**Project Report**

**ON**

**Handwritten Digits Recognition Using Neural Networks**

SUBMITTED BY

NISHANT

B.TECH (FINAL YEAR)

BATCH (2013-2017)

ROLL NO. 1306810094

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

MEERUT INSTITUTE OF ENGINEERING & TECHNOLOGY

Affiliated to

Dr. A.P.J. Abdul Kalam Technical University, Lucknow

TABLE OF CONTENTS

**CONTENT** **PAGE NO.**

DECLARATION ………………………………………………… 3

ACKNOWLEDGEMENT …………………………………… 4

COMPANY PROFILE........................................... 5

ABSTRACT ………………………………………………………………………………………….. 6

INTRODUCTION …………………………………………………………………………… 7

DATASET………………………………………………………………………………………… 9

VIZUALIZING THE DATASET…………………………………………………………….. 10

MODEL REPRESENTATION……………………………………………………………….. 10

FEEDFORWARD AND COST FUNCTION………………………………………………11

REGULARIZED COST FUNCTION…………………………………………………………13

BACKPROPOGATION………………………………………………………………………….13

SIGMOID GRADIENT……………………………………………………………………………14

RANDOM INTIALIZATION……………………………………………………………………..14

BACKPROPOGATION ALGO……………………………………………………………………15

GRADIENT CHECKING…………………………………………………………………………………17

REGULARIZED NEURAL NETWORK………………………………………………………….......18

VIZUALIZING THE HIDDEN LAYER…………………………………………………………………19

CLASSIFY YOUR OWN IMAGES………………………………………………………………………..21

CONCLUSION AND FUTURE WORKS………………………………………………………………..24

**DECLARATION**

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of my other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Name: NISHANT

Roll No.:1306810094

Date: 11 Nov 2016

**ACKNOWLEDGEMENT**

It gives me a great sense of pleasure to present the report of the Vocational Industrial Training during Summer Vacations. We owe special debt of gratitude to **Prof. Andrew Ng and Mr. Tom Mosher** my Guides & Teachers, Stanford University, Coursera for there constant support and guidance throughout the course of my work. There sincerity, thoroughness and perseverance have been a constant source of inspiration for me. It is only their cognizant efforts that my endeavors have seen light of the day.

I also do not like to miss the opportunity to acknowledge the contribution of all members of the Coursera Mentor Community and M.I.E.T for their kind assistance and cooperation during the development of the training report. Last but not the least, i acknowledge my friends for their contribution in the completion of the project given during training period.

Name: NISHANT

Roll No.: 1306810094

Date: 11 Nov 2016

**COMPANY PROFILE**

Coursera is a venture-backed, for-profit, educational technology company that offers massive open online courses (MOOCs). Coursera works with universities and other organizations to make some of their courses available online, offering courses in subjects such as physics, engineering, humanities, medicine, biology, social sciences, mathematics, business, computer science, digital marketing, and data science, among others.

Coursera offers all its courses "accessible for free"; some courses have the option to pay a fee to join the "Signature Track". Students on the Signature Track receive verified certificates, appropriate for employment purposes. These students authenticate their course submissions by sending webcam photos and having their typing pattern analyzed.

Coursera courses last approximately four to ten weeks, with one to two hours of video lectures a week. These courses provide quizzes, weekly exercises, peer-graded assignments, and sometimes a final project or exam. Courses are also provided on-demand, in which case users can take their time in completing the course with all of the material available at once.

Founded in 2012 by computer science professors Andrew Ng and Daphne Koller from Stanford University.

Coursera started in 2012, working with Stanford University, Princeton University, the University of Michigan, and the University of Pennsylvania. Twelve partners (including Johns Hopkins University and Caltech) were added in July 2012, followed by 17 more in September 2012. In February 2013, the company announced another 29 partner universities. The current total number of partners is 133, across 26 countries, offering 1,467 courses.

