CONTENTS

OBJECTIVE

The objective of this project is to get a better and a deep understanding of the working of neural networks and how to apply these networks in machine learning in form of algorithms which make our life easy by having a variety of real life applications.

INTRODUCTION

Gesture recognition is a topic in computer science and language technology with the goal of interpreting human gestures via mathematical algorithms. Gestures can originate from any bodily motion or state but commonly originate from the face or hand. Our project mainly focusses on hand gesture recognition. In real life this gesture recognition can be used to control or interact with devices without physically touching them. Many approaches have been made using cameras and computer vision algorithms to interpret sign language. However, the identification and recognition of posture, gait, proxemics , and human behaviour is also the subject of gesture recognition techniques.

Gesture recognition can be seen as a way for computers to begin to understand human body language, thus building a richer bridge between machines and humans than primitive text user interfaces or even GUIs (graphical user interfaces), which still limit the majority of input to keyboard and mouse and interact naturally without any mechanical devices. Using the concept of gesture recognition, it is possible to point a finger at this point will move accordingly. This could make conventional input on devices such and even redundant. In this project, we attempt to design a neural network using back propagation algorithm in GNU Octave software .

LITERATURE SURVEY

Data glove in essence is a wired interface with certain tactile or other sensory units that were attached to the fingers or joints of the glove, worn by the user. The tactile switches, optical goniometer or resistance sensors which measure the bending of different joints offered crude measurements as to determine a hand was open or closed and some finger joints were straight or bent. These results were mapped to unique gestures and were interpreted by a computer. The advantage of such a simple device was that there was no requirement for any kind of pre-processing. Using cameras to recognize hand gestures started very early along with the development of the first wearable data gloves. There were many hurdles at that time in interpreting camera based gestures. Coupled with very low computing power available only on main frame computers, cameras offered very poor resolution along with colour inconsistency. The theoretical developments that lead to identifying skin segmentation were in its infancy and were not widely recognized for its good performance that we see today. Despite these hurdles, the first computer vision gesture recognition system was reported in 1980s. Our project on hand based gesture recognition is more natural and comfortable for its users, as it does not constrain the flexibility of hand movements.

TARGET DELIVERABLES

- Paralyzed people
- Astronauts on a spacewalk
- Controlling Robots
- Interpretation of Sign Language

CODE

create image dataset

```
pkg load image;

% Remove previous folders
rmdir('dataset_resized', 's');
% Create required directories
mkdir('dataset_resized');
mkdir('dataset_resized/left');
mkdir('dataset_resized/right');
mkdir('dataset_resized/palm');
mkdir('dataset_resized/peace');
label_keys = { 'left', 'right', 'palm', 'peace'};
X_train = [];
y_train = [];
X_test = [];
y_test = [];

% Read all the dataset images
Files=dir('dataset/*/*.jpg');
for k=1:length(Files)
```

```
FileNames = Files(k).name;
dr = Files(k).folder;
fileLocation = strcat(dr, '\', FileNames);

% Extract folder name
path = strsplit(dr, '\');
folder = path{length(path)};
% Create the image label depending on the folder name
% Ex. [1 0 0 0] - left folder
label = ismember(label_keys, folder);
% Extract file name
filename = strsplit(FileNames, '.');
name = filename{1};
extension = filename{2};

% Process image by skin color
% Returns 50x50 image
image_out = processSkinImage(fileLocation);
% Generate random number from 1 to 10
randNum = ceil(rand() * 10);
% Split the images in 80%-20% train-test set
if randNum > 2
   % Create the features for the image
   X_train = [X_train; image_out(:)'];
```

```
    y_train = [y_train; label];
  else
    X_test = [X_test; image_out(:)'];
    y_test = [y_test; label];
  endif

  % Write the proceessed image in dataset_resized folder
  output_folder = strcat('dataset_resized', '/', folder, '/', name, '_resized.',
extension);
  imwrite(image_out, output_folder);
end

% Write the train features and labels in file
dlmwrite('x_features_train', X_train);
dlmwrite('y_labels_train', y_train);
% Write the test features and labels in file
dlmwrite('x_features_test', X_test);
dlmwrite('y_labels_test', y_test);
```

