv2_output
        for cols = 1 : size_of_conv2_output
            temp = 0;
            for feature_map_number = 1
: number_of_feature_maps_for_conv_layer1
                for kernelrows = 0 : (kernel_size_for_conv_layer2 - 1)
                    for kernelcols = 0 : (kernel_size_for_conv_layer2 -
1)
                        temp = temp + pooling1_output( rows + kernelrows
, cols + kernelcols , feature_map_number) * kernel_for_conv_layer( 1 +
kernelrows , 1 + kernelcols , feature_map_number);
                    end
                end

            end
            net = bias_weight_for_conv_layer2 + temp;
            conv_output2(rows,cols) = activation(net);
        end
    end
    figure;imshow(conv_output2);
end


function result = activation(net)
    result = 1/(1+exp(-net));
end
```

Convolved image as output:



This layer detects more complex features. For example, curves detection.

Step 8: Pooling layer 2
Pooling layer remains same as in step 6.
Pooled image as output:



Step 9: Vectorization and Concatenation layer

Here, vectorization layer is used to vectorize pooled maps. For example, if 12 pooled maps of size 4 x 4 are present, each pooled map produces a vector of size 16 which is obtained by scanning each of them column by column. Now, these 12 vectors of size 16 are concatenated one after other to produce a vector of size 12 x 16 = 192. This is done in concatenation layer. Output of concatenation layer is input to fully connected layer.

```
% Vectorizing image
    for i = 1 : number_of_feature_maps_for_conv_layer2
        vectorization_output(:,:,i) =
vectorization(size_of_pooling2_output_image, pooling2_output(:,:,i));
    end
```

```
% Function for Vectorization layer
function vectorization_output =
vectorization(size_of_pooling2_output_image , pooling2_output)

    vectorization_output=zeros(size_of_pooling2_output_image *
size_of_pooling2_output_image , 1);

    index=0;

    for cols = 1 : size_of_pooling2_output_image
        for rows = 1 : size_of_pooling2_output_image
           index=index+1;
           vectorization_output(index,1)= pooling2_output(rows,cols);
        end
    end
end
```

```
% Concatenating image

index=0;
for i=1:number_of_feature_maps_for_conv_layer2
    for j = 1:vectorization_output_size
        index = index+1;
        concatenation_output(index) = vectorization_output(j, 1, i);
    end
end
```

Step 10: Fully connected layer

In this step, we multiply weight matrix initialized by random uniform distribution and concatenation output. We then add bias and apply activation function on it.

```
% Computing Output of CNN

    output_of_cnn = weight_matrix_for_fully_connected_layer *
concatenation_output ;
    output_of_cnn = output_of_cnn + bias_for_output_of_cnn;
```

```
    for i=1:number_of_classes       % Applying activation function on net
        net=output_of_cnn(i,1);
        result = 1/(1+exp(-net));
        output_of_cnn(i,1)=result;
    end
```

Step 11: Training cycle

After passing image through all the above layers, we calculate loss function to check how much the actual output deviate from our desired output. We then start computing adjustment vectors using formulas in this paper.

Here, we use two for loops. One for training cycles to train the network until error lowers down to maximum tolerable error and other for number of patterns in our data-set.

```
for training_cycle = 1 : number_of_training_cycles
    error = 0;
    for pattern = 1:number_of_patterns

        image = dataset(:,:,pattern);

        % Processing image through Convolutional layer 1
        for i = 1 : number_of_feature_maps_for_conv_layer1
            output_of_conv_layer1(:,:,i) = convolutional_layer(image, 
size_of_conv_output1_image, kernel_size_for_conv_layer1, 
kernel_for_convolutional_layer1(:,:,i), 
bias_weight_for_convolutional_layer1(i,1));
        end

        % Processing image through Pooling layer 1
        for i = 1 : number_of_feature_maps_for_conv_layer1
            pooling1_output(:,:,i) = 
pooling_layer(size_of_conv_output1_image, size_of_pooling1_output_image, 
window_size_for_pooling_layer1, output_of_conv_layer1(:,:,i));
        end