**ABSTRACT**

*PROJECT NAME:- HANDWRITTEN DIGIT CLASSIFIER USING MACHINE LEARNING CONCEPTS*

**Handwriting recognition** (or **HWR**[[1]](https://en.wikipedia.org/wiki/Handwriting_recognition#cite_note-1)) is the ability of a computer to receive and interpret intelligible [handwritten](https://en.wikipedia.org/wiki/Handwriting) input from sources such as [paper](https://en.wikipedia.org/wiki/Paper) documents, [photographs](https://en.wikipedia.org/wiki/Photograph), [touch-screens](https://en.wikipedia.org/wiki/Touch-screen) and other devices. The image of the written text may be sensed "off line" from a piece of paper by optical scanning ([optical character recognition](https://en.wikipedia.org/wiki/Optical_character_recognition)) or [intelligent word recognition](https://en.wikipedia.org/wiki/Intelligent_word_recognition). Alternatively, the movements of the pen tip may be sensed "on line", for example by a pen-based computer screen surface, a generally easier task as there are more clues available.

Handwriting recognition principally entails [optical character recognition](https://en.wikipedia.org/wiki/Optical_character_recognition). However, a complete handwriting recognition system also handles formatting, performs correct [segmentation](https://en.wikipedia.org/wiki/Segment_%28handwriting%29) into characters and finds the most plausible words.

**INTRODUCTION**

Off-line handwriting recognition involves the automatic conversion of text in an image into letter codes which are usable within computer and text-processing applications. The data obtained by this form is regarded as a static representation of handwriting. Off-line handwriting recognition is comparatively difficult, as different people have different handwriting styles.

**Neural networks**

Neural network recognizers learn from an initial image training set. The trained network then makes the character identifications. Each neural network uniquely learns the properties that differentiate training images. It then looks for similar properties in the target image to be identified. Neural networks are quick to set up; however, they can be inaccurate if they learn properties that are not important in the target data.

**Feature extraction**

Feature extraction works in a similar fashion to neural network recognizers. However, programmers must manually determine the properties they feel are important.

Some example properties might be:

* Aspect Ratio.
* Percent of pixels above horizontal half point
* Percent of pixels to right of vertical half point
* Number of strokes
* Average distance from image center
* Is reflected y axis
* Is reflected x axis

**Multi-class Classification**

For this exercise, you will use logistic regression and neural networks to

recognize handwritten digits (from 0 to 9). Automated handwritten digit

recognition is widely used today - from recognizing zip codes (postal codes)

on mail envelopes to recognizing amounts written on bank checks. This

exercise will show you how the methods you've learned can be used for this classification task.

**Dataset**

You are given a data set in ex4data1.mat that contains 5000 training exam-

ples of handwritten digits.2 The .mat format means that that the data has

been saved in a native Octave/MATLAB matrix format, instead of a text

(ASCII) format like a csv-\_le. These matrices can be read directly into your

program by using the load command. After loading, matrices of the correct

dimensions and values will appear in your program's memory. The matrix

will already be named, so no need to assign names to them.

% Load saved matrices from file

load('ex3data1.mat');

% The matrices X and y will now be in your Octave environment

There are 5000 training examples in ex3data1.mat, where each training

example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is

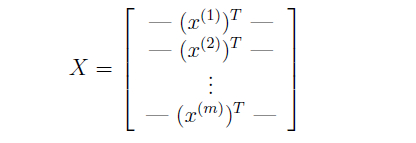
represented by a oating point number indicating the grayscale intensity at

that location. The 20 by 20 grid of pixels is \unrolled" into a 400-dimensional

vector. Each of these training examples becomes a single row in our data

matrix X. This gives us a 5000 by 400 matrix X where every row is a training

example for a handwritten digit image.



