Article
Browse Code
Stats
Revisions
Alternatives
Comments (153)

Tagged as

VS.NET2003
Windows
.NET
Dev
Intermediate

Related Articles

A C# Project in Optical Character Recognition (OCR) Using Chain Code

Unicode Optical Character Recognition

Daniel Admassu, 23 Aug 2006

★★★★★ 4.93 (131 votes)

Rate this: ☆☆☆☆☆

The Project is an optical character recongnition application using artificial neural networks.

Info

First Posted	23 Aug 2006
Views	430,374
Bookmarked	304 times

Download source files - 1.2 Mb

How To: Use Office 2007 OCR Using C#

Creating Optical Character Recognition (OCR) applications using Neural Networks

OCR with Microsoft® Office

Introduction to Programming with LEADTOOLS .NET OCR - Multi-threaded OCR SDK for 32 and 64 Bit Development

Abstract

The central objective of this project is demonstrating the capabilities of Artificial Neural Network implementations in recognizing extended sets of optical language symbols. The applications of this technique range from document digitizing and preservation to handwritten text recognition in handheld devices.

The classic difficulty of being able to correctly recognize even typed optical language symbols is the complex irregularity among pictorial representations of the same character due to variations in fonts, styles and size. This irregularity undoubtedly widens when one deals with handwritten characters.

Hence the conventional programming methods of mapping symbol images into matrices, analyzing pixel and/or vector data and trying to decide which symbol corresponds to which character would yield little or no realistic results. Clearly the needed methodology will be one that can detect ‘proximity’ of graphic representations to known symbols and make decisions based on this proximity. To implement such proximity algorithms in the conventional programming one needs to write endless code, one for each type of possible irregularity or deviation from the assumed output either in terms of pixel or vector parameters, clearly not a realistic fare.

An emerging technique in this particular application area is the use of Artificial Neural Network implementations with networks employing specific guides (learning rules) to update the links (weights) between their nodes. Such networks can be fed the data from the graphic analysis of the input picture and trained to output characters in one or another form. Specifically some network models use a set of desired outputs to compare with the output and compute an error to make use of in adjusting their weights. Such learning rules are termed as Supervised Learning.

One such network with supervised learning rule is the Multi-Layer Perceptron (MLP) model. It uses the Generalized Delta Learning Rule for adjusting its weights and can be trained for a set of input/desired output values in a number of iterations. The very nature of this particular model is that it will force the output to one of nearby values if a variation of input is fed to the network that it is not trained for, thus solving the proximity issue. Both concepts will be discussed in the introduction part of this report.

The project has employed the MLP technique mentioned and excellent results were obtained for a number of widely used font types. The technical approach followed in processing input images, detecting graphic symbols, analyzing and mapping the symbols and training the network for a set of desired Unicode characters corresponding to the input images are discussed in the subsequent sections. Even though the implementation might have some limitations in terms of functionality and robustness, the researcher is confident that it fully serves the purpose of addressing the desired objectives.

Introduction

I. Introduction

A. Artificial Neural Networks

Modeling systems and functions using neural network mechanisms is a relatively new and developing science in computer technologies. The particular area derives its basis from the way neurons interact and function in the natural animal brain, especially humans. The animal brain is known to operate in massively parallel manner in recognition, reasoning, reaction and damage recovery. All these seemingly sophisticated undertakings are now understood to be attributed to aggregations of very simple algorithms of pattern storage and retrieval. Neurons in the brain communicate with one another across special electrochemical links known as synapses. At a time one neuron can be linked to as many as 10,000 others although links as high as hundred thousands are observed to exist. The typical human brain at birth is estimated to house one hundred billion plus neurons. Such a combination would yield a synaptic connection of 10^{15}, which gives the brain its power in complex spatio-graphical computation.

Unlike the animal brain, the traditional computer works in serial mode, which is to mean instructions are executed only one at a time, assuming a uni-processor machine. The illusion of multitasking and real-time interactivity is simulated by the use of high computation speed and process scheduling. In contrast to the natural brain which communicates internally in electrochemical links, that can achieve a maximum speed in milliseconds range, the microprocessor executes instructions in the lower microseconds range. A modern processor such as the Intel Pentium-4 or AMD Opteron making use of multiple pipes and hyper-threading technologies can perform up to 20 MFloPs (Million Floating Point executions) in a single second.

It is the inspiration of this speed advantage of artificial machines, and parallel capability of the natural brain that motivated the effort to combine the two and enable performing complex 'Artificial Intelligence' tasks believed to be impossible in the past. Although artificial neural networks are currently implemented in the traditional serially operable computer, they still utilize the parallel power of the brain in a simulated manner.

Neural networks have seen an explosion of interest over the last few years, and are being successfully applied across an extraordinary range of problem domains, in areas as diverse as finance, medicine, engineering, geology and physics. Indeed, anywhere that there are problems of prediction, classification or control, neural networks are being introduced. This sweeping success can be attributed to a few key factors:

Power: Neural networks are very sophisticated modeling techniques capable of modeling extremely complex functions. In particular, neural networks are *nonlinear*. For many years linear modeling has been the commonly used technique in most modeling domains since linear models have well-known optimization strategies. Where the linear approximation was not valid (which was frequently the case) the models suffered accordingly. Neural networks also keep in check the *curse of dimensionality* problem that bedevils attempts to model nonlinear functions with large numbers of variables.

Ease of use: Neural networks *learn by example*. The neural network user gathers representative data, and then invokes *training algorithms* to automatically learn the structure of the data. Although the user does need to have some heuristic knowledge of how to select and prepare data, how to select an appropriate neural network, and how to interpret the results, the level of user knowledge needed to successfully apply neural networks is much lower than would be the case using (for example) some more traditional nonlinear statistical methods.

B. The Multi-Layer Perceptron Neural Network Model

To capture the essence of biological neural systems, an artificial *neuron* is defined as follows:

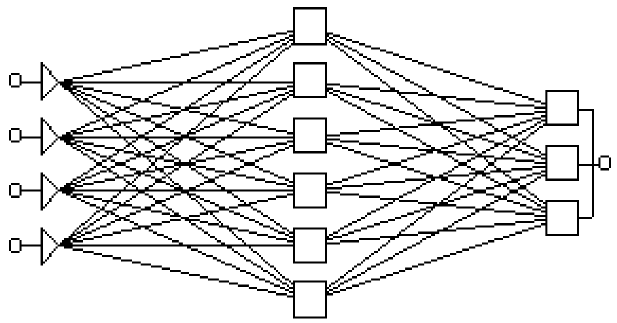
- It receives a number of inputs (either from original data, or from the output of other neurons in the neural network). Each input comes via a connection that has a strength (or *weight*); these weights correspond to synaptic efficacy in a biological neuron. Each neuron also has a single threshold value. The weighted sum of the inputs is formed, and the threshold subtracted, to compose the *activation* of the neuron (also known as the post-synaptic potential, or PSP, of the neuron).

- The activation signal is passed through an activation function (also known as a transfer function) to produce the output of the neuron.

If the step activation function is used (i.e., the neuron's output is 0 if the input is less than zero, and 1 if the input is greater than or equal to 0) then the neuron acts just like the biological neuron described earlier (subtracting the threshold from the weighted sum and comparing with zero is equivalent to comparing the weighted sum to the threshold). Actually, the step function is rarely used in artificial neural networks, as will be discussed. Note also that weights can be negative, which implies that the synapse has an inhibitory rather than excitatory effect on the neuron: inhibitory neurons are found in the brain.

This describes an individual neuron. The next question is: how should neurons be connected together? If a network is to be of any use, there must be inputs (which carry the values of variables of interest in the outside world) and outputs (which form predictions, or control signals). Inputs and outputs correspond to sensory and motor nerves such as those coming from the eyes and leading to the hands. However, there also can be hidden neurons that play an internal role in the network. The input, hidden and output neurons need to be connected together.

