Optical Character Recognition using Hopfield Neural Networks

By: Pradyumna Mukunda
Vishal Nagarajan

Abstract

The purpose of the project is to recognize printed English characters (capital alphabets, small alphabets, and numerals) using Hopfield networks. The code should be able to recognize individual characters, recognize words, and recognize lines of characters with words separated by whitespaces.

Problem Statement:

Design a project in GNU Octave to implement Optical Character Recognition of printed English characters using Hopfield networks

Introduction

Optical character Recognition

Optical character recognition (also optical character reader, OCR) is the mechanical or electronic conversion of images of typed, handwritten or printed text into machine-encoded text, whether from a scanned document, a photo of a document, a scene-photo (for example the text on signs and billboards in a landscape photo) or from subtitle text superimposed on an image (for example from a television broadcast). It is widely used as a form of information entry from printed paper data records, whether passport documents, invoices, bank statements, computerised receipts, business cards, mail, printouts of static-data, or any suitable documentation. It is a common method of digitising printed texts so that they can be electronically edited, searched, stored more compactly, displayed on-line, and used in machine processes such as cognitive computing, machine translation, (extracted) text-to-speech, key data and text mining. OCR is a field of research in pattern recognition, artificial intelligence and computer vision.

In our project we are going to implement OCR using a neural network known as Hopfield network.

Hopfield network

A Hopfield network is a form of recurrent artificial neural network popularized by John Hopfield in 1982. Hopfield nets serve as content-addressable memory systems with binary threshold nodes. Content-addressable memory (CAM) is a special type of computer memory used in certain very-high-speed searching applications. It is also known as associative memory. It compares input search data (tag) against a table of stored data, and returns the address of matching data (or in the case of associative memory, the matching data itself).

Hopfield networks are associative memories where the stored data are $n \times 1$ vectors called patterns whose elements can take the values 1 and -1. Let there be m such patterns, namely $v_1, v_2, v_3 \dots v_m$. A Hopfield Network may be represented in the form of a $n \times n$ matrix called Hopfield matrix, denoted by W. The formula to create a Hopfield matrix that stores the patterns $v_1, v_2, v_3 \dots v_m$ is:

$$W = \sum_{k=1}^{m} v_k {v_k}^T - mI$$

where I is the identity matrix. Afterwards, given a random input pattern v_{in} we apply the Hopfield network using the formula:

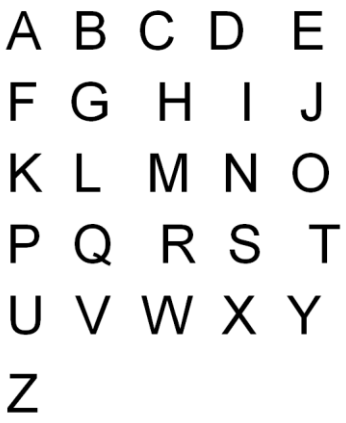
$$v_{out} = \text{signum}\left(\frac{1}{n}(W \times v_{in})\right)$$

The output v_{out} will be the stored pattern that v_{in} resembles the most. Our Hopfield network has the capability of correcting noisy inputs which makes it useful in OCR to recognize characters from many different fonts. However, sometimes v_{out} will be a false pattern that was never stored in the network, and this is a drawback of Hopfield networks.

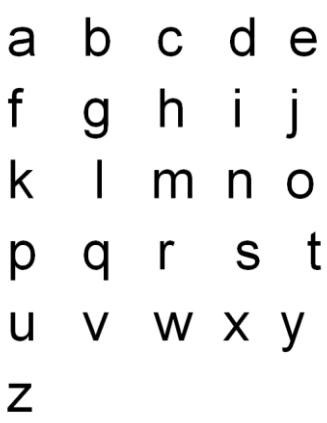
Implementation

Step 1: Creating Samples

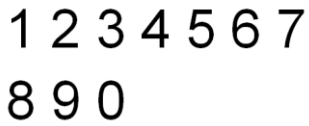
Our first step to be able to recognize characters is to create samples of all of the characters, which we will be using as stored data patterns in the Hopfield networks. We used Arial font for the samples as it is a very generic and widely used font. The following snapshots taken from Microsoft Word were used for extracting character samples. The project was implemented in GNU Octave which is a scripting language very similar to MATLAB.