        % Processing image through Convolutional layer 2
        for i = 1 : number_of_feature_maps_for_conv_layer2
            conv2_output(:,:,i) = 
convolutional_layer2(bias_weight_for_conv_layer2(i,1), 
size_of_conv2_output, number_of_feature_maps_for_conv_layer1, 
kernel_size_for_conv_layer2, kernel_for_conv_layer2(:,:,:,i) , 
pooling1_output);
        end


        % Processing image through Pooling layer 2
        for i = 1 : number_of_feature_maps_for_conv_layer2
            pooling2_output(:,:,i) = pooling_layer(size_of_conv2_output
, size_of_pooling2_output_image , window_size_for_pooling_layer2 , 
conv2_output(:,:,i));
        end

        % Vectorizing image
        for i = 1 : number_of_feature_maps_for_conv_layer2
```

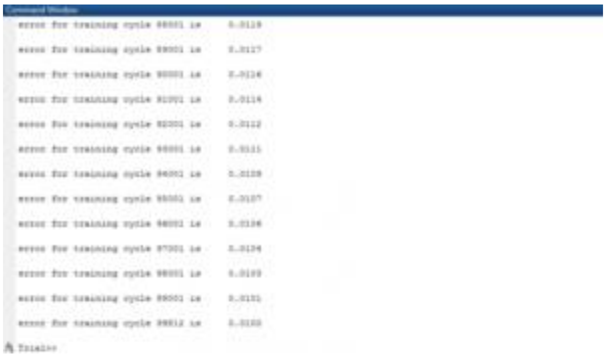
```
            vectorization_output(:,:,i) = 
vectorization(size_of_pooling2_output_image, pooling2_output(:,:,i));
        end

        % Concatenating image
        index=0;
        for i=1:number_of_feature_maps_for_conv_layer2
            for j = 1:vectorization_output_size
                index = index+1;
                concatenation_output(index) = vectorization_output(j, 1, 
i);
            end
        end

        % Computing Output of CNN
        output_of_cnn = weight_matrix_for_fully_connected_layer * 
concatenation_output ;
        output_of_cnn = output_of_cnn + bias_for_output_of_cnn;


        for i=1:number_of_classes       % Applying activation function 
on net
            net=output_of_cnn(i,1);
            result = 1/(1+exp(-net));
            output_of_cnn(i,1)=result;
        end

        % Calculating Loss and error
        loss=0.5*(norm(output_of_cnn - desired_output(:,pattern)))^2;
        error = error + loss;

        % Computing adjustment vector Y i.e vector of error signal terms 
requird to calculate weight and bias adjustment vectors
        for i = 1 : number_of_classes
            Y_i(i) = (output_of_cnn(i,1) - desired_output(i,pattern)) * 
output_of_cnn(i,1) * (1 - output_of_cnn(i,1));
        end
        delta_W_ij = Y_i * transpose(concatenation_output); % Computing 
weight adjustment vector
        delta_bias_i = Y_i;                                 % Computing 
bias adjustment vector



        % Computing adjustment vector for concatenation output
         delta_F = transpose(weight_matrix_for_fully_connected_layer) * 
Y_i;

        % Computing adjustment vector for pooling layer 2
        delta_S2_q = reshape(delta_F, size_of_pooling2_output_image, 
size_of_pooling2_output_image, number_of_feature_maps_for_conv_layer2);

        % Computing adjustment vector for convolutional layer 2 by 
upsampling
        for q = 1 : number_of_feature_maps_for_conv_layer2
            for i = 1 : size_of_conv2_output
                for j = 1 : size_of_conv2_output
                   delta_C2_q(i,j,q) = (1/4) * delta_S2_q(ceil(i/2), 
ceil(j/2),q);
                end
```

```
            end
        end

        % computing adjustment vector for convolutional layer 2 before sigmoid function
        for q = 1 : number_of_feature_maps_for_conv_layer2
            for i = 1 : size_of_conv2_output
                for j = 1 : size_of_conv2_output
                    delta_c2_q_sigmoid(i, j, q) = delta_C2_q(i, j, q) * conv2_output(i, j, q) * (1 - conv2_output(i, j, q));
                end
            end
        end