A typical feedforward network has neurons arranged in a distinct layered topology. The input layer is not really neural at all: these units simply serve to introduce the values of the input variables. The hidden and output layer neurons are each connected to all of the units in the preceding layer. Again, it is possible to define networks that are partially-connected to only some units in the preceding layer; however, for most applications fully-connected networks are better.



The Multi-Layer Perceptron Neural Network is perhaps the most popular network architecture in use today. The units each perform a biased weighted sum of their inputs and pass this activation level through an activation function to produce their output, and the units are arranged in a layered feedforward topology. The network thus has a simple interpretation as a form of input-output model, with the weights and thresholds (biases) the free parameters of the model. Such networks can model functions of almost arbitrary complexity, with the number of layers, and the number of units in each layer, determining the function complexity. Important issues in Multilayer Perceptrons (MLP) design include specification of the number of hidden layers and the number of units in each layer.

Fig.1 A Typical Feedforward Network

Most common activation functions are the logistic and hyperbolic tangent sigmoid functions. The project used the **hyperbolic tangent function:** $f(x) = \frac{2}{(1+e^{-\lambda x})} - 1$ **and derivative:** . $f'(x) = f(x)(1-f(x))$

C. Optical Language Symbols

Several languages are characterized by having their own written symbolic representations (characters). These characters are either a delegate of a specific audioglyph, accent or whole words in some cases. In terms of structure world language characters manifest various levels of organization. With respect to this structure there always is an issue of compromise between ease of construction and space conservation. Highly structured alphabets like the Latin set enable easy construction of language elements while forcing the use of additional space. Medium structure alphabets like the Ethiopic (Ge'ez) conserve space due to representation of whole audioglyphs and tones in one symbol, but dictate the necessity of having extended sets of symbols and thus a difficult level of use and learning. Some alphabets, namely the oriental alphabets, exhibit a very low amount of structuring that whole words are delegated by single symbols. Such languages are composed of several thousand symbols and are known to need a learning cycle spanning whole lifetimes.

Representing alphabetic symbols in the digital computer has been an issue from the beginning of the computer era. The initial efforts of this representation (encoding) was for the alphanumeric set of the Latin alphabet and some common mathematical and formatting symbols. It was not until the 1960's that a formal encoding standard was prepared and issued by the American computer standards bureau ANSI and named the ASCII Character set. It is composed of and 8-bit encoded computer symbols with a total of 256 possible unique symbols. In some cases certain combination of keys were allowed to form 16-bit words to represent extended symbols. The final rendering of the characters on the user display was left for the application program in order to allow for various fonts and styles to be implemented.

At the time, the 256+ encoded characters were thought of suffice for all the needs of computer usage. But with the emergence of computer markets in the non-western societies and the internet era, representation of a further set of alphabets in the computer was necessitated. Initial attempts to meet this requirement were based on further combination of ASCII encoded characters to represent the new symbols. This however led to a deep chaos in rendering characters especially in web pages since the user had to choose the correct encoding on the browser. Further difficulty was in coordinating the usage of key combinations between different implementers to ensure uniqueness.

It was in the 1990s that a final solution was proposed by an independent consortium to extend the basic encoding width to 16-bit and accommodate up to 65,536 unique symbols. The new encoding was named Unicode due to its ability to represent all the known symbols in a single encoding. The first 256 codes of this new set were reserved for the ASCII set in order to maintain compatibility with existing systems. ASCII characters can be extracted form a Unicode word by reading the lower 8 bits and ignoring the rest or vise versa, depending on the type of endian (big or small) used.

The Unicode set is managed by the Unicode consortium which examines encoding requests, validate symbols and approve the final encoding with a set of unique 16-bit codes. The set still has a huge portion of it non-occupied waiting to accommodate any upcoming requests. Ever since it's founding, popular computer hardware and software manufacturers like Microsoft have accepted and supported the Unicode effort.

As an aside the researcher mentions that the Ethiopic alphabet (commonly known as Ge'ez) is represented from 1200^{H} – $137F^{H}$ in the Unicode set.

II. Technical Overview

A. Introduction

The operations of the network implementation in this project can be summarized by the following steps:

- **Training phase**
    - Analyze image for characters
    - Convert symbols to pixel matrices
    - Retrieve corresponding desired output character and convert to Unicode
    - Lineraize matrix and feed to network

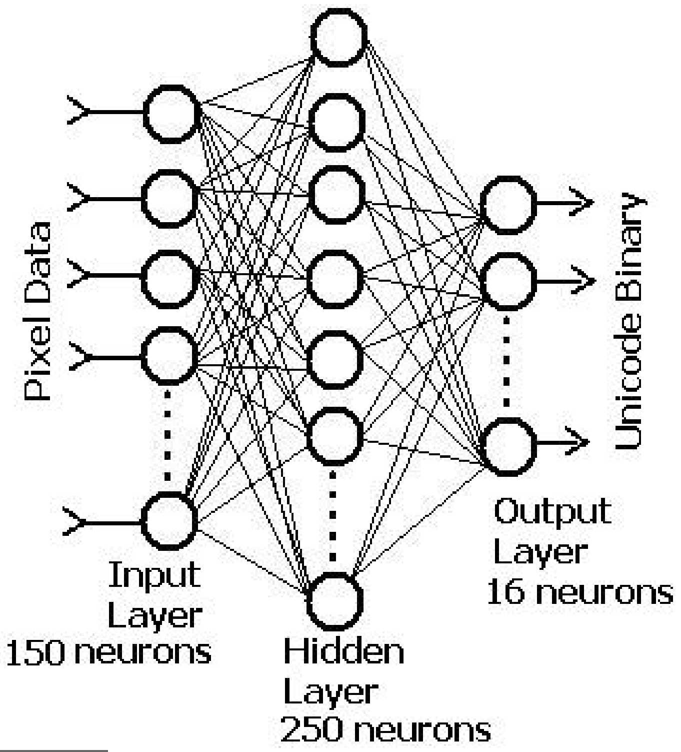
    - Compute output
    - Compare output with desired output Unicode value and compute error
    - Adjust weights accordingly and repeat process until preset number of iterations

- **Testing phase**
    - Analyze image for characters
    - Convert symbols to pixel matrices
    - Compute output
    - Display character representation of the Unicode output

Essential components of the implementation are:

- Formation of the network and weight initialization routine
- Pixel analysis of images for symbol detection
- Loading routines for training input images and corresponding desired output characters in special files named character trainer sets (*.cts)
- Loading and saving routines for trained network (weight values)
- Character to binary Unicode and vice versa conversion routines
- Error, output and weight calculation routines

1. Network Formation



The MLP Network implemented for the purpose of this project is composed of 3 layers, one input, one hidden and one output.

The input layer constitutes of 150 neurons which receive pixel binary data from a 10x15 symbol pixel matrix. The size of this matrix was decided taking into consideration the average height and width of character image that can be mapped without introducing any significant pixel noise.

The hidden layer constitutes of 250 neurons whose number is decided on the basis of optimal results on a trial and error basis.

The output layer is composed of 16 neurons corresponding to the 16-bits of Unicode encoding.

To initialize the weights a random function was used to assign an initial random number which lies between two preset integers named **±weight_bias**. The weight bias is selected from trial and error observation to correspond to average weights for quick convergence.