Cap_Arial.PNG



Small_Arial.PNG



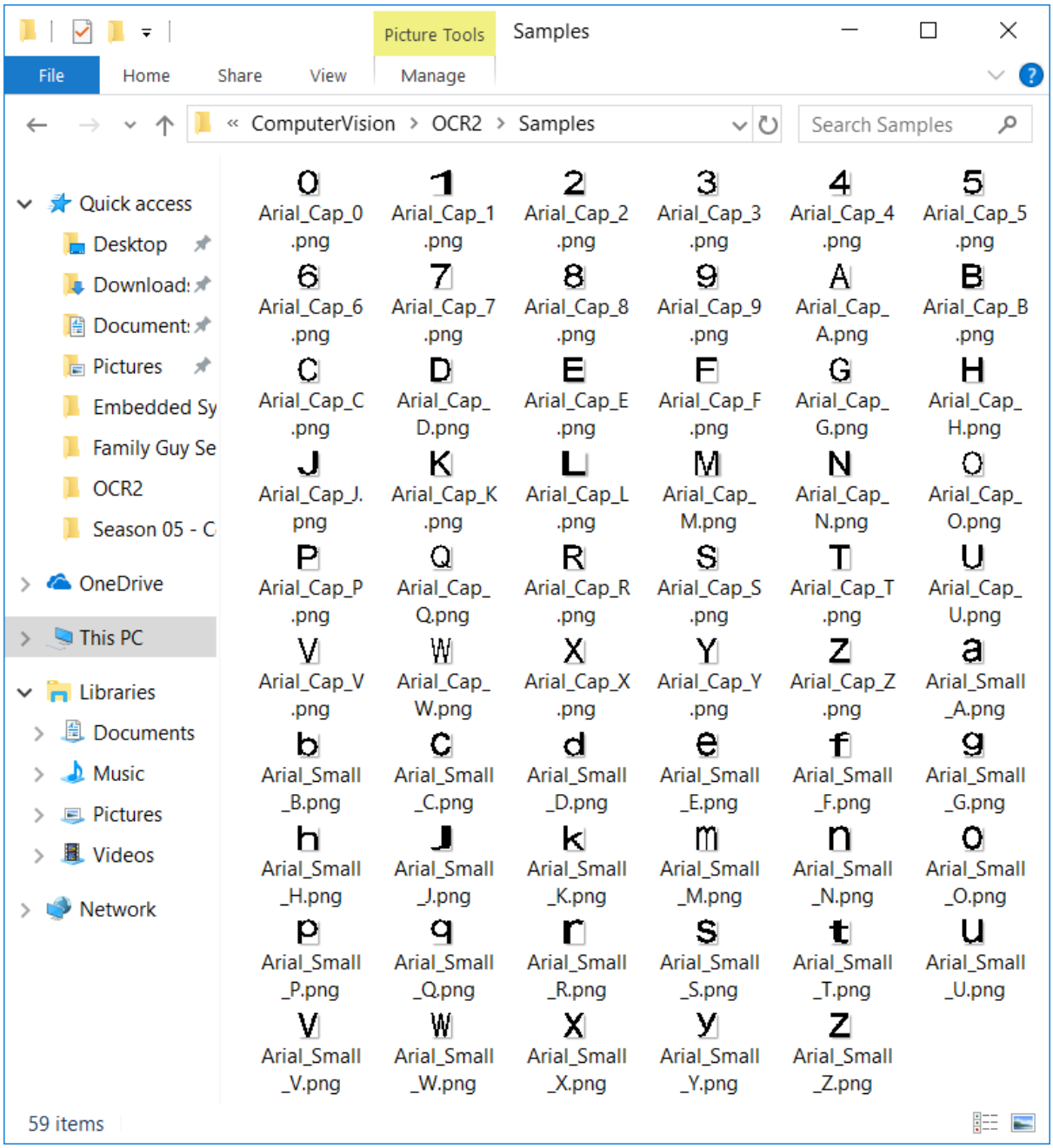
Num_Arial.PNG

Octave Script: “Create_Samples.m”

```
pkg load image
clc
clear all
close all

A = imread('Cap_Arial.png');        % Changed for each image
B = rgb2gray(A);
B = B <= 128;
name = 'Arial_Cap_'
for i = 1:26
  h = randi(size(B,1));
  w = randi(size(B,2));
  while (B(h,w) == 0)
    h = randi(size(B,1));
    w = randi(size(B,2));
  end
  C = bwselect(B,w,h,8);
  B = 1-bwfill(1-B,w,h,8);
  [H,W] = find(C);
  D = C(min(H):max(H),min(W):max(W));
  D = imresize(D, [18 15], 'linear');
  D = 1 - D;
  imwrite(D,strcat(name,num2str(i),'.png'))
end
```

Description

The input image is converted to grayscale and then into a binary image with threshold of 128. Pixel values less than 128 (black) become 1s and values greater than 128 (white) become 0s. Now the code selects a random point in the image. If the point is a 0 (miss) then it selects again. If the point is a 1 (hit), then the code performs region growing around that point and then crops the character out of the image. The new image is resized to a standard size of 18x15 and then exported. The text colour is black and background colour is white Here is the result. The files were renamed manually.



Step 2: Creating Hopfield networks

Now we are going to use these samples as our stored data in our Hopfield networks.