The second part of the training set is a 5000-dimensional vector y that

contains labels for the training set. To make things more compatible with

Octave/MATLAB indexing, where there is no zero index, we have mapped

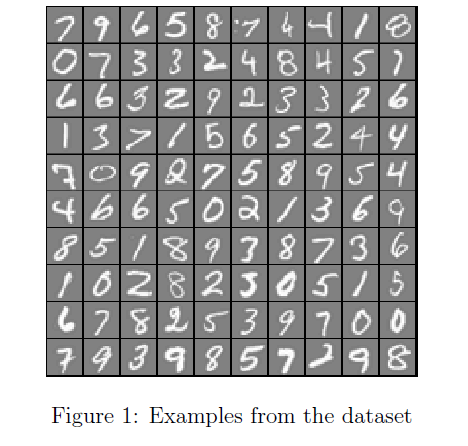
the digit zero to the value ten. Therefore, a \0" digit is labeled as \10", while

the digits \1" to \9" are labeled as \1" to \9" in their natural order.

**Visualizing the data**

In the first part of ex4.m, the code will load the data and display it on a

2-dimensional plot (Figure 1) by calling the function displayData.



**Model representation**

Our neural network is shown in Figure 2. It has 3 layers { an input layer,

a hidden layer and an output layer. Recall that our inputs are pixel valuesof digit images. Since the images are of size 20 \_ 20, this gives us 400 input

layer units (not counting the extra bias unit which always outputs +1). The

training data will be loaded into the variables X and y by the ex4.m script.

You have been provided with a set of network parameters (\_(1);\_(2))

. These are stored in ex4weights.mat and will be

loaded by ex4.m into Theta1 and Theta2. The parameters have dimensions

that are sized for a neural network with 25 units in the second layer and 10

output units (corresponding to the 10 digit classes).

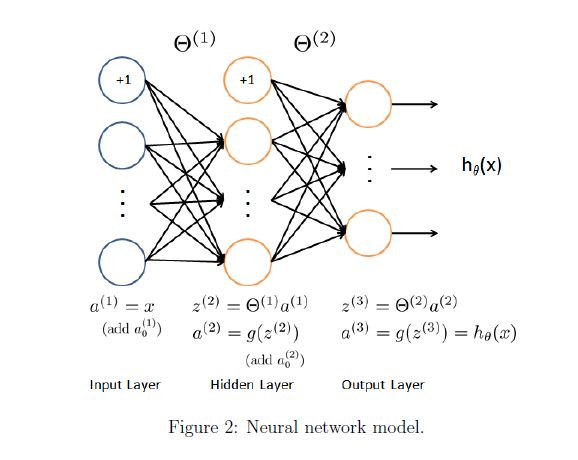
% Load saved matrices from file

load('ex4weights.mat');

% The matrices Theta1 and Theta2 will now be in your workspace

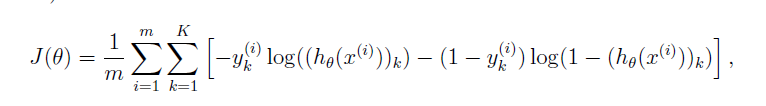
% Theta1 has size 25 x 401

% Theta2 has size 10 x 26

Figure 2: 

**Feedforward and cost function**

Now you will implement the cost function and gradient for the neural net-

work. First, complete the code in nnCostFunction.m to return the cost. Recall that the cost function for the neural network (without regularization) iswhere h\_(x(i)) is computed as shown in the Figure 2 and K = 10 is the total

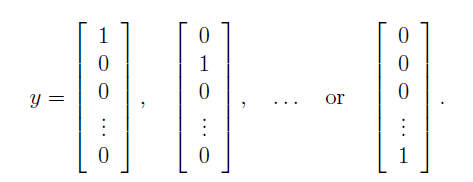
number of possible labels. Note that h\_(x(i))k = a(3)

k is the activation (outputvalue) of the k-th output unit. Also, recall that whereas the original labels

(in the variable y) were 1, 2, ..., 10, for the purpose of training a neural

network, we need to recode the labels as vectors containing only values 0 or

1, so that



For example, if x(i) is an image of the digit 5, then the corresponding

y(i) (that you should use with the cost function) should be a 10-dimensional

vector with y5 = 1, and the other elements equal to 0.