fmincg

```
function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
% Minimize a continuous differentialble multivariate function. Starting point
% is given by "X" (D by 1), and the function named in the string "f", must
```

```
% return a function value and a vector of partial derivatives. The Polack-
% Ribiere flavour of conjugate gradients is used to compute search directions,
% and a line search using quadratic and cubic polynomial approximations and the
% Wolfe-Powell stopping criteria is used together with the slope ratio method
% for guessing initial step sizes. Additionally a bunch of checks are made to
% make sure that exploration is taking place and that extrapolation will not
% be unboundedly large. The "length" gives the length of the run: if it is
% positive, it gives the maximum number of line searches, if negative its
% absolute gives the maximum allowed number of function evaluations. You can
% (optionally) give "length" a second component, which will indicate the
% reduction in function value to be expected in the first line-search (defaults
% to 1.0). The function returns when either its length is up, or if no further
% progress can be made (ie, we are at a minimum, or so close that due to
% numerical problems, we cannot get any closer). If the function terminates
% within a few iterations, it could be an indication that the function value
% and derivatives are not consistent (ie, there may be a bug in the
% implementation of your "f" function). The function returns the found
% solution "X", a vector of function values "fX" indicating the progress made
% and "i" the number of iterations (line searches or function evaluations,
% depending on the sign of "length") used.
%
% Usage: [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
%
if exist('options', 'var') && ~isempty(options) && isfield(options, 'MaxIter')
```

```
   length = options.MaxIter;
else
   length = 100;
end

RHO = 0.01;                    % a bunch of constants for line searches
SIG = 0.5;      % RHO and SIG are the constants in the Wolfe-Powell conditions
INT = 0.1;    % don't reevaluate within 0.1 of the limit of the current bracket
EXT = 3.0;                % extrapolate maximum 3 times the current bracket
MAX = 20;                 % max 20 function evaluations per line search
RATIO = 100;                          % maximum allowed slope ratio
argstr = ['feval(f, X'];              % compose string used to call function
for i = 1:(nargin - 3)
  argstr = [argstr, ',P', int2str(i)];
end

argstr = [argstr, ')'];
if max(size(length)) == 2, red=length(2); length=length(1); else red=1; end
S=['Iteration '];
i = 0;                                % zero the run length counter
ls_failed = 0;                        % no previous line search has failed
fX = [];
[f1 df1] = eval(argstr);              % get function value and gradient
i = i + (length<0);                                % count epochs?!
```

```
s = -df1;                           % search direction is steepest
d1 = -s'*s;                         % this is the slope
z1 = red/(1-d1);                    % initial step is red/(|s|+1)
while i < abs(length)               % while not finished
  i = i + (length>0);               % count iterations?!

  X0 = X; f0 = f1; df0 = df1;       % make a copy of current values
  X = X + z1*s;                     % begin line search
  [f2 df2] = eval(argstr);
  i = i + (length<0);               % count epochs?!
  d2 = df2'*s;

  f3 = f1; d3 = d1; z3 = -z1;       % initialize point 3 equal to point 1
  if length>0, M = MAX; else M = min(MAX, -length-i); end
  success = 0; limit = -1;          % initialize quanteties
  while 1
    while ((f2 > f1+z1*RHO*d1) || (d2 > -SIG*d1)) && (M > 0)
      limit = z1;                   % tighten the bracket
      if f2 > f1
        z2 = z3 - (0.5*d3*z3*z3)/(d3*z3+f2-f3);     % quadratic fit
      else
        A = 6*(f2-f3)/z3+3*(d2+d3);                 % cubic fit
        B = 3*(f3-f2)-z3*(d3+2*d2);
        z2 = (sqrt(B*B-A*d2*z3*z3)-B)/A;    % numerical error possible - ok!
```

```
  end
  if isnan(z2) || isinf(z2)
    z2 = z3/2;                % if we had a numerical problem then bisect
  end
  z2 = max(min(z2, INT*z3),(1-INT)*z3);  % don't accept too close to limits
  z1 = z1 + z2;                           % update the step
  X = X + z2*s;
  [f2 df2] = eval(argstr);
  M = M - 1; i = i + (length<0);                % count epochs?!
  d2 = df2'*s;
  z3 = z3-z2;                % z3 is now relative to the location of z2
end
if f2 > f1+z1*RHO*d1 || d2 > -SIG*d1
  break;                                  % this is a failure
elseif d2 > SIG*d1
  success = 1; break;                                % success
elseif M == 0
  break;                                          % failure
end

A = 6*(f2-f3)/z3+3*(d2+d3);                % make cubic extrapolation
B = 3*(f3-f2)-z3*(d3+2*d2);
z2 = -d2*z3*z3/(B+sqrt(B*B-A*d2*z3*z3));       % num. error possible - ok!
if ~isreal(z2) || isnan(z2) || isinf(z2) || z2 < 0 % num prob or wrong sign?
```

```
  if limit < -0.5                    % if we have no upper limit
    z2 = z1 * (EXT-1);               % the extrapolate the maximum amount
  else
    z2 = (limit-z1)/2;                        % otherwise bisect
  end
 elseif (limit > -0.5) && (z2+z1 > limit)        % extraplation beyond max?
  z2 = (limit-z1)/2;                           % bisect
 elseif (limit < -0.5) && (z2+z1 > z1*EXT)       % extrapolation beyond limit
  z2 = z1*(EXT-1.0);                  % set to extrapolation limit
 elseif z2 < -z3*INT
  z2 = -z3*INT;
 elseif (limit > -0.5) && (z2 < (limit-z1)*(1.0-INT))  % too close to limit?
  z2 = (limit-z1)*(1.0-INT);
 end

 f3 = f2; d3 = d2; z3 = -z2;              % set point 3 equal to point 2
 z1 = z1 + z2; X = X + z2*s;                % update current estimates
 [f2 df2] = eval(argstr);
 M = M - 1; i = i + (length<0);                    % count epochs?!
 d2 = df2'*s;
end                                      % end of line search
if success                               % if line search succeeded
 f1 = f2; fX = [fX' f1]';
 fprintf('%s %4i | Cost: %4.6e\r', S, i, f1);
```

```
  s = (df2'*df2-df1'*df2)/(df1'*df1)*s - df2;      % Polack-Ribiere direction
  tmp = df1; df1 = df2; df2 = tmp;                         % swap derivatives
  d2 = df1'*s;
  if d2 > 0                                  % new slope must be negative
    s = -df1;                          % otherwise use steepest direction
    d2 = -s'*s;
  end

  z1 = z1 * min(RATIO, d1/(d2-realmin));         % slope ratio but max RATIO
  d1 = d2;
  ls_failed = 0;                          % this line search did not fail
else
  X = X0; f1 = f0; df1 = df0;  % restore point from before failed line search
  if ls_failed || i > abs(length)         % line search failed twice in a row
    break;                          % or we ran out of time, so we give up
  end
  tmp = df1; df1 = df2; df2 = tmp;                         % swap derivatives
  s = -df1;                                          % try steepest
  d1 = -s'*s;
  z1 = 1/(1-d1);
  ls_failed = 1;                             % this line search failed
end

if exist('OCTAVE_VERSION')
```

```
   fflush(stdout);
  end
end
fprintf('\n');
```