        % Computing adjustment vector for rotated pooling layer 1
        delta_S1_rotate_p = rot90(pooling1_output, 2);

        % computing adjustment vector for kernels of convolutional layer 2
        for p = 1 : number_of_feature_maps_for_conv_layer1
            for q = 1 : number_of_feature_maps_for_conv_layer2
                delta_k2_pq(:,:,p,q)=conv2(delta_S1_rotate_p(p), delta_c2_q_sigmoid(q),'valid');
            end
        end

        % Computing adjustment vector for bias of convolutional layer 2
        for q = 1 : number_of_feature_maps_for_conv_layer2
        temp=0;
            for i = 1 : size_of_conv2_output
                for j = 1 : size_of_conv2_output
                    temp = temp + delta_c2_q_sigmoid(i,j,q);
                end
            end
            delta_b2_q(q,1)=temp;
        end

        % Rotating kernel of layer 2 by 180
        k2_pq_rotate = rot90(kernel_for_conv_layer2, 2);

        % Computing adjustment vector for pooling layer 1
        for p = 1 : number_of_feature_maps_for_conv_layer1
            temp = zeros(size_of_pooling1_output_image, size_of_pooling1_output_image);
            for q = 1 : number_of_feature_maps_for_conv_layer2
                temp(:,:)=temp + conv2(delta_c2_q_sigmoid(:,:,q), k2_pq_rotate(:,:,p,q), 'full');
            end
            delta_s1_p(:,:,p) = temp;
        end

        % Computing adjustment vector for convolutional layer 1 by upsampling
        for p = 1 : number_of_feature_maps_for_conv_layer1
            for i = 1 : size_of_conv_output1_image
                for j = 1 : size_of_conv_output1_image
                    delta_c1_p(i, j, p) = (1/4) * delta_s1_p( ceil(i/2), ceil(j/2), p );
                end
            end
```

```
        end

        % computing adjustment vector for convolutional layer 1 before sigmoid function
        for p = 1 : number_of_feature_maps_for_conv_layer1
            for i = 1 : size_of_conv_output1_image
                for j = 1 : size_of_conv_output1_image
                    delta_c1_p_sigmoid(i, j, p) = delta_c1_p(i, j, p) * output_of_conv_layer1(i , j , p) * (1 - output_of_conv_layer1(i , j , p));
                end
            end
        end

        % Rotating input pattern
         input_pattern_rotate = rot90(image, 2);

        % Computing adjustment vector for kernel of convolutional layer 1
        for p = 1: number_of_feature_maps_for_conv_layer1
            delta_k1_p(:,:,p) = conv2(input_pattern_rotate, delta_c1_p_sigmoid(:,:,p) , 'valid');
        end

        % Computing adjustment vector for bias of convolutional layer 1
        for p = 1 : number_of_feature_maps_for_conv_layer1
            temp=0;
            for i = 1 : size_of_conv_output1_image
                for j = 1 : size_of_conv_output1_image
                    temp = temp + delta_c1_p_sigmoid(i,j,p);
                end
            end
            delta_b1_p(p,1)=temp;
        end


        % Parameter Update

        % Adjusting kernel for convolutional layer 1
        for p = 1 : number_of_feature_maps_for_conv_layer1
            kernel_for_convolutional_layer1(:,:,p) = kernel_for_convolutional_layer1(:,:,p) - learning_rate * delta_k1_p(:,:,p);
        end

        % Adjusting bias for convolutional layer 1
        for p = 1 : number_of_feature_maps_for_conv_layer1
            bias_weight_for_convolutional_layer1(p,1) = bias_weight_for_convolutional_layer1(p,1) - learning_rate * delta_b1_p(p,1);
        end

        % Adjusting kernel for convolutional layer 2
         for p = 1 : number_of_feature_maps_for_conv_layer1
            for q = 1 : number_of_feature_maps_for_conv_layer2
                kernel_for_conv_layer2(:,:,p,q) = kernel_for_conv_layer2(:,:,p,q) - learning_rate * delta_k2_pq(:,:,p,q);
            end
         end
```