The matrix X contains the examples in rows

(i.e., X(i,:)' is the i-th training example x(i), expressed as a n \_ 1

vector.) When you complete the code in nnCostFunction.m, you will

need to add the column of 1's to the X matrix. The parameters for each

unit in the neural network is represented in Theta1 and Theta2 as one

row. Speci\_cally, the \_rst row of Theta1 corresponds to the \_rst hidden

unit in the second layer. You can use a for-loop over the examples to

compute the cost.

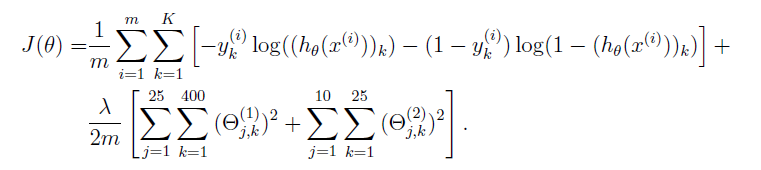
Once you are done, ex4.m will call your nnCostFunction using the loaded

set of parameters for Theta1 and Theta2. You should see that the cost is

about 0.287629.

**Regularized cost function**

The cost function for neural networks with regularization is given by

You can assume that the neural network will only have 3 layers { an input

layer, a hidden layer and an output layer.

Note that you should not be regularizing the terms that correspond to

the bias. For the matrices Theta1 and Theta2, this corresponds to the \_rst

column of each matrix. You should now add regularization to your cost

function. Notice that you can \_rst compute the unregularized cost function

J using your existing nnCostFunction.m and then later add the cost for the

regularization terms.

Once you are done, ex4.m will call your nnCostFunction using the loaded

set of parameters for Theta1 and Theta2, and \_ = 1. You should see that

the cost is about 0.383770.

**Backpropagation**

In this part of the exercise, you will implement the backpropagation algo-

rithm to compute the gradient for the neural network cost function. You

will need to complete the nnCostFunction.m so that it returns an appropri-

ate value for grad. Once you have computed the gradient, you will be able

to train the neural network by minimizing the cost function J(\_) using an

advanced optimizer such as fmincg.

You will \_rst implement the backpropagation algorithm to compute the

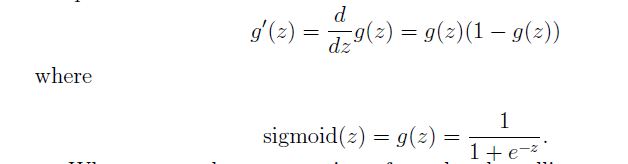
gradients for the parameters for the (unregularized) neural network. Afteryou have veri\_ed that your gradient computation for the unregularized case

is correct, you will implement the gradient for the regularized neural network.

**Sigmoid gradient**

To help you get started with this part of the project, you will \_rst implement

the sigmoid gradient function. The gradient for the sigmoid function can be

computed asWhen you are done, try testing a few values by calling sigmoidGradient(z)

at the Octave/MATLAB command line. For large values (both positive and

negative) of z, the gradient should be close to 0. When z = 0, the gradi-

ent should be exactly 0.25. Your code should also work with vectors and

matrices. For a matrix, your function should perform the sigmoid gradient

function on every element.

**Random initialization**

When training neural networks, it is important to randomly initialize the pa-

rameters for symmetry breaking. One e\_ective strategy for random initializa-

tion is to randomly select values for \_(l) uniformly in the range [􀀀\_init; \_init].

You should use \_init = 0:12.2 This range of values ensures that the parameters

are kept small and makes the learning more e\_cient.