nncostfunction

```
function [J grad] = nnCostFunction(nn_params, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, ...
                                   X, y, lambda)

%NNCOSTFUNCTION Implements the neural network cost function for a two layer
%neural network which performs classification
%   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
%   X, y, lambda) computes the cost and gradient of the neural network. The
%   parameters for the neural network are "unrolled" into the vector
%   nn_params and need to be converted back into the weight matrices.
%
%   The returned parameter grad should be a "unrolled" vector of the
%   partial derivatives of the neural network.
%
```

```
% Reshape nn_params back into the parameters Theta1 and Theta2, the weight matrices
% for our 2 layer neural network
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));

% Setup some useful variables
m = size(X, 1);
        % You need to return the following variables correctly
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));

% Feedforward the neural network and return the cost in the variable J
summary = 0;
for j = 1:m
   y_label = y(j,:);
   a_one = X(j,:);
   a_one = [1 a_one];
   z_one = a_one * Theta1';
   a_two = sigmoid(z_one);
```

```
    m_temp = size(a_two, 1);
    a_two = [ones(m_temp, 1) a_two];
    z_two = a_two * Theta2';
    a_three = sigmoid(z_two);
    h = a_three;
    sum_temp = 0;
    sum_temp = (-y_label .* log(h)) - ((1 - y_label) .* log(1 - h));
    summary = summary + sum(sum_temp);

% Implement Backpropagation algorithm
     d3 = (a_three .- y_label);
    z_one = [1 z_one];
    d2 = (d3 * Theta2) .* sigmoidGradient(z_one);
    d2 = d2(:, 2:end);

    Theta1_grad = Theta1_grad + d2' * a_one;
    Theta2_grad = Theta2_grad + d3' * a_two;
end;

% Add regularization to the cost
regularization = (lambda/(2*m)) * (sum(sum(Theta1(:,2:end) .^ 2, 2)) +
sum(sum(Theta2(:,2:end) .^ 2, 2)));
J = (summary/m) + regularization;

% Add regularization to the gradient
```

```
Theta1_grad(:,1) = Theta1_grad(:,1) ./ m;

Theta2_grad(:,1) = Theta2_grad(:,1) ./ m;

Theta1_grad(:,2:end) = (Theta1_grad(:,2:end) ./ m) + ((lambda ./ m) *
Theta1(:,2:end));

Theta2_grad(:,2:end) = (Theta2_grad(:,2:end) ./ m) + ((lambda ./ m) *
Theta2(:,2:end));

% -------------------------------------------------------------

%=========================================================================
========

% Unroll gradients

grad = [Theta1_grad(:) ; Theta2_grad(:)];

end
```