```
        % Adjusting bias for convolutional layer 2
        for q = 1 : number_of_feature_maps_for_conv_layer2
            bias_weight_for_conv_layer2(q,1) = bias_weight_for_conv_layer2(q,1) - learning_rate * delta_b2_q(q,1);
        end

        % Adjusting weight matrix
        weight_matrix_for_fully_connected_layer = weight_matrix_for_fully_connected_layer - learning_rate * delta_W_ij;

        % Adjusting bias
        bias_for_output_of_cnn = bias_for_output_of_cnn - learning_rate * delta_bias_i;


    end

    % Printing error after 1000 training cycles
    if (mod (training_cycle,1000))==1
    fprintf('error for training cycle %i is ',training_cycle);
    disp(error);
    end
    if(error <= emax)
        fprintf('error for training cycle %i is ',training_cycle);
        disp(error);
        break
    end
end

save C:\Users\SHREE\Documents\MATLAB\Test_CNN_2CLASS.mat;
```

Output:



Above are some snippets of errors obtained for training cycles. Instead of printing error after each training cycle, we print it after finishing 1000 training cycles so that it is easy to monitor errors on screen. Training stops after reaching specified tolerable error. We then save all kernels, weights and biases obtained after training in .mat file to use while testing. At this stage our model is ready to test.

Step 12: Testing model

In order to test model, we need to take test data (data not used in training), label it and then evaluate accuracy of results.

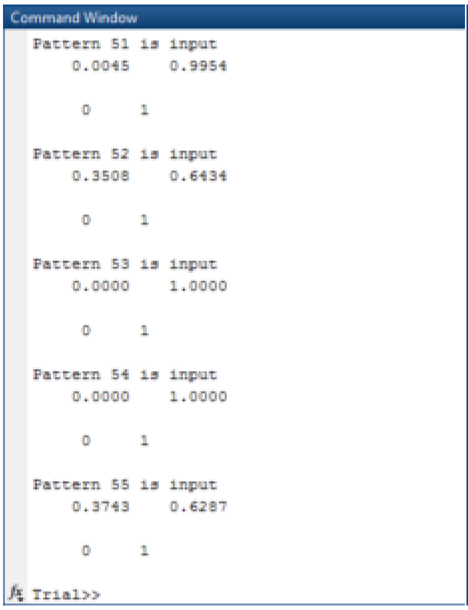
To label and create test data-set we use same code snippet as in step 1 by just changing train folder names to test folder names.

Now, we load kernels, weights and biases obtained after training from .mat file and pass our testing data through feed-forward part of network. We check obtained outputs against our desired outputs for each testing data and calculate accuracy.

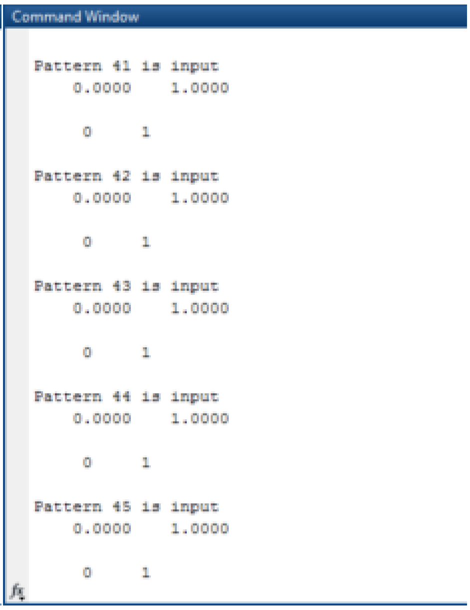
```
% Load trained parameters
    load C:\Users\SHREE\Documents\MATLAB\Test_CNN_2CLASS.mat;

% Load test dataset and its desired output
    load C:\Users\SHREE\Documents\MATLAB\disguiseddataset.mat;

    for pattern = 1:number_of_patterns
            fprintf('Pattern %i is input\n',pattern);
            image = dataset(:,:,pattern);

            % Processing image through Convolutional layer 1
            for i = 1 : number_of_feature_maps_for_conv_layer1
                output_of_conv_layer1(:,:,i) = convolutional_layer(image,
size_of_conv_output1_image, kernel_size_for_conv_layer1,
kernel_for_convolutional_layer1(:,:,i),
bias_weight_for_convolutional_layer1(i,1));
            end