Octave Function: “Create_Hop_Arial.m”

```
function [W,Vect] = Create_Hop_Arial(chars)
m = length(chars);
name = 'Samples/Arial_';
for j = 1:m
  if (chars(j) >= 'a')
    ccase = 'Small_';
  else
    ccase = 'Cap_';
  end
  Img = imread([name,ccase,toupper(chars(j)),'.png']);
  Img = 2*(Img >= 32768)-1;
  Vect(:,j) = reshape(Img',270,1);
  Mat = Vect(:,j) * Vect(:,j)';
  if (j == 1)
    Sum = Mat;
  else
    Sum = Sum + Mat;
  endif
endfor
I = eye(size(Mat));
W = (Sum-m*I);
endfunction
```

Description

The user passes an input `chars` to this function, which is a variable length string that specifies the characters for which to create a Hopfield network. The code reads the selected samples from the Samples folder which are specified by the `chars` input. Remember that the image is black on white text. Each sample is converted to a binary image where white is represented as 1 and black is represented as -1. The 18×15 image is then reshaped into a 270×1 vector in row-major order. Remember that a Hopfield network operates on n×1 vectors whose elements take values 1 and -1. So here n is 270. The Hopfield network is created using the equation:

$$W = \sum_{k=1}^{m} v_k {v_k}^T - mI$$

The Hopfield matrix W is returned as the output of the function. The function also returns a matrix `Vect` which is all of the stored patterns concatenated horizontally.

Octave Script: “Init_Setup_Arial.m”