Your job is to complete randInitializeWeights.m to initialize the weights

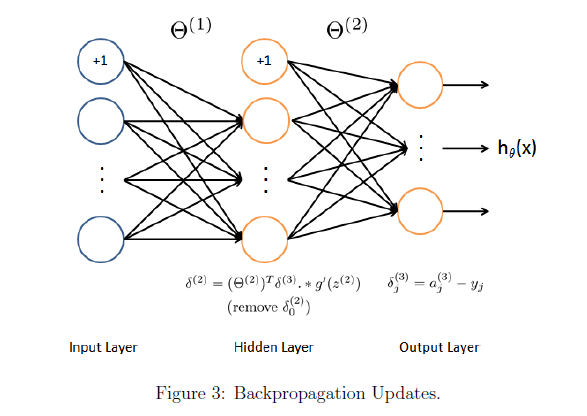
for \_; modify the \_le and \_ll in the following code:

% Randomly initialize the weights to small values

epsilon init = 0.12;

W = rand(L out, 1 + L in) \* 2 \* epsilon init -epsilon init;

**Backpropagation**



Now, you will implement the backpropagation algorithm. Recall that

the intuition behind the backpropagation algorithm is as follows. Given a

training example (x(t); y(t)), we will \_rst run a \forward pass" to compute

all the activations throughout the network, including the output value of the

hypothesis h\_(x). Then, for each node j in layer l, we would like to compute

an \error term" \_(l)

j that measures how much that node was \responsible"

for any errors in our output.

For an output node, we can directly measure the di\_erence between the

network's activation and the true target value, and use that to de\_ne \_(3)

j

(since layer 3 is the output layer). For the hidden units, you will compute

\_(l)

j based on a weighted average of the error terms of the nodes in layer

(l + 1). In detail, here is the backpropagation algorithm (also depicted in Figure

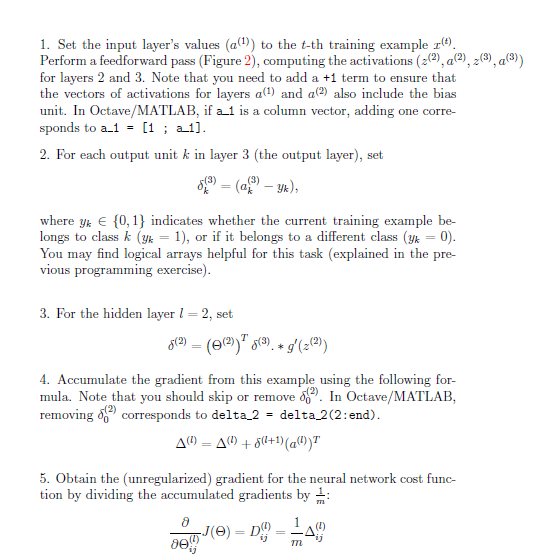
3). You should implement steps 1 to 4 in a loop that processes one example

at a time. Concretely, you should implement a for-loop for t = 1:m and

place steps 1-4 below inside the for-loop, with the tth iteration performing

the calculation on the tth training example (x(t); y(t)). Step 5 will divide the

accumulated gradients by m to obtain the gradients for the neural network

cost function. 

After you have implemented the backpropagation algorithm, the script

ex4.m will proceed to run gradient checking on your implementation. The

gradient check will allow you to increase your con\_dence that your code is

computing the gradients correctly.

**Gradient Checking**

In your neural network, you are minimizing the cost function J(\_). To

perform gradient checking on your parameters, you can imagine \unrolling"

the parameters \_(1);\_(2) into a long vector \_. By doing so, you can think of

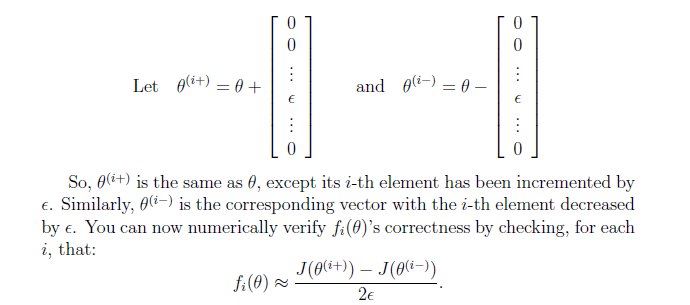
the cost function being J(\_) instead and use the following gradient checking

procedure.