            % Processing image through Pooling layer 1
            for i = 1 : number_of_feature_maps_for_conv_layer1
                pooling1_output(:,:,i) =
pooling_layer(size_of_conv_output1_image, size_of_pooling1_output_image,
window_size_for_pooling_layer1, output_of_conv_layer1(:,:,i));
            end

            % Processing image through Convolutional layer 2
            for i = 1 : number_of_feature_maps_for_conv_layer2
                conv2_output(:,:,i) =
convolutional_layer2(bias_weight_for_conv_layer2(i,1),
size_of_conv2_output, number_of_feature_maps_for_conv_layer1,
```

```
kernel_size_for_conv_layer2, kernel_for_conv_layer2(:,:,:,i) ,
pooling1_output);
            end


            % Processing image through Pooling layer 2
            for i = 1 : number_of_feature_maps_for_conv_layer2
                pooling2_output(:,:,i) =
pooling_layer(size_of_conv2_output , size_of_pooling2_output_image ,
window_size_for_pooling_layer2 , conv2_output(:,:,i));
            end

            % Vectorizing image
            for i = 1 : number_of_feature_maps_for_conv_layer2
                vectorization_output(:,:,i) =
vectorization(size_of_pooling2_output_image, pooling2_output(:,:,i));
            end

            % Concatenating image
            index=0;
            for i=1:number_of_feature_maps_for_conv_layer2
                for j = 1:vectorization_output_size
                    index = index+1;
                    concatenation_output(index) = vectorization_output(j,
1, i);
                end
            end

            % Computing Output of CNN for image
            output_of_cnn = weight_matrix_for_fully_connected_layer *
concatenation_output ;
            output_of_cnn = output_of_cnn + bias_for_output_of_cnn;


            for i=1:number_of_classes       % Applying activation
function on net
                net=output_of_cnn(i,1);
                result = 1/(1+exp(-net));
                output_of_cnn(i,1)=result;
            end

            % Comparing obtained output against desired output
            disp(transpose(output_of_cnn));
            disp(transpose(desired_output(:,pattern)));
    end
```

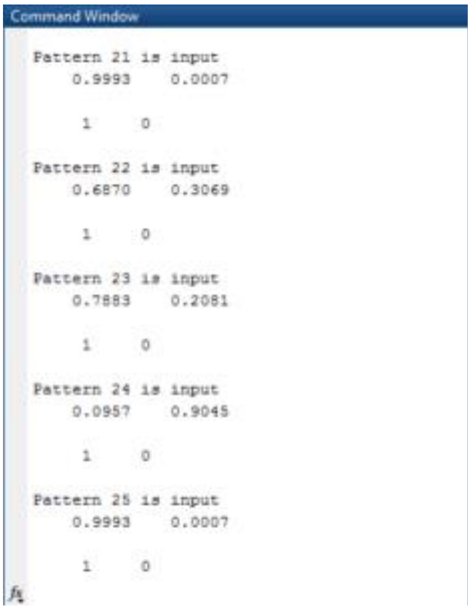
Output:



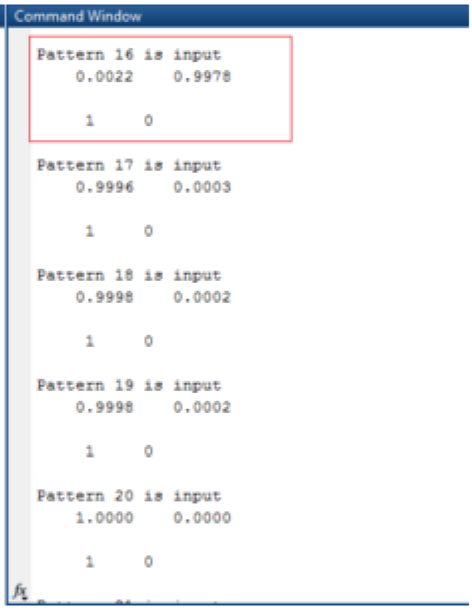
```
Command Window
Pattern 51 is input
     0.0045     0.9954