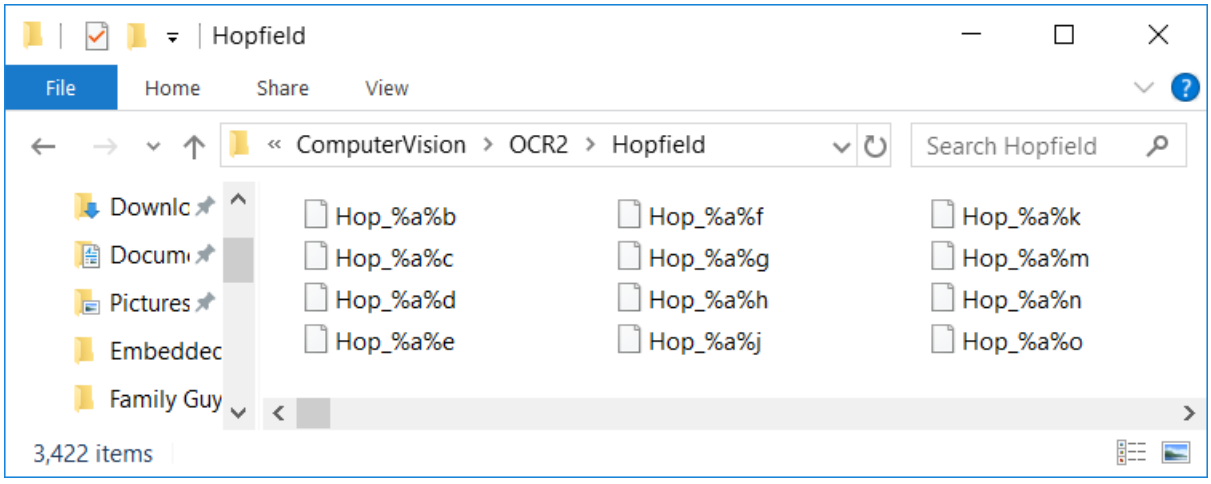
```
time0 = time;
pkg load image
chars = 'abcdefghjkmnopqrstuvwxyzABCDEFGHJKLMNOPQRSTUVWXYZ0123456789';
```

```
m = length(chars);
for i = 1:m
  for j = i+1:m
    [W,Vect] = Create_Hop_Arial([chars(i) chars(j)]);
    if (chars(i) >= 'a')
    str1 = ['%' chars(i)];
    else
    str1 = chars(i);
    end
    if (chars(j) >= 'a')
    str2 = ['%' chars(j)];
    else
    str2 = chars(j);
    end
    name = ['Hopfield/Hop_' str1 str2];
    name2 = ['Hopfield/Vect_' str1 str2];
    fid = fopen(name,'w');
    val = fwrite(fid,W,'int8');
    fclose(fid);
    fid = fopen(name2,'w');
    fwrite(fid,Vect,'int8');
    fclose(fid);
  endfor
endfor
disp(time-time0)
```

Description:

The code creates a Hopfield network for every combination of two characters from the character set `chars` i.e. ab, ac, ad … a9, then bc, bd, … b9 etc. Although a Hopfield network can be created with more than two characters it was limited to two for accuracy. The code uses the `Create_Hop_Arial` function as defined before. It then exports the matrices `W` and `Vect` into two files (without file extension). This is done for all combinations of characters.

The resultant size of all the Hopfield matrices created is 120MB. The code takes around 1-2 minutes to execute completely and the result is shown below. Note that the codes “Create_Samples.m” and “Init_Setup_Arial.m” for steps 1 and 2 need to be executed only once just to setup the project and they will not be used further on.



Step 3: Applying Hopfield network

After we have created and exported all possible Hopfield matrices we now have to write a code to apply them.

Octave Function: “Hop_Test.m”

```
function ch = Hop_Test(IN_img,chars)

% Add % sign for lowercase
if (chars(1) >= 'a')
str1 = ['%' chars(1)];
else
str1 = chars(1);
end
if (chars(2) >= 'a')
str2 = ['%' chars(2)];
else
str2 = chars(2);
end

% Open Hopfield Matrices and vectors stored in files
name = ['Hopfield/' 'Hop_' str1 str2];
fid = fopen(name,'r');
W = fread(fid,[270 270],'int8');
fclose(fid);
name2 = ['Hopfield/' 'Vect_' str1 str2];
fid = fopen(name2,'r');
Vect = fread(fid,[270 2],'int8');
fclose(fid);

% Apply Hopfield network on image
IN_vect = reshape(IN_img',270,1);
IN_vect = sign(double(W)/270*IN_vect);

% Compare with standard vectors
ch = '.';
for i = 1:length(chars)
  if(isequal(IN_vect,Vect(:,i)))
    ch = chars(i);
    break;
  endif
endfor

endfunction
```