predict

```
function p = predict(Theta1, Theta2, X)

%PREDICT Predict the label of an input given a trained neural network

%   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X given the

%   trained weights of a neural network (Theta1, Theta2)

% Initial values

m = size(X, 1);

num_labels = size(Theta2, 1);

p = zeros(size(X, 1), 1);

h1 = sigmoid([ones(m, 1) X] * Theta1');

h2 = sigmoid([ones(m, 1) h1] * Theta2');
```

```
[value, p] = max(h2, [], 2);
end
```

processSkinImage

```
function image_out = processSkinImage(filename)
%PROCESSSKINIMAGE Segment hand from image
%   image_out = PROCESSSKINIMAGE(filename)

%   Segment hand by skin color from given image
    % Load image package
    pkg load image;
    % Initialize
    numrows = 50;
    numcols = 50;
    scale = [numrows numcols];
    % Read the image
    original = imread(filename);
    [M N Z] = size(original);
    % Read the image, and capture the dimensions
    height = size(original,1);
    width = size(original,2);
    % Resize the image to 50x50
    image_resized = imresize(original, scale);
```

```
[M N Z] = size(image_resized);

% Initialize the output image
image_out = zeros(M,N);

% Convert the image from RGB to YCbCr
img_ycbcr = rgb2ycbcr(image_resized);
Cb = img_ycbcr(:,:,2);
Cr = img_ycbcr(:,:,3);

% Get the central color of the image
% Expected the hand to be in the central of the image
central_color = img_ycbcr(int32(M/2),int32(N/2),:);
Cb_Color = central_color(:,:,2);
Cr_Color = central_color(:,:,3);
% Set the range
Cb_Difference = 15;
Cr_Difference = 10;

% Detect skin pixels
[r,c,v] = find(Cb>=Cb_Color-Cr_Difference & Cb<=Cb_Color+Cb_Difference &
Cr>=Cr_Color-Cr_Difference & Cr<=Cr_Color+Cr_Difference);
match_count = size(r,1);

% Mark detected pixels
```

```
   for i=1:match_count
      image_out(r(i),c(i)) = 1;
   end
   %imshow(image_out);
end
```

