

REPORTTTTTT

Content

1. Abstract
2. Introduction
3. Character Extraction
4. Neural Networks  
   i) Sigmoid function  
   ii) Model Representation  
   iii) Multi-class Classification  
   iv) Cost function   
   v) Back-Propagation Algorithm
5. Ten-Fold Cross validation
6. Online Digit Classification using Java Files: Extending the neural network to English alphabet

Abstract

Handwriting recognition is the translation of handwritten documents into digital data that can be used by a computer. Typically, this data is then translated into digital text. Handwriting recognition can be performed using multiple methods, but the two principle methods are using OCR (Optical Character Recognition) and Neural Networks. Neural Networks are advantageous over OCR, since they are faster and more accurate than OCR (Accuracy of more than 95% can be achieved using Neural Networks).

Neural Networks are preferred over linear regression. This is due to the fact that performing linear regression with a complex set of data with many features is unwieldy and in applications such as Artificial intelligence where features (linear, quadratic, cubic) using the pixel values are to be generated, the growth of the number of features for n pixels is O(n2) for quadratic features and O(n3) for cubic features. These are very steep growths and as the number of features increase, the number of quadratic and cubic features increases very rapidly and become quite impractical. Neural Networks offers an alternative way to perform machine learning when we have complex hypothesis with many features. It has a flexibility to create its own features.  
This project work is based on Machine Learning course conducted by Andrew G at courser.

Introduction

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. And, as of today, OCR engines are primarily focused on machine printed text and ICR for hand "printed" (written in capital letters) text. There is no OCR/ICR engine that supports handwriting recognition as of today.

**Problem domain reduction techniques**

Narrowing the problem domain often helps increase the accuracy of handwriting recognition systems. A form field for a ZIP code for example, would contain only the characters 0-9. This fact would reduce the number of possible identifications.

Primary techniques:

* Specifying specific character ranges
* Utilization of specialized forms

**Character extraction**

Off-line character recognition often involves scanning a form or document written sometime in the past. This means the individual characters contained in the scanned image will need to be extracted. Tools exist that are capable of performing this step. However, there are several common imperfections in this step. The most common is when characters that are connected are returned as a single sub-image containing both characters. This causes a major problem in the recognition stage. Yet many algorithms are available that reduce the risk of connected characters.

**Character recognition**

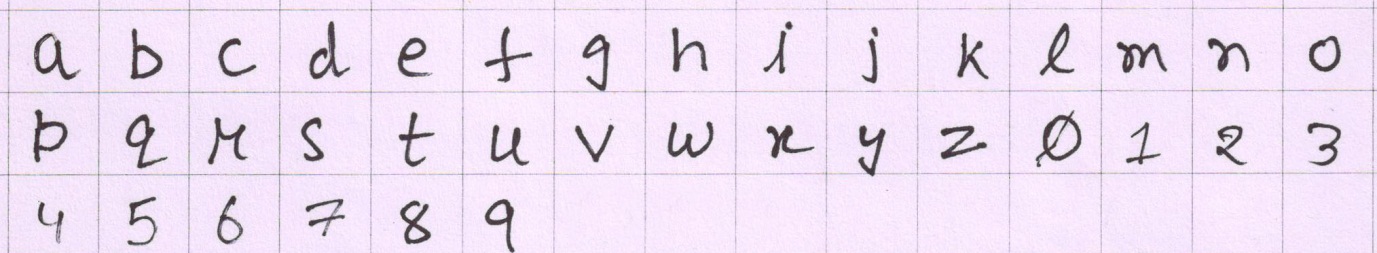
After the extraction of individual characters occurs, a recognition engine is used to identify the corresponding computer character. Several different recognition techniques are currently available.

**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.

Character Extraction

Extraction of character form a normal hand written document is very difficult. So to make our results more accurate and fast we read the hand written characters from a grid as shown



We then resize this image and reduce the number of pixels such that each cell in the gird is made of 20 X 20 pixels.

Then each cell is then converted to a linear array, so we make a new matrix of size number of cells X 400 on which we now apply neural networks.

Neural Networks

# Neurons and the Brain

Neural networks are limited imitations of how our own brains work. They've had a big recent resurgence because of advances in computer hardware.

There is evidence that the brain uses only one "learning algorithm" for all its different functions. Scientists have tried cutting the connection between the ears and the auditory cortex and found that the auditory cortex then takes up all the functions of seeing.

This principle is called "neuroplasticity" and has many examples and experimental evidence.

# Why are neural networks so popular

Neural Networks are a pretty old algorithm that was originally motivated by the goal of having machines that can mimic the brain. They work really well for different machine learning problems.