Description:

The user passes an 18x15 binary image `IN_img` as an input. The image is reshaped into a 270x1 vector `IN_vect`. The user also passes an input `chars` to this function, which is a string of two characters. The code imports the corresponding `W` and `Vect` matrices which were created earlier. It then applies the Hopfield matrix `W` on the vector `IN_vect` using the formula:

$$v_{out} = \text{signum}\left(\frac{1}{n}(W \times v_{in})\right)$$

Here the input v_{in} is `IN_vect` and the output v_{out} is stored back in `IN_vect` itself. After applying the Hopfield network `IN_vect` will be equal to one of the two stored vectors in the Hopfield network and the function returns the character ch corresponding to the matching stored vector. In case `IN_vect` is not equal to either of the stored vectors then the function returns a period (.), indicating that the Hopfield network returned a false pattern.

Step 4: Recognizing isolated characters

This code will enable us to recognize isolated characters.

Octave function: “Char_Test.m”

```
function ch = Char_Test(IN_img,charset)

% Define character set
charset = toupper(charset);
if (isequal(charset,'FULL'))
  chars = 'abcdefghjkmnopqrstuvwxyzABDEFGHJKLMNPQRTUY0123456789';
end
if (isequal(charset,'ALPHA'))
  chars = 'abcdefghjkmnopqrstuvwxyzABDEFGHJKLMNPQRTUY';
end
if (isequal(charset,'CAPONLY'))
  chars = 'ABCDEFGHJKLMNOPQRSTUVWXYZ0123456789';
end
if (isequal(charset,'CAPALPHA'))
  chars = 'ABCDEFGHJKLMNOPQRSTUVWXYZ';
end

% Cycle through all permutation
i = 1;
j = 2;
while(true)
  if(j > length(chars))
    break;
  end
  ch = Hop_Test(IN_img,[chars(i),chars(j)]);
  if(ch == chars(j))
    i = j;
  end
  j = j+1;
endwhile

endfunction
```

Description:

The code applies the function `Hop_Test` on the 18x15 binary image `IN_img` for various combinations of two characters from the character set `charset`. The output `ch` is the recognized character.

The user can choose from four different character sets as follows:

1. All characters (FULL)
2. Only alphabets (ALPHA)
3. Only capital alphabets + numerals (CAPONLY)
4. Only capital alphabets (CAPALPHA)

This is done for the purpose of speed. The code takes longer to execute for larger character sets, and to use FULL character set to recognize a text with, say, only capital alphabets, would waste time.

The method of cycling through the characters is as follows. Let us say our `IN_img` is of the character ‘G’ and we are using the “CAPALPHA” character set. We start by testing `IN_img` on the “AB” Hopfield network, which will return ‘A’ if `IN_img` resembles A more than B and ‘B’ if vice versa. In this case the network returns ‘A’. Next we test it on the “AC” Hopfield network. Here it returns ‘C’, since obviously G resembles C more than A. Now we won’t go testing `IN_img` on the “BC” Hopfield network, since we know that G resembles A more than B and resembles C more than A, therefore obviously it resembles C more than B. Instead we test it on the “CD” network. The “CD”, “CE, “CF” networks all return ‘C’ as an output. The “CG” network returns ‘G’. Again, we won’t test it on “DG”, “EG”, and “FG” networks since we already know the result will be ‘G’. Hence next we test it on the “GH” network. The networks “GH” to “GZ” all return ‘G’, therefore we conclude that `IN_img` resembles G more than any other character and we return ‘G’ as the output.

Step 5: Recognizing words

We are able to successfully recognize characters. Now the next challenge is to recognize entire words. For this we have to extract each character from the word and recognize them separately, all while not disrupting the order of the characters in the word.