Fig. 2 The Project MLP Network

2. Symbol image detection

The process of image analysis to detect character symbols by examining pixels is the core part of input set preparation in both the training and testing phase. Symbolic extents are recognized out of an input image file based on the color value of individual pixels, which for the limits of this project is assumed to be either black **RGB(255,0,0,0)** or white **RGB(255,255,255,255)**. The input images are assumed to be in bitmap form of any resolution which can be mapped to an internal bitmap object in the Microsoft Visual Studio environment. The procedure also assumes the input image is composed of only characters and any other type of bounding object like a boarder line is not taken into consideration.

The procedure for analyzing images to detect characters is listed in the following algorithms:

i. Determining character lines

Enumeration of character lines in a character image ('page') is essential in delimiting the bounds within which the detection can proceed. Thus detecting the next character in an image does not necessarily involve scanning the whole image all over again.

Algorithm:

1. start at the first x and first y pixel of the image pixel(0,0), Set number of lines to 0
2. scan up to the width of the image on the same y-component of the image
    a. if a black pixel is detected register y as top of the first line
    b. if not continue to the next pixel
    c. if no black pixel found up to the width increment y and reset x to scan the next horizontal line
3. start at the top of the line found and first x-component pixel(0,line_top)
4. scan up to the width of the image on the same y-component of the image
    a. if no black pixel is detected register y-1 as bottom of the first line. Increment number of lines
    b. if a black pixel is detected increment y and reset x to scan the next horizontal line
5. start below the bottom of the last line found and repeat steps 1-4 to detect subsequent lines
6. If bottom of image (image height) is reached stop.

ii. Detecting Individual symbols

Detection of individual symbols involves scanning character lines for orthogonally separable images composed of black pixels.

Algorithm:

1. start at the first character line top and first x-component

2. scan up to image width on the same y-component
    a. if black pixel is detected register y as top of the first line
    b. if not continue to the next pixel
3. start at the top of the character found and first x-component, pixel(0,character_top)
4. scan up to the line bottom on the same x-component
    a. if black pixel found register x as the left of the symbol
    b. if not continue to the next pixel
    c. if no black pixels are found increment x and reset y to scan the next vertical line
5. start at the left of the symbol found and top of the current line, pixel(character_left, line_top)
6. scan up to the width of the image on the same x-component
    a. if no black characters are found register x-1 as right of the symbol
    b. if a black pixel is found increment x and reset y to scan the next vertical line
7. start at the bottom of the current line and left of the symbol, pixel(character_left,line_bottom)
8. scan up to the right of the character on the same y-component
    a. if a black pixel is found register y as the bottom of the character
    b. if no black pixels are found decrement y and reset x to scan the next vertical line

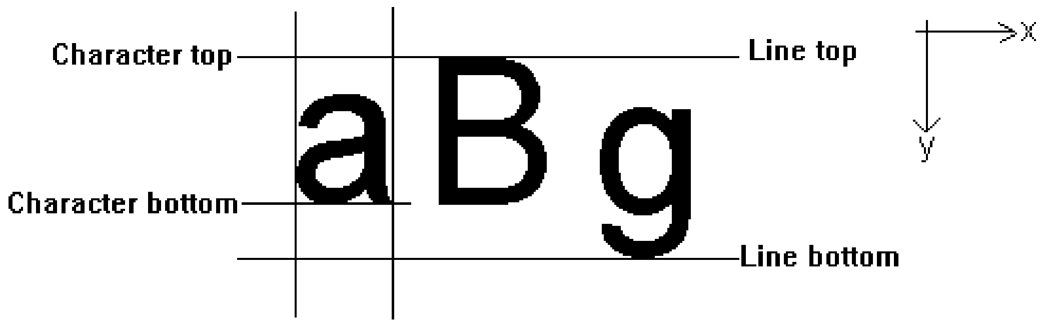


Fig 3. Line and Character boundary detection

From the procedure followed and the above figure it is obvious that the detected character bound might not be the actual bound for the character in question. This is an issue that arises with the height and bottom alignment irregularity that exists with printed alphabetic symbols. Thus a line top does not necessarily mean top of all characters and a line bottom might not mean bottom of all characters as well.

Hence a confirmation of top and bottom for the character is needed.

An optional confirmation algorithm implemented in the project is:

A. start at the top of the current line and left of the character
B. scan up to the right of the character

   1. if a black pixels is detected register y as the confirmed top
   2. if not continue to the next pixel
   3. if no black pixels are found increment y and reset x to scan the next horizontal line

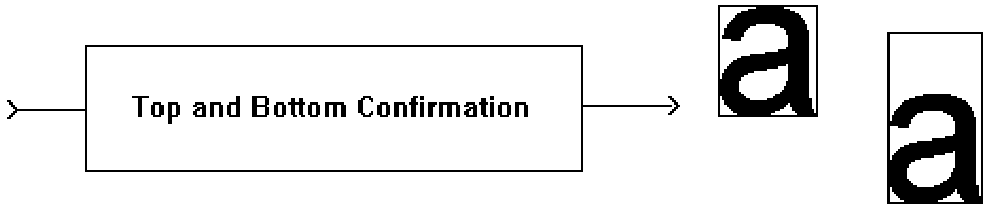


Fig 4. Confirmation of Character boundaries

3. Symbol Image Matrix Mapping

The next step is to map the symbol image into a corresponding two dimensional binary matrix. An important issue to consider here will be deciding the size of the matrix. If all the pixels of the symbol are mapped into the matrix, one would definitely be able to acquire all the distinguishing pixel features of the symbol and minimize overlap with other symbols. However this strategy would imply maintaining and processing a very large matrix (up to 1500 elements for a 100x150 pixel image). Hence a reasonable tradeoff is needed in order to minimize processing time which will not significantly affect the separability of the patterns. The project employed a sampling strategy which would map the symbol image into a 10x15 binary matrix with only 150 elements. Since the height and width of individual images vary, an adaptive sampling algorithm was implemented. The algorithm is listed below:

Algorithm:

a. For the width (initially 20 elements wide)

1. Map the first (0,y) and last (width,y) pixel components directly to the first (0,y) and last (20,y) elements of the matrix
2. Map the middle pixel component (width/2,y) to the 10th matrix element
3. subdivide further divisions and map accordingly to the matrix

b. For the height (initially 30 elements high)

1. Map the first x,(0) and last (x,height) pixel components directly to the first (x,0) and last (x,30) elements of the matrix
2. Map the middle pixel component (x,height/2) to the 15th matrix element

3. subdivide further divisions and map accordingly to the matrix

c. Further reduce the matrix to 10x15 by sampling by a factor of 2 on both the width and the height

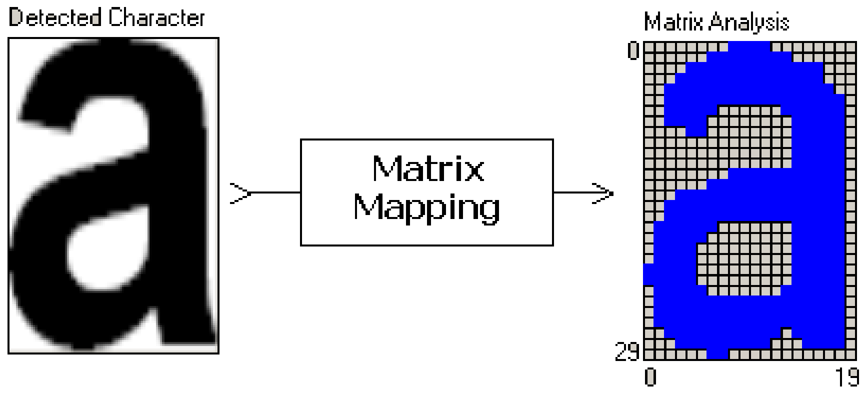
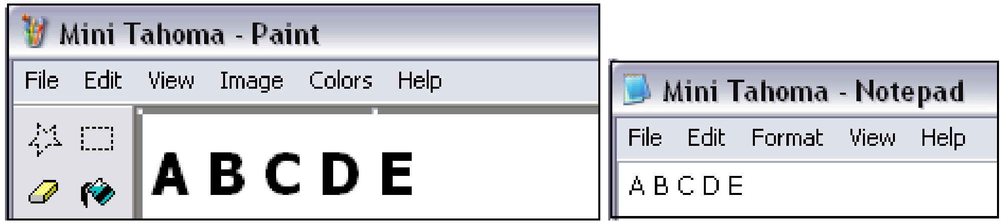