The origins of Neural Networks was as algorithms that try to mimic the brain in a sense that if we want to build learning systems, why not mimic the most amazing learning machine we know about, which is the brain? NNs were very widely used throughout the 1980’s and 1990’s and, for various reasons, their popularity diminished in the late 90’s. But more recently NNs have had a major resurgence. One of the reasons for this resurgence is that NNs are somewhat computationally expensive algorithms and so it was only recently that computers became fast enough to run really large scale NNs. Because of that, modern NNs are today the state of the art technique for many applications.

So, when you think about mimicking the brain, you realize that the human brain does amazing things, right? The brain can learn to see and process images, learn to hear, and learn to process our sense of touch. We can learn to do math, learn to do calculus. And the brain does so many different and amazing things that it seems like if you want to mimic the brain you have to write lots of different pieces of software to mimic all of these different, fascinating, amazing things that the brain does.

But there is this fascinating hypothesis that *the way the brain does all of these different things is not worth like a thousand different programs, but instead, the way the brain does it is worth just a single learning algorithm*.

Thisis just a hypothesis, but still there are few evidences for this.

That little red part of the brain in figure is the auditory cortex. The way you understand sound is because your ear is taking the sound signal and routing that signal to your auditory cortex, and that’s what’s allowing you to understand the voice of people around you.

Neuroscientists have done the following fascinating experiment where you cut the wire from the ears to the auditory cortex in an animal’s brain and rewire so that the signal going from the eyes to the optic nerve eventually gets routed to the auditory cortex. If you do this, it turns out the auditory cortex will learn to see – in every single sense of the word see, as we know it.

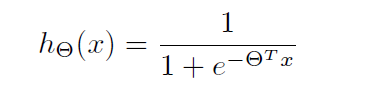
And instead of needing to implement a thousand different programs or algorithms to do the thousand wonderful things that the brain does, maybe what we need to do is just to figure out and implement some approximation to whatever the brain’s one and unique learning algorithm is, and then just let the brain (or algorithm) learn by itself how to process these different types of data. To a surprisingly large extent it seems that we can plug in almost any sensor to almost any part of the brain and so, by some reason, the brain will learn to deal with it. Here are a few more examples.

This is an example of learning to see with your tongue. This is actually a system called *Brain Port*.

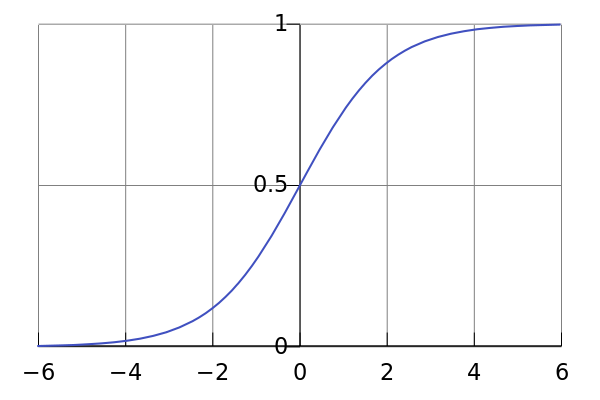
Here’s a second example of human echo location, or human sonar. So, there are two ways you can do this: you can either snap your fingers, or click your tongue. Blind people today that are being trained in schools to do this and learn to interpret the pattern of sounds bouncing off the environment.

So, it’s pretty amazing to what extent you can plug in almost any sensor to the brain and the brain’s learning algorithm will just figure out how to learn from and deal with that data. There’s a sense that if we can figure out what the brain’s learning algorithm is, and implement it or implement some approximation to it on a computer would be our best shot at making real progress towards the Artificial Intelligence dream of someday building truly intelligent machines.

# Sigmoid Function



A **sigmoid function** is a mathematical function having an "S" shape (**sigmoid curve**). Often, *sigmoid function* refers to the special case of the logistic function shown at right and defined by the formula



Computation of Complex functions can be done through use of sigmoid function. Let’s start with simple functions.

AND FUNCTION

Y=x1&x2

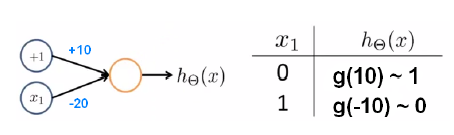
Y,x1,x2 E {0,1}



**NOT FUNCTION**

Y=NOT(x1)

Y, x1 E {0,1}



Now using the above two we can calculate the NOR function.

(x1)NOR(x2) = (NOT(x1)) AND (NOT(x2))

Hence complicated functions were calculated by their separate calculation and aggregation.

# Model Representation

Let's examine how we will represent a hypothesis function using neural networks.