Octave Script: “Word_Recognition_demo.m”

```
clc
clear all
close all
time0 = time;

pkg load image

charset = 'ALPHA';
text = '';

A = imread('logo.png');
if (isrgb(A))
  B = rgb2gray(A);
else
  B = A;
end
C = (B <= 200);
X = strel('disk',5,0);
```

```
%C = imerode(C,X);

height = size(C,1);
width = size(C,2);

% Least square fitting of horizontal line on image
H = (1:height)'*ones(1,height);
dH = (H-H').^2;
S = dH*C*ones(width,1);
[Smin,h] = min(S);
B(h,:) = 0;
%imshow(B)

% Isolate and recognize characters
for w = 1:width
  if (C(h,w) == 1)
  D = bwselect(C,w,h,4);
  C = 1-bwfill(1-C,w,h,4);
  [H2,W2] = find(D);
  E = D(min(H2):max(H2),min(W2):max(W2));
  E = imresize(E, [18 15], 'linear');
  E = -2*E+1;
  if(mean(mean(E))<-0.8)
    if(isempty(text))
      ch = 'I';
    else
      ch = 'l';
    end
  else
    ch = Char_Test(E,charset);
  end
  text = [text ch];
  endif
endfor
clc
disp(text);
disp(time - time0);
```

Description:

We are going to demonstrate this algorithm by recognizing a popular logo.

FIGURE 1: INPUT IMAGE

The input image is converted to grayscale and then into a binary image with threshold of 200. Pixel values less than 200 (black) become 1s and values greater than 200 (white) become 0s.

FIGURE 2: AFTER GRAYSCALING

FIGURE 3: AFTER THRESHOLDING

Next we use region growing to extract characters, similar to what we did in step 1 to create the samples. In step 1 the seeds were chosen randomly. However we cannot do that here as we need to preserve the order of the characters. The logical solution to this would be to scan the entire image from left to right and top to bottom, but that would waste time unnecessarily. Instead we least-square fit a horizontal line on the image, in a process similar to least-square fitting of a straight line on a set of data points. Then we can scan only that line. For representation purpose we have shown the least square fitted line in red colour, but this is not done in the actual code. The goal is to only determine the height at which to scan for characters. Here is the result:

FIGURE 4: LEAST SQUARE FIT LINE

Now we scan every pixel in FIGURE 3 at the height of the least square line from left to right. If the pixel is a 0 (miss) then it moves on. If the pixel is a 1 (hit), then the code performs region growing around that pixel and then isolates the character from the image.

FIGURE 5.1A: ISOLATED G FROM GOOGLE

FIGURE 5.1B: LEFTOVER IMAGE

The new image is cropped and then resized to a standard size of 18x15. Now the 0s and 1s are converted to 1s and -1s respectively. Remember that this is required for the Hopfield network. Here are the results:

FIGURE 5.1C: CROPPED 'G'

```
 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
 1   1   1   1  -1  -1  -1  -1  -1  -1  -1   1   1   1   1
 1   1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1   1
 1   1  -1  -1  -1  -1   1   1   1   1  -1  -1   1   1   1
 1   1  -1  -1   1   1   1   1   1   1   1   1   1   1   1
 1  -1  -1  -1   1   1   1   1   1   1   1   1   1   1   1
 1  -1  -1   1   1   1   1   1   1   1   1   1   1   1   1
 1  -1  -1   1   1   1   1   1   1   1   1   1   1   1   1
-1  -1  -1   1   1   1   1   1  -1  -1  -1  -1  -1  -1   1
-1  -1  -1   1   1   1   1   1  -1  -1  -1  -1  -1  -1  -1
 1  -1  -1   1   1   1   1   1  -1  -1  -1  -1  -1  -1  -1
 1  -1  -1   1   1   1   1   1   1   1   1   1  -1  -1   1
 1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
 1   1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
 1   1  -1  -1  -1   1   1   1   1   1  -1  -1  -1   1   1
 1   1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1
 1   1   1   1  -1  -1  -1  -1  -1  -1  -1   1   1   1   1
 1   1   1   1   1   1   1  -1  -1   1   1   1   1   1   1
```