Fig. 5 Mapping symbol images onto a binary matrix

In order to be able to feed the matrix data to the network (which is of a single dimension) the matrix must first be linearized to a single dimension. This is accomplished with a simple routine with the following algorithm:

1. start with the first matrix element (0,0)
2. increment x keeping y constant up to the matrix width
    a. map each element to an element of a linear array (increment array index)
    b. if matrix width is reached reset x, increment y
3. repeat up to the matrix height (x,y)=(width, height)

Hence the linear array is our input vector for the MLP Network. In a training phase all such symbols from the trainer set image file are mapped into their own linear array and as a whole constitute an input space. The trainer set would also contain a file of character strings that directly correspond to the input symbol images to serve as the desired output of the training. A sample mini trainer set is shown below:



A B C D E

Fig. 6 Input Image and Desired output text files for the sample Mini-Tahoma trainer set

B. Training

Once the network has been initialized and the training input space prepared the network is ready to be trained. Some issues that need to be addressed upon training the network are:

- How chaotic is the input space? A chaotic input varies randomly and in extreme range without any predictable flow among its members.
- How complex are the patterns for which we train the network? Complex patterns are usually characterized by feature overlap and high data size.
- What should be used for the values of:
  - Learning rate
  - Sigmoid slope
  - Weight bias
- How many Iterations (Epochs) are needed to train the network for a given number of input sets?
- What error threshold value must be used to compare against in order to prematurely stop iterations if the need arises?

Alphabetic optical symbols are one of the most chaotic input sets in pattern recognitions studies. This is due to the unpredictable nature of their pictorial representation seen from the sequence of their order. For instance the Latin alphabetic consecutive character ‘A’ and ‘B’ have little similarity in feature when represented in their pictorial symbolic form. The figure below demonstrates the point of chaotic and non-chaotic sequence with the Latin and some factious character set:

Fig. 7 Example of chaotic and non-chaotic symbol sequences

The complexity of the individual pattern data is also another issue in character recognition. Each symbol has a large number of distinct features that need to be accounted for in order to correctly recognize it. Elimination of some features might result in pattern overlap and the minimum amount of data required makes it one of the most complex classes of input space in pattern recognition.

Other than the known issues mentioned, the other numeric parameters of the network are determined in real time. They also vary greatly from one implementation to another according to the number of input symbols fed and the network topology.

For the purpose of this project the parameters use are:

- Learning rate = 150
- Sigmoid Slope = 0.014

- Weight bias = 30 (determined by trial and error)
- Number of Epochs = 300-600 (depending on the complexity of the font types)
- Mean error threshold value = 0.0002 (determined by trial and error)

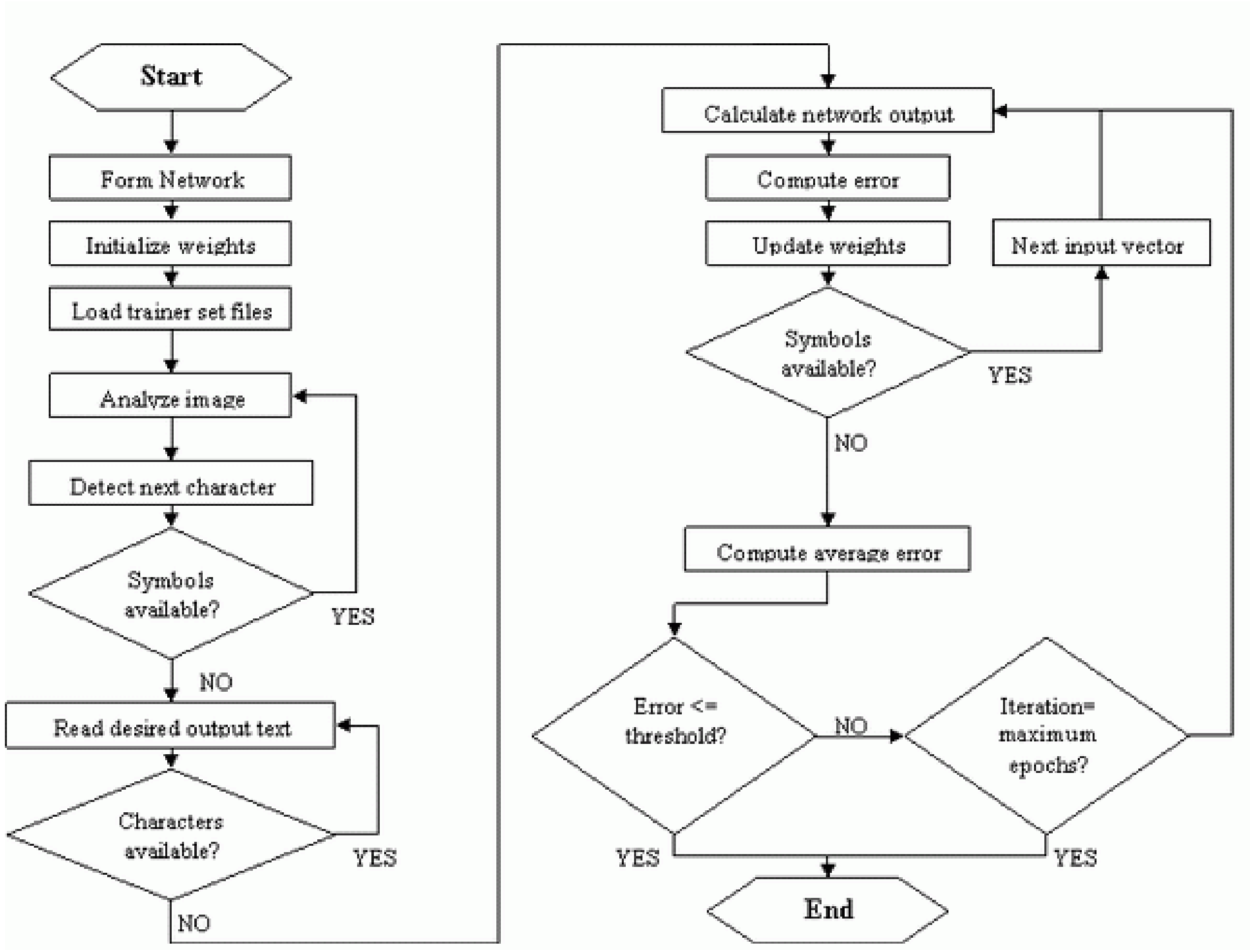
Algorithm:

The training routine implemented the following basic algorithm

1. Form network according to the specified topology parameters
2. Initialize weights with random values within the specified ±weight_bias value
3. load trainer set files (both input image and desired output text)
4. analyze input image and map all detected symbols into linear arrays
5. read desired output text from file and convert each character to a binary Unicode value to store separately
6. for each character :
    a. calculate the output of the feed forward network
    b. compare with the desired output corresponding to the symbol and compute error
    c. back propagate error across each link to adjust the weights
7. move to the next character and repeat step 6 until all characters are visited
8. compute the average error of all characters
9. repeat steps 6 and 8 until the specified number of epochs
    a. Is error threshold reached? If so abort iteration
    b. If not continue iteration

Flowchart:

The flowchart representation of the algorithm is illustrated below



C. Testing

The testing phase of the implementation is simple and straightforward. Since the program is coded into modular parts the same routines that were used to load, analyze and compute network parameters of input vectors in the training phase can be reused in the testing phase as well.