randInitializeWeights

```
function W = randInitializeWeights(L_in, L_out)
%RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with L_in
%incoming connections and L_out outgoing connections
% Initialize W randomly so that we break the symmetry while
%               training the neural network.

% Note that W should be set to a matrix of size(L_out, 1 + L_in) as
% the first column of W handles the "bias" terms
epsilon = sqrt(6) / (L_in + L_out);
W = zeros(L_out, 1 + L_in);
W = (rand(L_out, 1 + L_in) * 2 * epsilon) - epsilon;
end
```

sigmoid

```
function g = sigmoid(z)
%SIGMOID Compute sigmoid functoon
%   J = SIGMOID(z) computes the sigmoid of z.
g = 1.0 ./ (1.0 + exp(-z));
end
```

sigmoid gradient

```
function g = sigmoidGradient(z)
%SIGMOIDGRADIENT returns the gradient of the sigmoid function
% Initial values
g = zeros(size(z));
% Compute gradient of the sigmoid
g = sigmoid(z) .* (1 - sigmoid(z));
end
```

main

```
%% Setup the parameters you will use for this exercise
input_layer_size  = 2500;  % 50x50 Input Images of Digits
hidden_layer_size = 25;   % 25 hidden units
num_labels = 4;          % 4 labels
```

```
% Proccessed image features with 2500 columns for each row
% since there are 2500 pixels (50x50) from every processed image
X_train = dlmread('x_features_train');
X_test = dlmread('x_features_test');

% Labels for each processed training and test image
%[1 0 0 0] - left, [0 1 0 0] - right, [0 0 1 0] - palm, [0 0 0 1] - peace
y_train = dlmread('y_labels_train');
y_test = dlmread('y_labels_test');

label_keys = { 'left', 'right', 'palm', 'peace'};

% Initialize random weights for start
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);

% Unroll parameters
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];

options = optimset('MaxIter', 100);
lambda = 1;

% Create the cost function that needs to be minimized
```

```
costFunction = @(p) nnCostFunction(p, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, X_train, y_train, lambda);

% Now, costFunction is a function that takes in only one argument (the
% neural network parameters)
[nn_params, cost] = fmincg(costFunction, initial_nn_params, options);

% Obtain Theta1 and Theta2 back from nn_params
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size +
1))):end), ...
                 num_labels, (hidden_layer_size + 1));

% Make the prediction based on obtained Theta values
pred = predict(Theta1, Theta2, X_train);

% Compare the prediction with the actual values
[val idx] = max(y_train, [], 2);
fprintf('\nTraining Set Accuracy: %f%%\n', mean(double(pred == idx)) * 100);
```

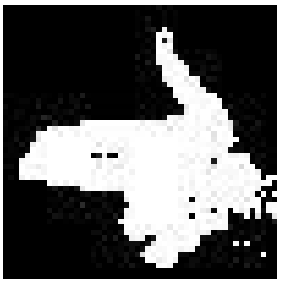
```
% Make the prediction based on obtained Theta values
pred = predict(Theta1, Theta2, X_test);

% Compare the prediction with the actual values
[val idx] = max(y_test, [], 2);
fprintf('\nTest Set Accuracy: %f%%\n', mean(double(pred == idx)) * 100);

test_img = processSkinImage("test/test1.jpg");
imshow(test_img);
pred = predict(Theta1, Theta2, test_img(:)')
fprintf('\nType: %s\n', label_keys{pred});
pause;

% Predict multiple test images
test_img = processSkinImage("test/test2.jpg");
imshow(test_img);
pred = predict(Theta1, Theta2, test_img(:)')
fprintf('\nType: %s\n', label_keys{pred});
pause;
test_img = processSkinImage("test/test3.jpg");
imshow(test_img);
pred = predict(Theta1, Theta2, test_img(:)')
fprintf('\nType: %s\n', label_keys{pred});
pause;
```

```
test_img = processSkinImage("test/test4.jpg");
imshow(test_img);
pred = predict(Theta1, Theta2, test_img(:)')
fprintf('\nType: %s\n', label_keys{pred});
```