FIGURE 5.1D: 18X15 BIPOLAR FORMAT

The matrix is fed as input to the `Char_Test` function, which successfully recognizes it as `'G'` and returns the same as output `ch`. The character `ch` is appended to the string `text`, which was initialized as an empty string but now stores `"G"`. Now we will repeat the same for the next character:

```
 1   1   1   1   1   1   1  -1   1   1   1   1   1   1   1
 1   1   1   1  -1  -1  -1  -1  -1  -1  -1   1   1   1   1
 1   1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1   1
 1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1
 1  -1  -1  -1  -1  -1   1   1   1  -1  -1  -1  -1  -1   1
 1  -1  -1  -1  -1   1   1   1   1   1  -1  -1  -1  -1   1
 1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
 1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
-1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1  -1
-1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1  -1
 1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
 1  -1  -1  -1   1   1   1   1   1   1   1  -1  -1  -1   1
 1  -1  -1  -1  -1   1   1   1   1   1  -1  -1  -1  -1   1
 1  -1  -1  -1  -1  -1   1   1   1  -1  -1  -1  -1  -1   1
 1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1
 1   1   1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1   1   1
 1   1   1   1  -1  -1  -1  -1  -1  -1  -1   1   1   1   1
 1   1   1   1   1   1  -1  -1  -1   1   1   1   1   1   1
```

The `Char_Test` function recognizes this character as `'o'`. This is appended with the string `text` which is `"G"`. Afterwards the string text contains `"Go"`. Now we move on to the next character.

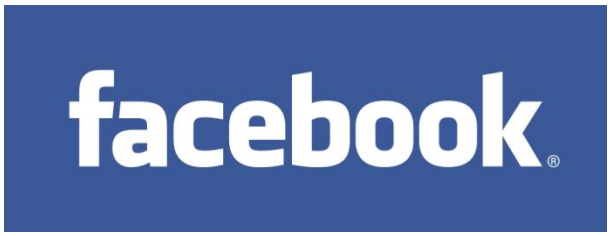
Note that there are no Hopfield networks for the letters I (capital i), l (small i) and l (small L). These characters return solid rectangles when we apply our segmentation algorithm, resulting in matrices that are all -1s, which we can't use in our Hopfield network as it will disrupt the performance. Instead we use a shortcut method. After obtaining the matrix, if the mean of all the values is less than -0.8, the character is assumed to be capital 'i' or small 'L', depending on whether it is the first character or not respectively.

After all the characters have been recognized the code prints the string `text` on the console, along with the time taken in seconds for the code to execute. The final result on the console is:

```
Google

 1.4014
```

The conclusion is that the code recognizes the logo successfully and the execution time is around 1.4 seconds, on a system running on an Intel Core i5-5200U processor clocked at 2.20 GHz.

Here are some other results:




```
facebook

 1.2481
```

The previous case was dark-on-light text and this one is light-on-dark text. Hence the line

`C = (B <= 200);` needs to be changed to `C = (B >= 200);` for this to work.

```
SAMSUNG

 0.73540
```

This is also dark-on-light text, so the line `C = (B <= 200);` remains unchanged. But since the characters are thick some extra image processing (erosion) has to be performed. The line `%C = imerode(C,X);`, which was previously commented out, is changed to:

```
C = imerode(C,X);
```

Additionally, we use the `CAPALPHA` character set instead of `ALPHA`, resulting in a significant speed up.

Step 6: Recognizing lines of text

Previously the text was assumed to be a single word. Now we are going to write a text recognition code that will separate individual words with whitespace characters.