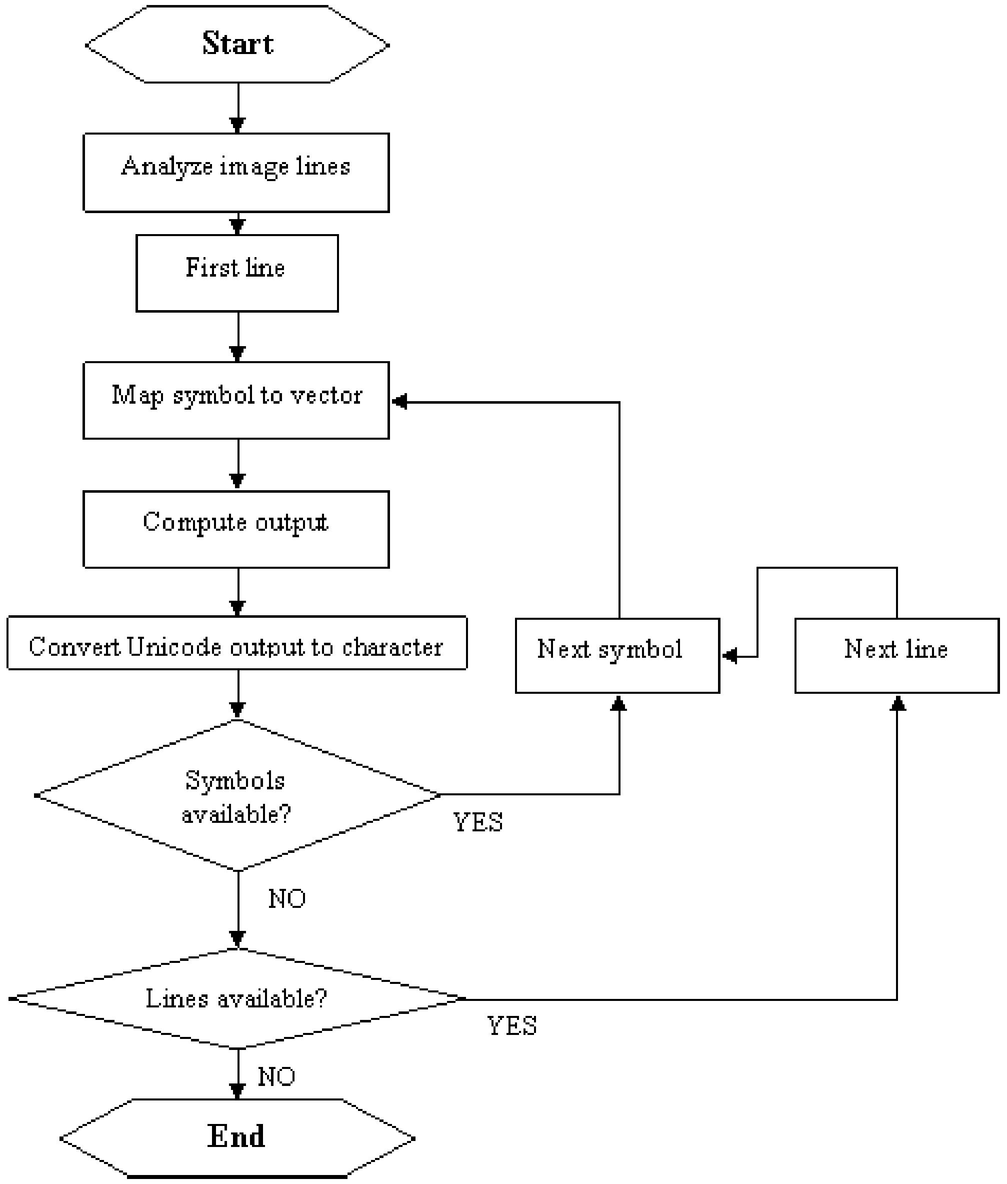
The basic steps in testing input images for characters can be summarized as follows:

Algotithm:

- load image file
- analyze image for character lines
- for each character line detect consecutive character symbols
    - analyze and process symbol image to map into an input vector

- feed input vector to network and compute output
- convert the Unicode binary output to the corresponding character and render to a text box

Flowchart:



III. Results and Discussion

The network has been trained and tested for a number of widely used font type in the Latin alphabet. Since the implementation of the software is open and the program code is scalable, the inclusion of more number of fonts from any typed language alphabet is straight forward.

The necessary steps are preparing the sequence of input symbol images in a single image file (*.bmp [bitmap] extension), typing the corresponding characters in a text file (*.cts [character trainer set] extension) and saving the two in the same folder (both must have the same file name except for their extensions). The application will provide a file opener dialog for the user to locate the *.cts text file and will load the corresponding image file by itself.

Although the results listed in the subsequent tables are from a training/testing process of symbol images created with a 72pt. font size the use of any other size is also straight forward by preparing the input/desired output set as explained. The application can be operated with symbol images as small as 20pt font size.

Note: Due to the random valued initialization of weight values results listed represent only typical network performance and exact reproduction might not be obtained with other trials.

A. Results for variation in number of Epochs

Number of characters=90, Learning rate=150, Sigmoid slope=0.014

Font Type	**300**		**600**		**800**	
	№ of wrong characters	% Error	№ of wrong characters	% Error	№ of wrong characters	% Error
Latin Arial	4	4.44	3	3.33	1	1.11
Latin Tahoma	1	1.11	0	0	0	0
Latin Times Roman	0	0	0	0	1	1.11

B. Results for variation in number of Input characters

Number of Epochs=100, Learning rate=150, Sigmoid slope=0.014

	20		**50**		**90**	

Font Type	№ of wrong characters	% Error	№ of wrong characters	% Error	№ of wrong characters	% Error
Latin Arial	0	0	6	12	11	12.22
Latin Tahoma	0	0	3	6	8	8.89
Latin Times Roman	0	0	2	4	9	10

C. Results for variation in Learning rate parameter

Number of characters=90, Number of Epochs=600, Sigmoid slope=0.014

	50		**100**		**120**	
Font Type	№ of wrong characters	% Error	№ of wrong characters	% Error	№ of wrong characters	% Error
Latin Arial	82	91.11	18	20	3	3.33
Latin Tahoma	56	62.22	11	12.22	1	1.11
Latin Times Roman	77	85.56	15	16.67	0	0

D. Performance Observation

1. Influence of parameter variation

i. Increasing the number of iterations has generally a positive proportionality relation to the performance of the network. However in certain cases further increasing the number of epochs has an adverse effect of introducing more number of wrong recognitions. This partially can be attributed to the high value of learning rate parameter as the network approaches its optimal limits and further weight updates result in bypassing the optimal state. With further iterations the network will try to 'swing' back to the desired state and back again continuously, with a good chance of missing the optimal state at the final epoch. This phenomenon is known as over learning.

ii. The size of the input states is also another direct factor influencing the performance. It is natural that the more number of input symbol set the network is required to be trained for the more it is susceptible for error. Usually the complex and large sized input sets require a large topology network with more number of iterations. For the above maximum set number of 90 symbols the optimal topology reached was one hidden layer of 250 neurons.

iii. Learning rate parameter variation also affects the network performance for a given limit of iterations. The less the value of this parameter, the lower the value with which the network updates its weights. This intuitively implies that it will be less likely to face the over learning difficulty discussed above since it will be updating its links slowly and in a more refined manner. But unfortunately this would also imply more number of iterations is required to reach its optimal state. Thus a trade of is needed in order to optimize the overall network performance. The optimal value decided upon for the learning parameter is 150.