PROCESSED IMAGES

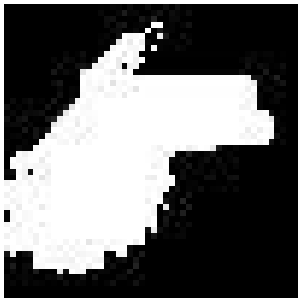
ORIGINAL IMAGE (LEFT)

PROCESSED IMAGE



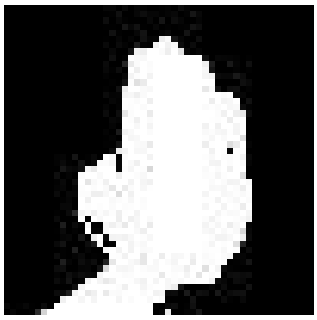
ORIGINAL IMAGE(RIGHT)

PROCESSED IMAGE



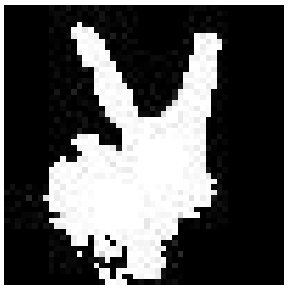
ORIGINAL IMAGE(PALM)

PROCESSED IMAGE



ORIGINAL IMAGE(PEACE)

PROCESSED IMAGE



OUTPUT

```
GNU Octave, version 5.1.0
Copyright (C) 2019 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-w64-mingw32".

Additional information about Octave is available at https://www.octave.org

Please contribute if you find this software useful.
For more information, visit https://www.octave.org/get-involved.html

Read https://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

>> main

Iteration   100 | Cost: 3.627151e-01

Training Set Accuracy: 100.000000%

Test Set Accuracy: 90.909091%

pred =  2
Type: right

pred =  1
Type: left

pred =  3
Type: palm

pred =  3
Type: palm
>>
```

METHODOLOGY

In the function main() following function blocks are included –

randInitializeWeights() - Randomly initialize the weights of a layer with L_in incoming connections and L_out outgoing connections. Initialize W randomly so that we break the symmetry while training the neural network. Note that W should be set to a matrix of size(L_out, 1 + L_in) as the first column of W handles the "bias" terms.

nncostfunction() - Implements the neural network cost function for a two layer neural network which performs classification. The cost function determines how good our neural network. It is determined by calculating the square of error. When the two consecutive cost functions are equal, the BPN algorithm can be stopped.

[J grad] = nncostfunction(nn_params, hidden_layer_size, num_labels, X, Y, lambda) computes the cost and gradient of the neural network. The parameters for the neural network are "unrolled" into the vector nn_params and need to be converted back into the weight matrices. The returned parameter grad should be an "unrolled" vector of the partial derivatives of the neural network.

Reshape nn_params back into the parameters Theta1 and Theta2, the weight matrices for our 2 layer neural network. The network is further corrected by adding regularisation term.

fmincg() - Minimize a continuous differentiable multivariate function. Starting point is given by "X" (D by 1), and the function named in the string "f", must return a function value and a vector of partial derivatives.

predict() - Predict the label of an input given a trained neural network p = predict(Theta1, Theta2, X) outputs the predicted label of X given the trained weights of a neural network (Theta1, Theta2). processskinimage() - The image is converted from RGB to YcbCr image and resized into 50*50 image. The skin pixels are detected by picking up the pixel values of central pixel of palm and the pixels which lie between certain range determined by the skin pixel value, the pixel is assigned value, else black.

In the program following parameters are defined as follows –

x_train –

Contains pixel values of the training images after processing the 80% images selected for training.

y_train –

Contains label values of the training images :

[1 0 0 0] – left.