Suppose you have a function fi(\_) that purportedly computes @

@\_i

J(\_);

you'd like to check if fi is outputting correct derivative values. 

The degree to which these two values should approximate each other will

depend on the details of J. But assuming \_ = 10􀀀4, you'll usually \_nd that

the left- and right-hand sides of the above will agree to at least 4 signi\_cant

digits (and often many more).

We have implemented the function to compute the numerical gradient in computeNumericalGradient.m.

In the next step of ex4.m, it will run the provided function checkNNGradients.m

which will create a small neural network and dataset that will be used for

checking your gradients. If your backpropagation implementation is correct, you should see a relative di\_erence that is less than 1e-9.

Practical Tip: When performing gradient checking, it is much more

e\_cient to use a small neural network with a relatively small number

of input units and hidden units, thus having a relatively small number

of parameters. Each dimension of \_ requires two evaluations of the cost

function and this can be expensive. In the function checkNNGradients,

our code creates a small random model and dataset which is used with

computeNumericalGradient for gradient checking. Furthermore, after

you are con\_dent that your gradient computations are correct, you should

turn o\_ gradient checking before running your learning algorithm.

Practical Tip: Gradient checking works for any function where you are

computing the cost and the gradient. Concretely, you can use the same

computeNumericalGradient.m function to check if your gradient imple-

mentations for the other exercises are correct too (e.g., logistic regression's

cost function).

**Regularized Neural Networks**

After you have successfully implemeted the backpropagation algorithm, you

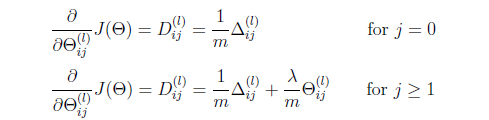
will add regularization to the gradient. To account for regularization, it

turns out that you can add this as an additional term after computing the

gradients using backpropagation.

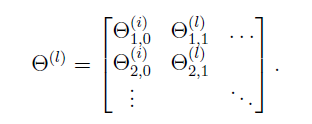
Speci\_cally, after you have computed \_(l)

ij using backpropagation, you

should add regularization using

Note that you should not be regularizing the \_rst column of \_(l) which

is used for the bias term. Furthermore, in the parameters \_(l)

ij , i is indexedstarting from 1, and j is indexed starting from 0. Thus, 

Somewhat confusingly, indexing in Octave/MATLAB starts from 1 (for

both i and j), thus Theta1(2, 1) actually corresponds to \_(l)

2;0 (i.e., the entry

in the second row, \_rst column of the matrix \_(1) shown above)

Now modify your code that computes grad in nnCostFunction to account

for regularization. After you are done, the ex4.m script will proceed to run

gradient checking on your implementation. If your code is correct, you should

expect to see a relative di\_erence that is less than 1e-9.

**Learning parameters using fmincg**

After you have successfully implemented the neural network cost function

and gradient computation, the next step of the ex4.m script will use fmincg

to learn a good set parameters.

After the training completes, the ex4.m script will proceed to report the

training accuracy of your classi\_er by computing the percentage of examples

it got correct. If your implementation is correct, you should see a reported

training accuracy of about 95.3% (this may vary by about 1% due to the

random initialization). It is possible to get higher training accuracies by

training the neural network for more iterations. We encourage you to try

training the neural network for more iterations (e.g., set MaxIter to 400) and

also vary the regularization parameter \_. With the right learning settings, it

is possible to get the neural network to perfectly \_t the training set.

**Visualizing the hidden layer**

**.**

One way to understand what your neural network is learning is to visualize

what the representations captured by the hidden units. Informally, given a

particular hidden unit, one way to visualize what it computes is to \_nd an

input x that will cause it to activate (that is, to have an activation value

(a(l)

i ) close to 1). For the neural network you trained, notice that the ith row

of \_(1) is a 401-dimensional vector that represents the parameter for the ithhidden unit. If we discard the bias term, we get a 400 dimensional vector

that represents the weights from each input pixel to the hidden unit.