2. Pictorial representation overlap anomalies

One can easily observe from the results listing that the entry for the 'Latin Arial' font type has, in general, the lowest performance among its peers. This has been discovered to arise due to an overlap in the pictorial representation of two of its symbols, namely the upper case letter 'I' ('I' in Times Roman) and the lower case letter 'l' ('l' in Times Roman).

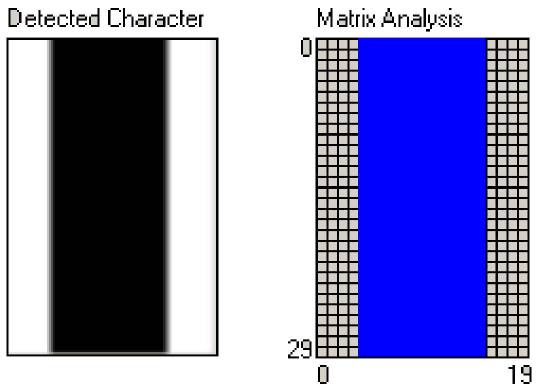
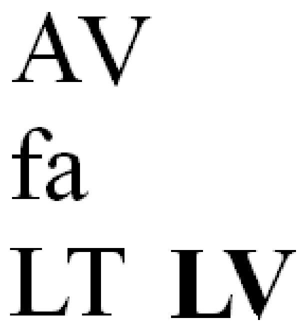


Fig. 8 Matrix analysis for both lower case 'l' (006C^{h}) and upper case 'I' (0049^{h}) of the Arial font.

This would definitely present a logically non-separable recognition task to the network as the training set will be instructing it to output one state for a symbolic image and at some other time another state for the same image. This will be disturbing not only the output vectors of the two characters but also nearby states as well as can be seen in the number of wrong characters. The best state the network can reach in such a case is to train itself to output one vector for both inputs, necessitating a wrong state to one of the output. Still this optimal state can be reached only with more number of iterations which for this implementation was 800. At such high number of epochs the other sets tend to jump into over learning states as discussed above.

3. Orthogonal inseparability

Some symbol sequences are orthogonally inseparable. This is to mean there can not be a vertical line that passes between the two symbols without crossing bitmap areas of either. Such images could not be processed for individual symbols within the limits of the project since it requires complex image processing algorithms. Some cases are presented below:



Upper case ‘A’ followed by upper case ‘V’

Lower case 'f' followed by some short characters

Upper case 'L' followed by characters with side extensions

Fig 9. Some orthogonally inseparable symbolic combinations in the Latin alphabet

V. Appendix

A. ASCII Table of codes

1. Character Codes Chart 1 (0-127)

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20		64	40	@	96	60	`
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	"	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	'	71	47	G	103	67	g
^H	8	08		BS	40	28	(	72	48	H	104	68	h
^I	9	09		HT	41	29	)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B		ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	|
^]	29	1D		GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂*

* ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.

2. Character Codes Chart 2 (128-255)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

B. Code Listing [MS Visual C#.NET]

1. Image line Identification

Collapse | Copy Code

```
public void identify_lines()
{
    int y=image_start_pixel_y;
    int x=image_start_pixel_x;
    bool no_black_pixel;
    int line_number=0;
    line_present=true;

    while(line_present)
    {
        x=image_start_pixel_x;

        while(Convert.ToString (input_image.GetPixel (x,y))==
            "Color [A=255, R=255, G=255, B=255]")
        {
            x++;
            if(x==input_image_width)
            {
                x=image_start_pixel_x;
                y++;
            }
            if(y>=input_image_height)
            {
                line_present=false;
                break;
            }
        }
        if(line_present)
        {
            line_top[line_number]=y;
            no_black_pixel=false;
            while(no_black_pixel==false)
            {
                y++;
                no_black_pixel=true;
                for(x=image_start_pixel_x;x<input_image_width;x++)
                    if((Convert.ToString (input_image.GetPixel (x,y))==
                        "Color [A=255, R=0, G=0, B=0]"))

                no_black_pixel=false;
            }
            line_bottom[line_number]=y-1;
            line_number++;
        }
    }
    number_of_lines=line_number;
}
```

2. Character Symbol Detection

Collapse | Copy Code

```
public void get_character_bounds()
{
    int x=image_start_pixel_x;
    int y=image_start_pixel_y;
    bool no_black_pixel=false;
    if(y<=input_image_height && x<=input_image_width)
    {
        while(Convert.ToString (input_image.GetPixel (x,y))==
            "Color [A=255, R=255, G=255, B=255]")
        {
            x++;
            if(x==input_image_width)
            {
                x=image_start_pixel_x;  y++;
            }
            if(y>=line_bottom[current_line])
            {
                character_present=false; break;
            }
        }
        if(character_present)
        {
            top=y;
            x=image_start_pixel_x; y=image_start_pixel_y;
            while(Convert.ToString (input_image.GetPixel (x,y))==
                "Color [A=255, R=255, G=255, B=255]")
            {
                y++;
                if(y==line_bottom[current_line])
                {
                    y=image_start_pixel_y;  x++;
                }
                if(x>input_image_width) break;
            }
            if(x<input_image_width) left=x;
            no_black_pixel=true;
            y=line_bottom[current_line]+2;
            while(no_black_pixel==true)
            {
                y--;
                for(x=image_start_pixel_x;x<input_image_width;x++)
                    if((Convert.ToString (input_image.GetPixel (x,y))==
                        "Color [A=255, R=0, G=0, B=0]"))
```

```
                             no_black_pixel=false;
               }
               bottom=y;
               no_black_pixel=false;
               x=left+10;
               while(no_black_pixel==false)
               {
                    x++; no_black_pixel=true;
                    for(y=image_start_pixel_y;y<line_bottom[current_line];y++)
                         if((Convert.ToString (input_image.GetPixel (x,y))==
                             "Color [A=255, R=0, G=0, B=0]"))
                             no_black_pixel=false;
               }
               right=x-1;
          }
     }
```

3. Network Forming

Collapse | Copy Code

```
public void form_network()
{
       layers[0]=number_of_input_nodes;
       layers[number_of_layers-1]=number_of_output_nodes;
       for(int i=1;i<number_of_layers-1;i++)
              layers[i]=maximum_layers;
}
```

4. Weight initialization

Collapse | Copy Code

```
public void initialize_weights()
{
     for(int i=1;i<number_of_layers;i++)
          for(int j=0;j<layers[i];j++)
               for(int k=0;k<layers[i-1];k++)
                    weight[i,j,k]=(float)(rnd.Next(-weight_bias,weight_bias));
}
```

5. Network Training

Collapse | Copy Code

```
public void train_network()
{
     int set_number;
```

```
    float average_error=0.0F;
    progressBar1.Maximum =epochs;

    for(int epoch=0;epoch<=epochs;epoch++)
    {
        average_error=0.0F;
        for(int i=0;i<number_of_input_sets;i++)
        {
            set_number=rnd.Next(0,number_of_input_sets);
            get_inputs(set_number);
            get_desired_outputs(set_number);
            calculate_outputs();
            calculate_errors();
            calculate_weights();
            average_error=average_error+get_average_error();
        }

        progressBar1.PerformStep ();
        label15.Text = epoch.ToString ();
        label15.Update ();
        average_error=average_error/number_of_input_sets;

        if(average_error<error_threshold)
        {
            epoch=epochs+1;
            progressBar1.Value =progressBar1.Maximum;
            label22.Text ="<"+error_threshold.ToString ();
            label22.Update ();
        }
    }
}
```

