Fuzzy-Zoning-Based Classification for Handwritten Characters

ABSTRACT:

In zoning-based classification, a membership function defines the way a feature influences the different zones of the zoning method. This paper presents a new class of membership functions, which are called fuzzy-membership functions (FMFs), for zoning-based classification. These FMFs can be easily adapted to the specific characteristics of a classification problem in order to maximize classification performance. In this research, a real-coded genetic algorithm is presented to find, in a single optimization procedure, the optimal FMF, together with the optimal zoning described by Voronoi tessellation. The experimental results, which are carried out in the field of handwritten digit and character recognition, indicate that optimal FMF performs better than other membership functions based on abstract level, ranked-level, and measurement-level weighting models, which can be found in the literature.

EXISTING SYSTEM:

- In the field of handwritten character recognition, zoning offers an easy way to obtain spatial information on feature distribution. Therefore, zoning has been largely considered to be a useful technique to handle the enormous variability of handwritten characters, due to different handwriting styles and personal variations.

- In the past, the static zoning methods used could be obtained by superimposing regular \(m \times n\) grids on a pattern image. In this case, the most-profitable zoning for a given classification problem was selected on the basis of a designer’s personal experience and on experimental tests.
PROPOSED SYSTEM:

- More recently, dynamic zoning methods have been introduced in which zoning design is considered as an optimization problem.

- This paper introduces a new class of fuzzy-membership functions (FMFs) and presents a real-coded genetic algorithm to detect, in a single optimization procedure, the optimal FMF, together with the optimal zoning, for the specific classification problem.

- The experimental tests were carried out in the field of handwritten numeral and character recognition, using datasets.

- The proposed system confirms the effectiveness of the genetic technique for the combined selection of the optimal zoning and optimal FMF.

A. 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

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.

**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:
  o Learning rate
  o Sigmoid slope
  o 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.

**Hardware Requirements**

- SYSTEM : Pentium IV 2.4 GHz
- HARD DISK : 40 GB
- FLOPPY DRIVE : 1.44 MB
- MONITOR : 15 VGA colour
- MOUSE : Logitech.
• RAM : 256 MB
• KEYBOARD : 110 keys enhanced.

Software Requirements

• Operating system :- Windows XP Professional
• Front End :- Microsoft Visual Studio
• Coding Language :- C#.NET

MODULES:
Training Phase
Testing Phase
Symbol image detection
Symbol image matrix mapping

MODULES DESCRIPTION:

Training Phase

Training Phase has various functionalities such as:

✓ 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

Testing Phase has various functionalities such as:

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

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 border 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

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:
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](image)

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

**REFERENCE:**