Thus, one way to visualize the \representation" captured by the hidden

unit is to reshape this 400 dimensional vector into a 20 \_ 20 image and

display it.3 The next step of ex4.m does this by using the displayData

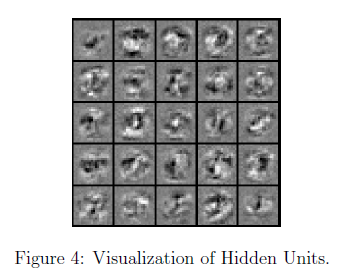
function and it will show you an image (similar to Figure 4) with 25 units,

each corresponding to one hidden unit in the network.

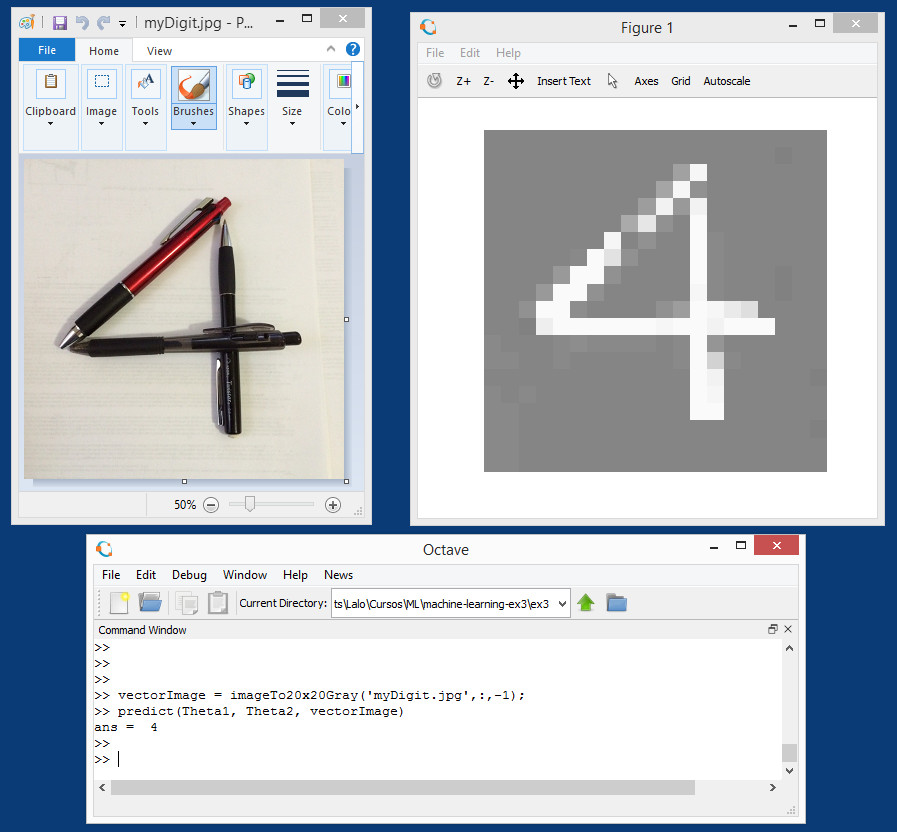
In your trained network, you should \_nd that the hidden units corre-

sponds roughly to detectors that look for strokes and other patterns in the

input.



# Classify your own images



## Approach

1. The approach is to have a function that converts our photo to the format the classifier is expecting. As if it was just a sample from the training data set.
2. Use the classifier to predict the digit in the converted image.

## Introduction

The classifier provided expects 20 x 20 pixels black and white images converted in a row vector of 400 real numbers like this

[ 0.14532, 0.12876, ...]

Each pixel is represented by a real number between -1.0 to 1.0, meaning -1.0 equal black and 1.0 equal white (any number in between is a shade of gray, and number 0.0 is exactly the middle gray).