6. Output Computation

⊟ Collapse | Copy Code

```
public void calculate_outputs()
{
    float f_net;
    int number_of_weights;
    for(int i=0;i<number_of_layers;i++)
        for(int j=0;j<layers[i];j++)
        {
            f_net=0.0F;
            if(i==0) number_of_weights=1;
            else number_of_weights=layers[i-1];
            for(int k=0;k<number_of_weights;k++)
                if(i==0)
                    f_net=current_input[j];
                else
                    f_net=f_net+node_output[i-1,k]*weight[i,j,k];
                node_output[i,j]=sigmoid(f_net);
            }
}
```

7. Activation functions

⊟ Collapse | Copy Code

```
public float sigmoid(float f_net)
{
      float result=(float)((2/(1+Math.Exp(-1*slope*f_net)))-1); // Bipolar
      return result;
}

public float sigmoid_derivative(float result)
{
      float derivative=(float)(0.5F*(1-Math.Pow(result,2))); // Bipolar
      return derivative;
}
```

8. Error Computation

⊟ Collapse | Copy Code

```
public void calculate_errors()
{
    float sum=0.0F;
    for(int i=0;i<number_of_output_nodes;i++)
        error[number_of_layers-1,i] =
            (float)((desired_output[i]-node_output[number_of_layers-1,i])*
            sigmoid_derivative(node_output[number_of_layers-1,i]));

    for(int i=number_of_layers-2;i>=0;i--)
        for(int j=0;j<layers[i];j++)
        {
            sum=0.0F;
            for(int k=0;k<layers[i+1];k++)
                sum=sum+error[i+1,k]*weight[i+1,k,j];
            error[i,j] = (float)(sigmoid_derivative(node_output[i,j])*sum);
        }
}
```

9. Weight Update

⊟ Collapse | Copy Code

```
public void calculate_weights()
{
    for(int i=1;i<number_of_layers;i++)
        for(int j=0;j<layers[i];j++)
            for(int k=0;k<layers[i-1];k++)
            {
                weight[i,j,k] = (float)(weight[i,j,k] +
                    learning_rate*error[i,j]*node_output[i-1,k]);
            }
}
```

10. Unicode to character and vice versa conversion

⊟ Collapse | Copy Code

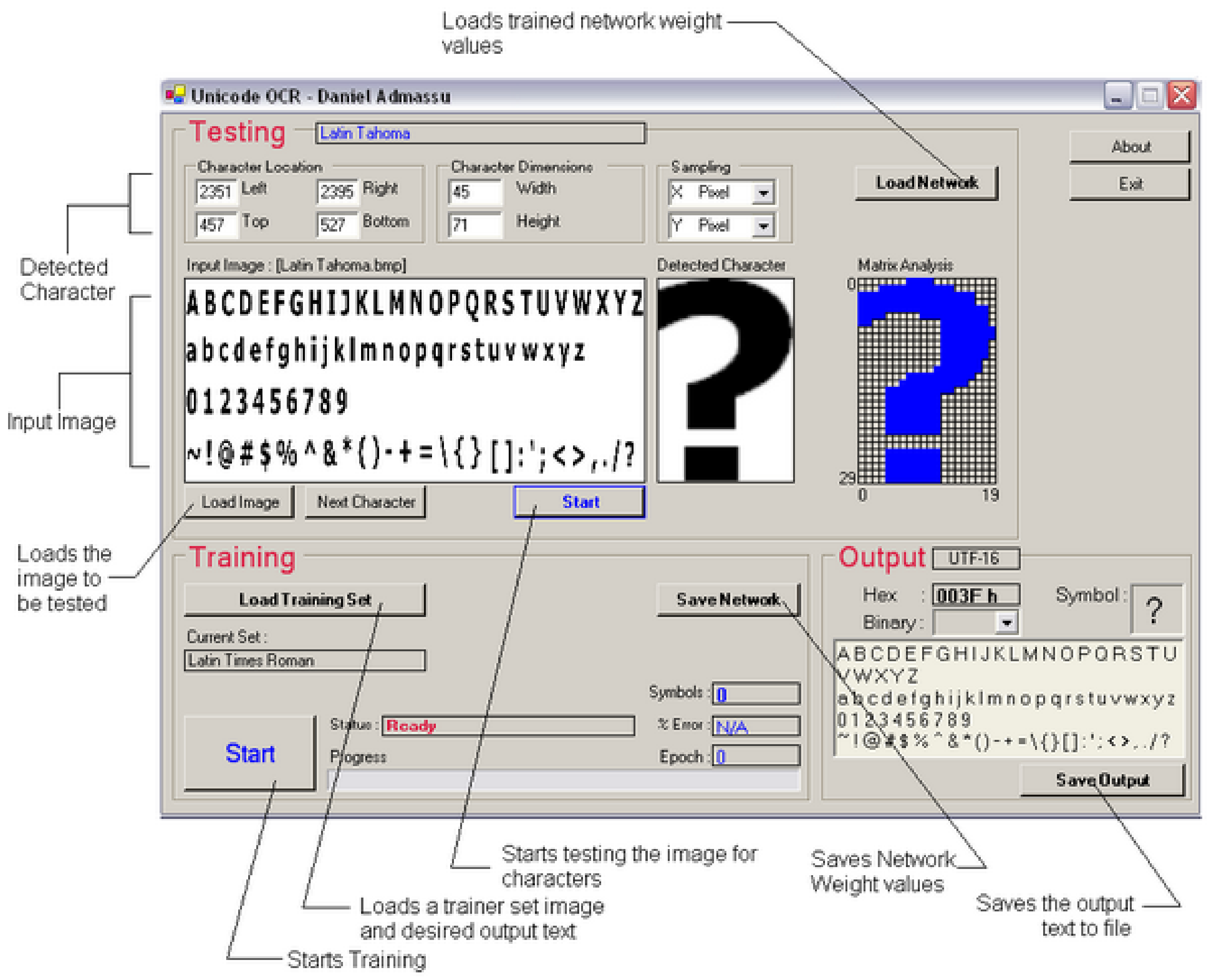
```
public void character_to_unicode(string character)
{
      int byteCount = unicode.GetByteCount(character.ToCharArray());
      byte[] bytes = new Byte[byteCount];
      bytes= unicode.GetBytes(character);
      BitArray bits = new BitArray( bytes );
      System.Collections.IEnumerator bit_enumerator = bits.GetEnumerator();
      int bit_array_length = bits.Length;
      bit_enumerator.Reset ();

      for(int i=0;i<bit_array_length;i++)
      {
            bit_enumerator.MoveNext();
            if(bit_enumerator.Current.ToString()=="True")
                  desired_output_bit[i]=1;
            else
                  desired_output_bit[i]=0;
      }
}

public char unicode_to_character()
{
      int dec=binary_to_decimal();
      Byte[] bytes = new Byte[2];
      bytes[0]=(byte)(dec);
      bytes[1]=0;
      int charCount = unicode.GetCharCount(bytes);
      char[] chars = new Char[charCount];
      chars=unicode.GetChars(bytes);
      return chars[0];
}
```

C. Software User Interface

© 2006, Daniel Admassu


Loads trained network weight values
Detected Character
Input Image
Loads the image to be tested
Starts testing the image for characters
Loads a trainer set image and desired output text
Starts Training
Saves Network Weight values
Saves the output text to file
Unicode OCR - Daniel Admassu
Testing
Load Network
About
Exit
Load Image
Next Character
Start
Training
Load Training Set
Save Network
Output
Save Output

Testing: Latin Tahoma

Character Location: 2351 Left, 2395 Right, 457 Top, 527 Bottom

Character Dimensions: 45 Width, 71 Height

Sampling: X Pixel, Y Pixel

Input Image : [Latin Tahoma.bmp]

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
~!@#$%^&*()-+=\{}[]:';<>,./?
```

Detected Character: ?

Matrix Analysis: 0, 29, 0, 19

Current Set : Latin Times Roman

Status : Ready

Progress

Symbols : 0

% Error : N/A

Epoch : 0

Output: UTF-16

Hex : 003F h

Binary :

Symbol : ?