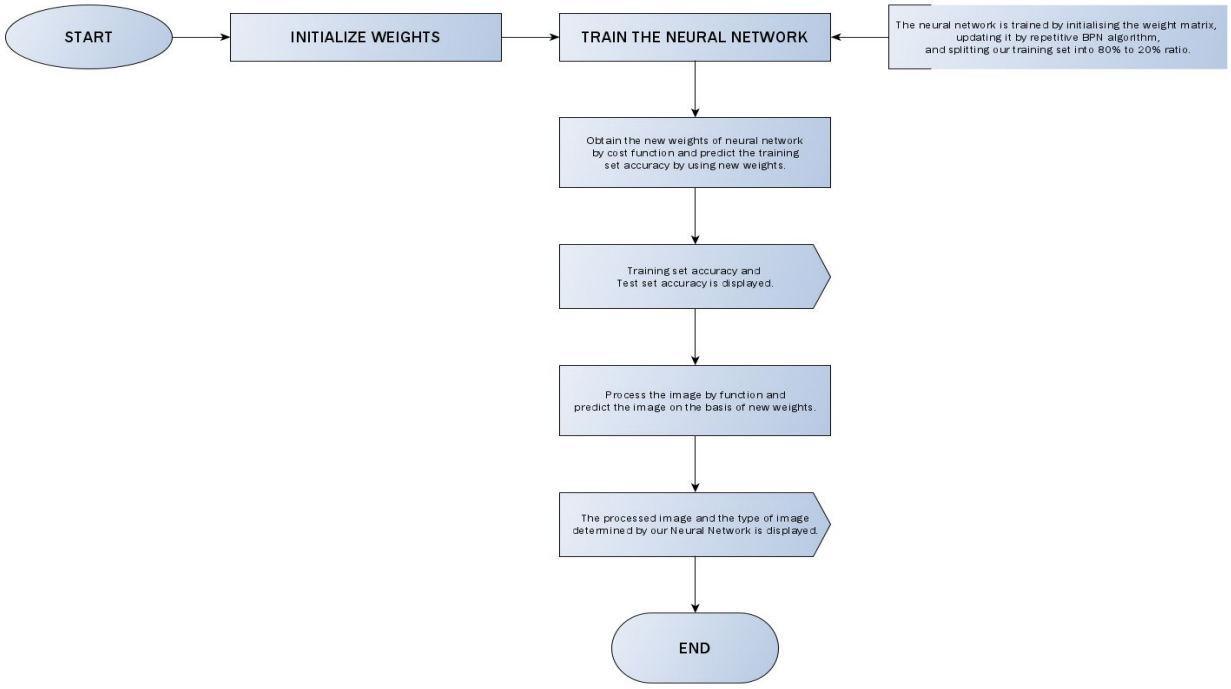
[0 1 0 0] – right.

[0 0 1 0] – palm.

[0 0 0 1] – peace.

x_test and **y_test** contains the values of pixels and labels respectively after testing the network.

WORKING


START
INITIALIZE WEIGHTS
TRAIN THE NEURAL NETWORK
The neural network is trained by initialising the weight matrix, updating it by repetitive BPN algorithm, and splitting our training set into 80% to 20% ratio.
Obtain the new weights of neural network by cost function and predict the training set accuracy by using new weights.
Training set accuracy and Test set accuracy is displayed.
Process the image by function and predict the image on the basis of new weights.
The processed image and the type of image determined by our Neural Network is displayed.
END

PROCEDURE

The detailed procedure is as follows -

The weights are initialized by randInitializeWeights().

The cost function which determines the new weights using nncostfunction().

The cost function is further corrected by fmincg() which minimises the cost function making our cost

function further fast to determine the weights more accurately. Such is done by calculating minimum

value of cost function by the means of partial derivatives.

The image is predicted as soon as the neural network is trained according to the train and test image

dataset.

The type of image predicted is shown along with the processed skin image.

CONCLUSION

Thus, we have successfully trained a neural network to recognize four different types of hand gestures and this project can be, hence, used as a building block to develop more complicated neural networks.

REFERENCES

- https://onlinelibrary.wiley.com/doi/abs/10.1002/scj.4690230304
- https://towardsdatascience.com/training-a-neural-network-to-detect-gestures-with-o pencv-in-python-e09b0a12bdf1