### .jpg and color RGB images

The most common image format that can be read by Octave is .jpg using function that outputs a three-dimensional matrix of integer numbers from 0 to 255, representing the height x width x 3 integers as indexes of a color map for each pixel (explaining color maps is beyond scope).

Image3DmatrixRGB = imread("myOwnPhoto.jpg");

### Convert to Black & White

A common way to convert color images to black & white, is to convert them to a YIQ standard and keep only the Y component that represents the luma information (black & white). I and Q represent the chrominance information (color).  
Octave has a function **rgb2ntsc()** that outputs a similar three-dimensional matrix but of real numbers from -1.0 to 1.0, representing the height x width x 3 (Y luma, I in-phase, Q quadrature) intensity for each pixel.

Image3DmatrixYIQ = rgb2ntsc(MyImageRGB);

To obtain the Black & White component just discard the I and Q matrices. This leaves a two-dimensional matrix of real numbers from -1.0 to 1.0 representing the height x width pixels black & white values.

Image2DmatrixBW = Image3DmatrixYIQ(:,:,1);

### Cropping to square image

It is useful to crop the original image to be as square as possible. The way to crop a matrix is by selecting an area inside the original B&W image and copy it to a new matrix. This is done by selecting the rows and columns that define the area. In other words, it is copying a rectangular subset of the matrix like this:

croppedImage = Image2DmatrixBW(origen1:size1, origin2:size2);

Cropping does not have to be all the way to a square. **It could be cropping just a percentage of the way to a square** so you can leave more of the image intact. The next step of scaling will take care of streaching the image to fit a square.

### Scaling to 20 x 20 pixels

The classifier provided was trained with 20 x 20 pixels images so we need to scale our photos to meet. It may cause distortion depending on the height and width ratio of the cropped original photo. There are many ways to scale a photo but we are going to use the simplest one. We lay a scaled grid of 20 x 20 over the original photo and take a sample pixel on the center of each grid. To lay a scaled grid, we compute two vectors of 20 indexes each evenly spaced on the original size of the image. One for the height and one for the width of the image. For example, in an image of 320 x 200 pixels will produce to vectors like

[9 25 41 57 73 ... 313]  % 20 indexes

[6 16 26 36 46 ... 196]  % 20 indexes

Copy the value of each pixel located by the grid of these indexes to a new matrix. Ending up with a matrix of 20 x 20 real numbers.

### Black & White to Gray & White

The classifier provided was trained with images of white digits over gray background. Specifically, the 20 x 20 matrix of real numbers ONLY range from 0.0 to 1.0 instead of the complete black & white range of -1.0 to 1.0, this means that we have to normalize our photos to a range 0.0 to 1.0 for this classifier to work. But also, we invert the black and white colors because is easier to "draw" black over white on our photos and we need to get white digits. So in short, we **invert black and white** and **stretch black to gray**.

### Rotation of image

Some times our photos are automatically rotated like in our celular phones. The classifier provided can not recognize rotated images so we may need to rotate it back sometimes. This can be done with an Octave function **rot90()** like this.

ImageAligned = rot90(Image, rotationStep);

Where rotationStep is an integer: -1 mean rotate 90 degrees CCW and 1 mean rotate 90 degrees CW.

**CONCLUSION AND FUTURE WORKS**

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers total.

* Number of input units = dimension of features x(i)
* Number of output units = number of classes
* Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
* Defaults: 1 hidden layer. If more than 1 hidden layer, then the same number of units in every hidden layer.

**Training a Neural Network**

1. Randomly initialize the weights
2. Implement forward propagation to get hθ(x(i))
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.
7. When we perform forward and back propagation, we loop on every training example:
8. for i = 1:m,
9. Perform forward propagation and backpropagation using example (x(i),y(i))
10. (Get activations a(l) and delta terms d(l) for l = 2,...,L

The future aspect of this application will be to apply this network on linear classification problems, and enhance the pattern recognition techniques using image processing and computer vision systems.