```
ABCDEFGHIJKLMNOPQRSTU
VWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
~!@#$%^&*()-+=\{}[]:';<>,./?
```

D. Sample Network File [Latin Times Roman.ann]

⊟ Collapse | Copy Code

```
Unicode OCR ANN Weight values. © 2006 Daniel Admassu.
Network Name        = Latin Times Roman
Hidden Layer Size   = 250
Number of Patterns  = 90
Number of Epochs    = 300
```

```
Learning Rate      = 150
Sigmoid Slope      = 0.014
Weight Bias        = 30

Weight[1 , 0 , 0] = -344.8769
Weight[1 , 0 , 1] = -490.7207
Weight[1 , 0 , 2] = -739.5387
Weight[1 , 0 , 3] = -401.8878
Weight[1 , 0 , 4] = 183.874
Weight[1 , 0 , 5] = 167.6926
Weight[1 , 0 , 6] = -29.25896
Weight[1 , 0 , 7] = -42.90004
Weight[1 , 0 , 8] = 443.4576
Weight[1 , 0 , 9] = -208.5291
Weight[1 , 0 , 10] = -264.9088
Weight[1 , 0 , 11] = 16.65468
Weight[1 , 0 , 12] = 401.0692
Weight[1 , 0 , 13] = 409.3685
Weight[1 , 0 , 14] = -161.1487
Weight[1 , 0 , 15] = 89.9006
Weight[1 , 0 , 16] = -851.1705
Weight[1 , 0 , 17] = -96.01544
Weight[1 , 0 , 18] = 290.6281
Weight[1 , 0 , 19] = 64.17367
.
.
.
Weight[2 , 15 , 237] = -5.202068
Weight[2 , 15 , 238] = -16.83099
Weight[2 , 15 , 239] = 23.25441
Weight[2 , 15 , 240] = -24.27003
Weight[2 , 15 , 241] = -4.82569
Weight[2 , 15 , 242] = 36.25824
Weight[2 , 15 , 243] = -28.49981
Weight[2 , 15 , 244] = -8.105846
Weight[2 , 15 , 245] = -1.679604
Weight[2 , 15 , 246] = -3.154837
Weight[2 , 15 , 247] = -11.18855
Weight[2 , 15 , 248] = 8.335608
Weight[2 , 15 , 249] = 30.13228
```

VI. Reference

1. **Artificial Intelligence and cognitive science**
   © 2006, Nils J. Nilsson
   Stanford AI Lab
   http://ai.stanford.edu/~nilsson
2. **Off-line Handwriting Recognition Using Artificial Neural Networks**

© 2000, Andrew T. Wilson
University of Minnesota, Morris
http://wilsonat@mrs.umn.edu

3. **Using Neural Networks to Create an Adaptive Character Recognition System**
   © 2002, Alexander J. Faaborg
   Cornell University, Ithaca NY
4. **Hand-Printed Character Recognizer using Neural Network**
   © 2000, Shahzad Malik
5. **Neural Networks and Fuzzy Logic**.
   © 1995, Rao, V., Rao, H.
   MIS Press, New York
6. **Neural Networks**
   © 2003, StatSoft Inc.
   http://www.statsoft.com/textbook/stneunet.html

License

Share

EMA

About the Author

Daniel Admassu
Web Developer
Ethiopia

Name: Daniel Admassu
Nationality: Ethiopian (hope someone has heard of the place)
E-Mail : daniel_admasu@yahoo.com
Tel :+251911684987

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

▣ Profile popups Spacing Relaxed Noise Very Low Layout Normal Per page 50 Update

First Prev Next

how to calculate	**Member 10568110**	**4-May-14 22:00**
Use of more fonts	**Member 10476040**	**1-Apr-14 0:47**
please reply	**Member 10454183**	**6-Feb-14 1:13**
reply soon	**Member 10454183**	**4-Feb-14 21:33**
Nice Article	**JoCodes**	**14-Jan-14 18:29**

Thank You	Karelsy	25-Dec-13 0:25
this is very good ocr article [modified]	iieyes	31-Jul-13 22:31
Error	programmer_ananth	25-Apr-13 0:52
Re: Error	ndhieuvn212	24-May-13 0:26
Re: Error	FIFXC	7-Mar-14 6:58
How to use training set of patterns	Parno Bego	17-Apr-13 22:46
Error	pradip2609	12-Jan-13 4:05
Re: Error	ARMDeveloper	8-Mar-13 11:50
Very nice work	Member 3839551	10-Jan-13 4:51
Nice	Abdunnasar	8-Jan-13 20:52
Reading Number Plates	Rohant Kunnat	26-Nov-12 21:38
Re: Reading Number Plates	Daniel Admassu	26-Nov-12 23:52
thank you-cai nay rat hay	cosnet	6-Oct-12 8:05
question	rajeshsoftorix	15-May-12 23:16
OCR Character Recognition	nazeim	18-Apr-12 2:55
Thinning process	hafizhans90	7-Nov-11 16:08
learning rate number	SilAli	19-Jun-11 16:56
My vote of 4	Member 7836829	17-Apr-11 22:45
RE OCR	newid4csharp	16-Apr-11 12:44

how to use more fonts	Simon Berson	26-Dec-10 0:14
Re: how to use more fonts	hoangminh2503	19-Mar-12 17:53
Character separation	Elmue	11-Nov-10 11:40
My vote of 5	**Mikaël Dallaire Côté**	8-Nov-10 5:57
charater_to_unicode	Truongphuc	12-Jun-10 8:53
plz 4r god sake let me knw very very urgent...how u got the tabular results by varying fonts, inputs, learning parameter	wringhdl	24-Feb-10 21:32
very very urgent plz 4r god sake...how u got the tabular results by varying fonts, inputs, learning parameter	wringhdl	24-Feb-10 21:31
problem to recognize	dtaprasetia	26-May-09 17:06
Re: problem to recognize	lexcanh	23-Sep-10 17:04
problem with multiple lines	Jia.C	16-Apr-09 14:14
Doubt	**GokulVS**	24-Feb-09 1:31
hi	**mohanshanthi**	30-Jan-09 1:15
hi	mohanshanthi	30-Jan-09 1:09
Best book to learn Neural Network ? [modified]	scanreg	28-Jan-09 10:27
Will this work with any characters ? - will it work with any scanners ?	scanreg	27-Jan-09 17:16
Russian Unicode [modified]	sambistt	12-Jan-09 14:18
i would like to make some optimization work on this, if it is ok.	DeeJayX	3-Dec-08 6:19
Re: i would like to make some optimization work on this, if it is ok.	Unruled Boy	2-Jan-09 6:12

Re: i would like to make some optimization work on this, if it is ok.	Daniel Admassu	5-Jan-09 6:08
whether will support for Kannada (Indic) ?	**74yrsold**	**27-Aug-08 21:45**
please help me!!!	**lulukuku**	**31-May-08 22:16**
Download	**abhinav singh**	**25-May-08 20:51**
Is it really perceptron?	**hnkaraca79**	**25-May-08 9:22**
Urgent Help	**Member 4775647**	**19-May-08 0:31**
Re: Urgent Help	Member 2336084	20-May-08 6:57
please help me	**vishwaa**	**18-Apr-08 22:40**

Last Visit: 31-Dec-99 19:00 Last Update: 8-Feb-15 21:11 Refresh 1 2 3 4 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.