      0     1

Pattern 52 is input
     0.3508     0.6434

      0     1

Pattern 53 is input
     0.0000     1.0000

      0     1

Pattern 54 is input
     0.0000     1.0000

      0     1

Pattern 55 is input
     0.3743     0.6287

      0     1

fx Trial>>
```



```
Command Window

Pattern 41 is input
     0.0000     1.0000

      0     1

Pattern 42 is input
     0.0000     1.0000

      0     1

Pattern 43 is input
     0.0000     1.0000

      0     1

Pattern 44 is input
     0.0000     1.0000

      0     1

Pattern 45 is input
     0.0000     1.0000

      0     1
fx
```



```
Command Window

Pattern 21 is input
     0.9993     0.0007

      1     0

Pattern 22 is input
     0.6870     0.3069

      1     0

Pattern 23 is input
     0.7883     0.2081

      1     0

Pattern 24 is input
     0.0957     0.9045

      1     0

Pattern 25 is input
     0.9993     0.0007

      1     0
fx
```



```
Command Window
Pattern 16 is input
     0.0022     0.9978

      1     0

Pattern 17 is input
     0.9996     0.0003

      1     0

Pattern 18 is input
     0.9998     0.0002

      1     0

Pattern 19 is input
     0.9998     0.0002

      1     0

Pattern 20 is input
     1.0000     0.0000

      1     0
fx
```

Command Window

```
Pattern 31 is input
     0.0000     1.0000

      0     1

Pattern 32 is input
     0.0000     1.0000

      0     1

Pattern 33 is input
     0.0003     0.9997

      0     1

Pattern 34 is input
     0.0003     0.9997

      0     1

Pattern 35 is input
     0.0000     1.0000

      0     1
```

Command Window

```
Pattern 26 is input
     0.0000     1.0000

      0     1

Pattern 27 is input
     0.0000     1.0000

      0     1

Pattern 28 is input
     0.0000     1.0000

      0     1

Pattern 29 is input
     0.0131     0.9874

      0     1

Pattern 30 is input
     0.0025     0.9976

      0     1
```

Command Window

```
Pattern 11 is input
     0.9568     0.0429

      1     0

Pattern 12 is input
     0.9923     0.0076

      1     0

Pattern 13 is input
     0.9999     0.0001

      1     0

Pattern 14 is input
     1.0000     0.0000

      1     0

Pattern 15 is input
     1.0000     0.0000

      1     0
```

Command Window

```
Pattern 46 is input
     0.0000     1.0000

      0     1

Pattern 47 is input
     0.0000     1.0000

      0     1

Pattern 48 is input
     0.9750     0.0249

      0     1

Pattern 49 is input
     0.0000     1.0000

      0     1

Pattern 50 is input
     0.6326     0.3678

      0     1
```

Command Window

```
Pattern 6 is input
     1.0000     0.0000

      1     0

Pattern 7 is input
     0.9620     0.0383

      1     0

Pattern 8 is input
     1.0000     0.0000

      1     0

Pattern 9 is input
     0.9999     0.0001

      1     0

Pattern 10 is input
     1.0000     0.0000

      1     0
```

Command Window

```
Pattern 36 is input
     0.0000     1.0000

      0     1

Pattern 37 is input
     1.0000     0.0000

      0     1

Pattern 38 is input
     1.0000     0.0000

      0     1

Pattern 39 is input
     0.0000     1.0000

      0     1

Pattern 40 is input
     0.0000     1.0000

      0     1
```

```
Command Window
Trial>> TestCNN
Pattern 1 is input
    1.0000    0.0000

     1     0

Pattern 2 is input
    0.9886    0.0113

     1     0

Pattern 3 is input
    0.9861    0.0136

     1     0

Pattern 4 is input
    1.0000    0.0000

     1     0

Pattern 5 is input
    0.9999    0.0001

     1     0

fx
```

Accuracy of CNN does get better by adding few more layers. It definitely gives better accuracy for training data but performs really poor on test data i.e. it causes over-fitting. On top of that, it also depends on number of filters used in each convolutional layer. Therefore, to train a model we need to find perfect trade-off with trail and error method.