At a very simple level, neurons are basically computational units that take input (**dendrites**) as electrical input (called "spikes") that are channelled to outputs (**axons**).

In our model, our dendrites are like the input features (http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20x_1\cdots%20x_n), and the output is the result of our hypothesis function:

In this model our http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20x_0 input node is sometimes called the "bias unit." It is always equal to 1.

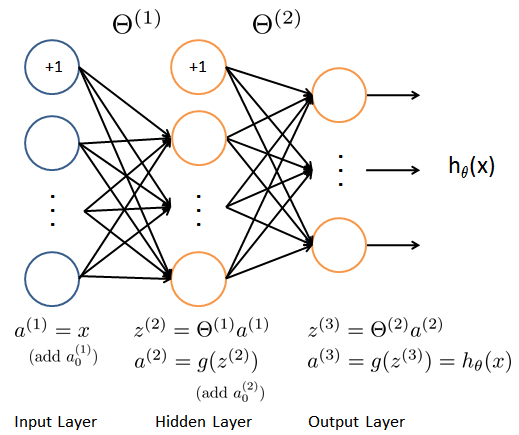
In neural networks, we use the same logistic function as in classification: http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20\frac%7b1%7d%7b1%20+%20e%5e%7b-\theta%5eTx%7d%7d. In neural networks however we sometimes call it a sigmoid (logistic) **activation** function.

Our "theta" parameters are sometimes instead called "weights" in the neural networks model.

Visually, a simplistic representation looks like:

C:\Users\Aman\Downloads\CodeCogsEqn (1).png

Our input nodes (layer 1) go into another node (layer 2), and are output as the hypothesis function.

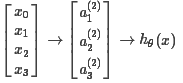


The first layer is called the "input layer" and the final layer the "output layer," which gives the final value computed on the hypothesis.

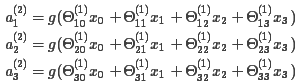
We can have intermediate layers of nodes between the input and output layers called the "hidden layer."

We label these intermediate or "hidden" layer nodes http://latex.codecogs.com/png.latex?\large%20\dpi%7b120%7d%20a%5e2_0%20\cdots%20a%5ej_n and call them "activation units." 

If we had one hidden layer, it would look visually something like:



The value for each of the “activation” nodes is obtained as follows:





This is saying that we compute our activation nodes by using a http://latex.codecogs.com/png.latex?\large%20\dpi%7b120%7d%203\times%204 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix http://latex.codecogs.com/png.latex?\large%20\dpi%7b120%7d%20\Theta%5e%7b(2)%7d containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, http://latex.codecogs.com/png.latex?\large%20\dpi%7b120%7d%20\Theta%5e%7b(j)%7d.

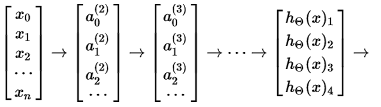
The dimensions of these matrices of weights are determined as follows: 

The +1 comes from the addition in http://latex.codecogs.com/png.latex?\large%20\dpi%7b120%7d%20\Theta%5e%7b(j)%7d of the "bias nodes," http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20x_0 and http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20\Theta_0%5e%7b(j)%7d . In other words the output nodes will not include the bias nodes while the inputs will.

Example: layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20\Theta%5e%7b(1)%7d is going to be 4 X 3 where http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20s_j%20=%202 and http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20s_%7bj+1%7d%20=%204 , so http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20s_%7bj+1%7d%20\times%20(s_j%20+%201)%20=%204%20\times%203  .

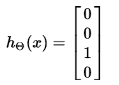
# Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four final resulting classes:



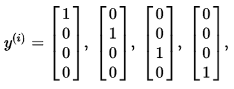
Our final layer of nodes, when multiplied by its theta matrix, will result in another vector, on which we will apply the g() logistic function to get a vector of hypothesis values.

Our resulting hypothesis for one set of inputs may look like:



In which case our resulting class is the third one down, or  .

We can define our set of resulting classes as y:



Our final value of our hypothesis for a set of inputs will be one of the elements in y.

Cost Function

Let's first define a few variables that we will need to use:

L = total number of layers in the network

s =number of units (not counting bias unit) in layer l

K = number of output units/classes

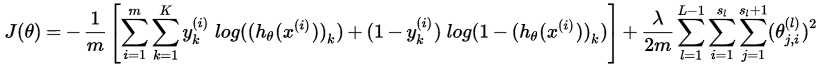
Recall that in neural networks, we may have many output nodes. We denote http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20h_\Theta(x)_k as being a hypothesis that results in the kth output.

Our cost function for neural networks is going to be a generalization of the one we used for logistic regression.