Octave function: “Word_Recognition.m”

```
function text = Word_Recognition(C,charset)

text = '';

height = size(C,1);
width = size(C,2);

% Least square fitting of horizontal line on image
H = (1:height)'*ones(1,height);
dH = (H-H').^2;
S = dH*C*ones(width,1);
[Smin,h] = min(S);

% Isolate and recognize characters
for w = 1:width
  if (C(h,w) == 1)
  D = bwselect(C,w,h,4);
  C = 1-bwfill(1-C,w,h,4);
  [H2,W2] = find(D);
  E = D(min(H2):max(H2),min(W2):max(W2));
  E = imresize(E, [18 15], 'linear');
  E = -2*E+1;
```

```
  if(mean(mean(E))<-0.8)
    if(isempty(text))
      ch = 'I';
    else
      ch = 'i';
    end
  else
    ch = Char_Test(E,charset);
  end
  text = [text ch];
  endif
endfor

endfunction
```

Description:

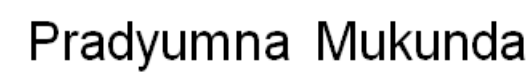
This code is the same as “Word_Recognition_demo.m”, but written as a function instead of a script.

Octave Script: “Line_Recognition_demo.m”

```
clc
clear all
close all

time0 = time;
pkg load image
charset = 'alpha';
text = '';

A = imread('sample.png');
if (isrgb(A))
  B = rgb2gray(A);
else
  B = A;
end
height = size(B,1);
C = imresize(B, 100/height, 'cubic');
C = (C <= 150);

height = size(C,1);
width = size(C,2);
X = strel("line",10,0);
D = imdilate(C,X,'same');
[E,N] = bwlabel(D);

% Isolate and recognize words
for n = 1:N
  F = (E == n);
  [H2,W2] = find(F);
  G = C(min(H2):max(H2),min(W2):max(W2));
  word = Word_Recognition(G,charset);
  text = [text word ' '];
end
disp(text);
disp(time - time0);
```

Description:

We will demonstrate the code on the following image:



The grayscaling and thresholding operations are performed as usual.

Now we perform dilation on the image. Dilation will bridge the gaps between characters in the same word, however, whitespace characters remain unbridged.

Using this image we perform segmentation to find the left and right limits of a word, then we use those limits to crop out the word from the *previous* image.

Now we input these images to the `Word_Recognition` function, recognize the words and display the result on the console. The code adds a whitespace character after every word.

```
Pradyumna Mukunda

 2.0042
```

Limitations

The project has the following limitations:

1. The image processing is not automatic and must be done by the user.
2. The code cannot recognize whether the text is dark-on-light text or light-on-dark text and the program must be modified for both cases.
3. The code cannot differentiate between capital i (I), small i (i) and small L (l) characters.
4. Also the code is not designed to differentiate between characters whose capitals and smalls look the same (i.e. C and c, O and o, S and s, V and v, W and w, Z and z)
5. The code cannot recognize special characters.
6. The code cannot recognize handwritten text or text printed in newspaper style font (although an extension can be made for the latter)
7. The project was extended for recognizing entire paragraphs of characters, but the results were very inaccurate and the code took very long time to execute. The

character resolution used is 18x15. Increasing the resolution improves accuracy at the cost of speed. Decreasing the resolution does vice versa.

Future Work

1. The project can be extended to recognize characters printed in newspaper style fonts (e.g. Times New Roman, Georgia etc.).
2. The project can be extended to recognize special characters.
3. The project can be extended to perform all image processing automatically using Machine Learning.
4. Further optimization for speed is possible in the following ways:
    a. The resolution can be decreased from 18x15, but as mentioned earlier this comes at the cost of accuracy.
    b. Currently the Hopfield networks store only two vectors. Using Hopfield Networks that store 3, 4 or more vectors would speed up the code dramatically. However this also comes at a cost of accuracy.
    c. We can use shortcuts instead of cycling through all of the characters
    d. The project is currently written in GNU Octave scripting language. We could try implementing the same in Python or C-language and see if it would speed up the code
5. After the code is further optimized for speed the project can be extended for recognizing entire paragraphs and documents
6. Accuracy can be improved by using a dictionary to correct wrongly recognized words, or by constraining the output of the code to dictionary words.
7. The project is implemented for printed English characters, but it can be extended for other scripts as well. (e.g. Hindi, Kannada, Tamil etc.)