Recall that the cost function for regularized logistic regression was:



For neural networks, it is going to be slightly more complicated:

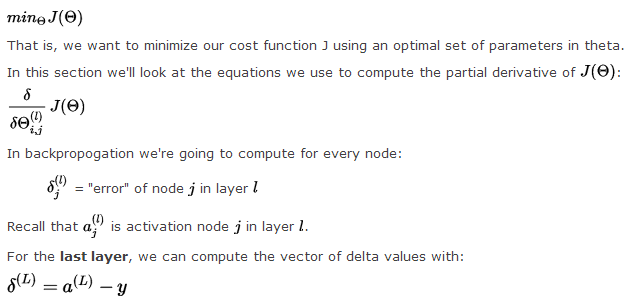


We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The columns in our current theta matrix is equal to the number of nodes in our current layer. The rows in our current theta matrix is equal to the number of nodes in the next layer. As before with logistic regression, we square every term.

Back propagation Algorithm

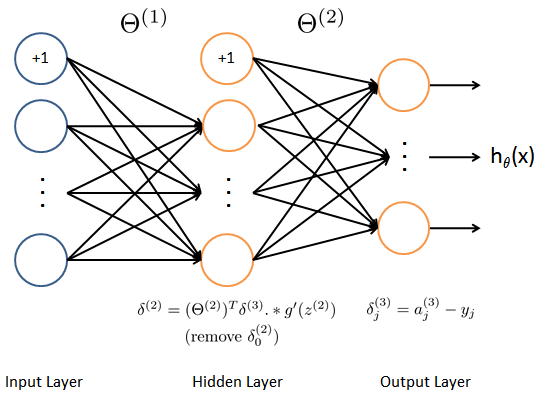
"Back propagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.



Where L is our total number of layers and  is the vector of activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.

To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:





The delta values of layer are calculated by multiplying the delta values in the next layer with the theta matrix of layer. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function evaluated with the input values given by.

The g-prime derivative terms can also be written out as:



This can be shown and proved in calculus.

The full back propagation equation for the inner nodes is then:

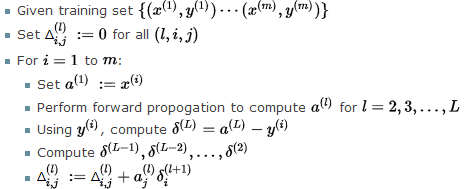


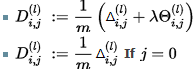
We can compute our partial derivative terms by multiplying our activation values and our error values:

This however ignores regularization, which we'll deal with later.

We can now take all these equations and put them together into a back propagation algorithm:

**Back propagation Algorithm**





The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The  terms are the partial derivatives and the results we are looking for.

Putting it Together

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  http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20x%5e%7b(i)%7d
* Number of output units = number of classes
* 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 http://latex.codecogs.com/png.latex?\large%20\dpi%7b80%7d%20h_\theta(x%5e%7b(i)%7d)
3. Implement the cost function
4. Implement back propagation to compute partial derivatives
5. Use gradient checking to confirm that your back propagation 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.

When we perform forward and back propagation, we loop on every training example:

For i = 1:m,

Perform forward propagation and back propagation using example (x(i),y(i))

(Get activations a(l) and delta terms d(l) for l = 2,...,L

Ten-Fold Cross Validation

After the training of the network, the neural weights/parameters Theta1 & Theta2 are stored in <neuralParameters.mat> which are further used for ten-fold cross validation and Online digit classification. For the 10x fold cross validation, the entire data set is divided into 10 equal sized sets and then training is done on 9 sets and testing on the unseen 10th data set. It makes use of the standard predict function to fetch the data and neural weights to predict the class with maximum probability. The average values for the 10x fold cross validation is in the range: 93% to 97%

Extending the neural network for English Alphabet

To extend the digit classification to the letters a-z, we make the output classes 0-9 & a-z, making a total of 36 labels. Training data is fetched from “./corpus/” directory where each data file set is stored in a .txt file in binary format which is created from online hand-written text on a java applet interface. The name of .txt file is the corresponding class to be identified with.

Java Files

The java project files include a simple interface to track the mouse pointer and draw any symbol (digit/letter) on an applet interface and then stores it on a matrix which then writes the content of matrix in the corpus directory.

References

[1] A. Graves. Supervised Sequence Labelling with Recurrent Neural Networks. PhD thesis.   
[2] A. Graves, S. Fern´andez, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In Proceedings of the Internationa Conference on Machine Learning, ICML 2006, Pittsburgh, USA, 2006.   
[3] Coursera  
[4] scholarly articles: