TALLINNA TEHNIKAÜLIKOOL
Arvutitehnika instituut

Töö kood: IAY40LT

Klassifitseeriv närvivõrk programmeerimiskeeles Java

Bakalaureusetöö

Üliõpilane: Filipp Keks
Tudengi kood: 030725 IASB
Juhendaja: Peeter Ellervee

Tallinn
2006

TALLINN UNIVERSITY OF TECHNOLOGY
Department of computer engineering

Thesis code: IAY40LT

Classifying neural network in Java programming language

Bachelor
thesis

Student: Filipp Keks
Student code: 030725 IASB
Tutor: Peeter Ellervee

Tallinn
2006

Author's Declaration

I hereby declare that I am the sole author of this thesis. All ideas, important opinions or data coming from any other sources and other authors are referenced.

Olen koostanud antud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

Filipp Keks

Abstract

Artificial neural networking as an effective image classification algorithm is described in this thesis. Algorithm is presented in real example of optical character recognition (OCR) program which is implemented in Java programming language. Besides classification algorithm this thesis includes descriptions of all problems that occur in the process of development of real working OCR system. Each problem description includes one or several solutions as well as their substantiations and working implementations. Among others the reviewed topics include: image preprocessing, invariant feature extraction, character identification, network training and character database generation.

The work that was done during writing of this thesis, that is OCR program itself, is released as an open-source project and its source code is freely available for everybody at the following URL: http://ocr4j.sourceforge.net

Kokkuvõte

Tehisnärvivõrk on üks moodustest modelleerimaks bioloogilist aju. Sellise võrgu teooria baseerub eeldusel, et arvutis on võimalik esitada bioloogilise neuroni peamisi omadusi. Eesmärgiks on saavutada inimajuga võrreldavad arvutuslikud võimalused. Vaatamata sellele, et tehisnärvivõrk on väga lihtsustatud koopia reaalsest bioloogilisest prototüübist, võib seda efektiivselt kasutada aladel, kus inimestel on eelis masinate ees, näiteks kujundituvastuses.

Klassifitseeriv närvivõrk on võrgu tüüp mille ülesandeks on mustrite tuvastamine. Sellised võrgud on väga levinud nii sõjanduses kui ka tsiviilelus. Üks populaarne klassifitseeriva võrgu kasutusala on optiline sümbolite tuvastus (OST). OST programmid aitavad teisendada trükitud või käsitsi kirjutatud teksti masinloetavaks koodiks, mis lubab teksti sisu muutmist ja edasist kasutamist.

Käesolevas töös on kirjeldatud tehisnärvivõrku efektiivse klassifitseeriva algoritmina. Algoritmide abil on realiseeritud OST programmi, kirjutatud programmeerimiskeeles Java. Lisaks klassifitseerimisalgoritmile on selles töös kirjeldatud ka probleeme, mis tekkivad OST süsteemi arendamisel. Iga probleemi kirjeldus sisaldab ühte või mitut lahendust koos põhjenduste ja realisatsiooniga. Täiendavate teemadena on kirjeldatud pildi eeltöötlust, sõltumatute erisuste eraldamist, tähtede identifitseerimist, närvivõrgu õpetamist ja sümbolite andmebaasi loomine.

Kõik kirjeldatud algoritmid on täielikult realiseeritud ja neid on kontrollitud praktilises töös. Samuti on analüüsitud algoritmide efektiivsust. Töö sisaldab mõningaid Java lähtekoodi lõike koos seletustega.

Kirjeldatud algoritmide realisatsioonid on avalikustatud SourceForge projektina ja on leitavad veebilehelt http://ocr4j.sourceforge.net.

Table of Contents

Illustration Index

Terminology

OCR	Optical Character Recognition
ANN	Artificial Neural Network
Character	A symbol, this term includes only the meaning of the symbol, not the representation
Glyph	A visual representation of the character (Optimized to be simple for human)
ASCII	American Standard Code for Information Interchange. One of the computer-optimized character representations.
NIST	National Institute of Standards and Technology

1.Introduction

Artificial neural network (ANN) is an attempt to simulate the biological brain. Neural network theory revolves around the idea that certain key properties of biological neurons can be extracted and applied to simulations. The idea is to reach the computing potential of the human brain. Artificial neural networks are even used as a way to study the human mind [1]. Though ANN is a very much simplified model of its biological prototype, it can be successfully used in the domain of tasks, where human has an advantage over machines, like for example image recognition.

Classifying neural networks is the type of networks which aim is to identify patterns. The domain on usage of these networks is very wide, for example in military technology these networks are used for automatic identification of weapons and equipment like tanks and airplanes, in civil domain they are helping police to recognize human faces, and in everyday life classifying neural networks are used for optical character recognition (OCR).

Optical character recognition programs help to read the printed or handwritten text from images of scanned documents and books. Programs convert text from pictures into a machine code understandable by the computer, so that content can be modified and reused. This thesis describes the process of development of real working OCR system based on the classifying neural network. Different approaches, algorithms and problem solutions are studied. All theoretically substantiated aspects are implemented and proven on practice. Together with OCR program itself, neural network training tools are created as well as sample character database and demo applications. The created program is very portable, it can be used on any platform which have a Java virtual machine implementation, these platforms include Linux, BSD, Solaris, MAC, Windows and many others.

Given thesis is structured such that parts of the recognition process are described in order of their application. One or several solutions are provided for each recognition step, Java implementations of some solutions are provided with descriptions. System

implementation structure and achieved results description can be found in the end.

2.OCR system

Optical character recognition system usually consists of separate modules which carry out the different parts of recognition process.

- **Preprocessing**. A paper document is scanned to produce a gray scale or colored image. In the first stage the raw scanned picture is filtered to remove colors and convert it to binary image. Preprocessing module is responsible for removing **noise** and locating **text areas**.

- **Clusterization** module finds the **individual characters** in the image and computes the correct **character order** in the text.

- **Feature extraction**. The goal of this module is to extract the **unique features** of the individual character so it can be recognized by the classification module. If we compare OCR system with a human, we can say that this module plays the role of the **human eye**.

- **Classification**. In the last stage OCR system tries to guess the character using the information about character features extracted in the previous stage. Although it can be done statistically which involves consistent comparison of characters from database, it was chosen to try how neural network approach performs for this task.

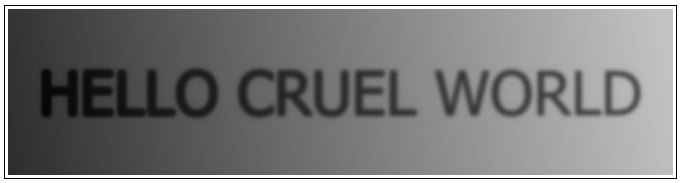
3.Image preprocessing

The complexity of image processing algorithms are beyond the topic of this thesis, it will neither be reviewed in much detail nor researched throughly. The main aim of the preprocessing is to divide color spectrum of the image into background and foreground.

Adaptive threshold

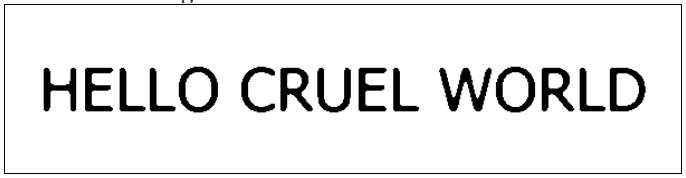
One of the approaches that can be used to detach background and foreground colors is called adaptive (or dynamic) threshold. Thresholding used to segment image by setting the pixels that have intensity value above the threshold to foreground value and those whose value is under threshold to background.

There are two different types of thresholding algorithms. The global thresholding approach uses a single threshold value computed for the whole image, this may work in most cases when the image is equally light in all its regions, but in our case images are not always so ideal. Another approach is local thresholding: the image is divided into smaller overlapping parts and threshold value is computed for each part independently adapting the local histogram. This technique allows us to segment characters in the image regardless of lightening.



Original image

Global thresholding



Local thresholding

Figure 1: Adaptive thresholding algorithm examples

As we can see in the Figure 1, image with gradient background is segmented incorrectly by the global thresholding algorithm, though local segmentation works well for the same image.

4.Clusterization

After defining the foreground and background areas of the image we need to identify separate glyphs. This part is implemented by the Java class ***ee.ttu.ocr.ImageClusterer***.

I made some rough assumptions to simplify this part of the recognition process.

Rough assumptions

1. Each character is fully in one solid peace without any additional excrescences. Which is not fair for "i" and "j" Latin lowercase characters. This assumption allows us to identify capital letters only.

2. Characters are not intersected, there is always a clear space between separate characters. This is not true for low quality scans. There are some algorithms allowing to guess the unclear character sequence using the knowledge about common language specific features, but the complexity of this work would grow too dramatically if I try to implement any of these.

3. No random noise exists in the picture. Any peace of foreground color in the picture assumed to be a character glyph.

Finding character boundaries

To find the character shape I used a simple flood fill algorithm, that is commonly used for "bucket" fill tool in paint programs to determine which parts of bitmap to fill with color. The idea is to find at lest one pixel of the glyph and then fill it with some predefined color. We are able to find the character boundaries in filling process and cut the glyph from the image so that it can be later processed by the feature extractor as a single character bitmap.

Flood-fill algorithm requires 3 input parameters: target color, replacement color and start point. As preprocessing stage guarantees that picture contains only 2 colors: background

and foreground, color choice is quite simple for us. Foreground color of the picture is a target color for flood-fill algorithm. Replacement color is not important, but it should be different from foreground and background.

Algorithm is quite simple. The first step is to find the leftmost and the rightmost pixel filled with target color in the start point horizontal line. Then we paint the line with the replacement color simultaneously adding the pixels above and under the line to the queue. We repeat the same procedure for each point in the queue and continue doing this until the queue is empty.

```
// From ImageClusterer.java
private ClusterBoundaries floodFill(Point startPoint) {
  ClusterBoundaries boundaries = new ClusterBoundaries();
  queue.offer(startPoint);
  Point point;
  do {
    point = queue.poll();
    if (point==null || !isPointSet(point.getX(), point.getY())) {
      continue;
    }
    int y = point.getY();
    boundaries.setTop(y);boundaries.setBottom(y);
    int w,e;
    // Find western and eastern points of the line
    for (w = point.getX(); isPointSet(w-1,y); w--);
    for (e = point.getX(); isPointSet(e+1,y); e++);
    boundaries.setLeft(w);boundaries.setRight(e);
    // Fill the line and add above and under points to the queue
    for (int x = w; x <= e; x++) {
      unsetPoint(x,y);
      if (isPointSet(x,y-1))
        queue.offer(new Point(x,y-1));
      if (isPointSet(x,y+1))
        queue.offer(new Point(x,y+1));
    }
  } while(point!=null);
  return boundaries;
}
```

Figure 2: Flood-fill algorithm listing

As you may see in Listing 2 method *floodFill* requires any point of the glyph as an input parameter and returns the its boundaries which can be used to cut the glyph from the image.

Finding correct character order

To use the flood-fill algorithm first we need to find at lest one point for each character in the image and compute the order of the characters in the text.

Figure 3: Clusterization example

We can find characters by simply scanning the image. The only problem is to determine the right scanning direction that would guarantee the correct order of the characters. For example if we scan image, presented in Figure 3 vertically from top to the bottom, the first character that we will find will be "L", and instead of the word "HELLO" we will get something like "LHOLE". If we try to scan image horizontally from left to right, the first letter will be "C".

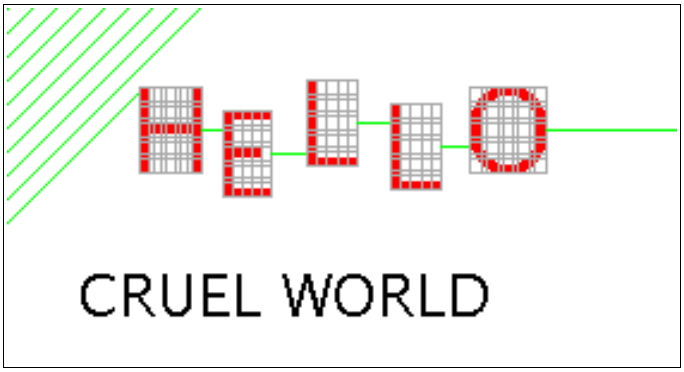


Figure 4: Clusterization example

I found a solution by scanning image in the diagonal direction. In Figure 4 you may see an example of the scan process. The diagonal scan started from the upper-left corner and when the first character is found, scan is continued horizontally. When horizontal scan reached the right border of the image this assumed to be a line break. The diagonal scan is repeated to find the next line.

Detecting spaces

Another problem is spaces between characters. I used some kind of self adapting word break detection algorithm. The average width of single character is computed constantly, if the space between characters is bigger than half of average character width then it

assumed to be a word break. But sometimes characters of different sizes are used in one text, much bigger font is used for headings or emphasized text, so I made the average character width to be reseted if size of a character is changed rapidly. See the Figure 5 for the implementation.

```
// From ImageClusterer.java
private boolean isSpace(ClusterBoundaries boundaries1,
                        ClusterBoundaries boundaries2) {
  if (boundaries1.getWidth() > averageClusterWidth*2 ||
      boundaries1.getWidth()*2 < averageClusterWidth) {
    averageClusterWidth = boundaries1.getWidth();
    numberOfFoundClusters=1;
  }
     else {
    averageClusterWidth = (float)(averageClusterWidth * numberOfFoundClusters +
                                  boundaries1.getWidth()) / (numberOfFoundClusters+1);
    numberOfFoundClusters++;
  }
  return boundaries2.getLeft()-boundaries1.getRight() >
         averageClusterWidth/2;
}
```

Figure 5: Space detection algorithm listing

5.Feature extraction

Feature extraction module converts the image into a sequence of numbers which is then processed by the classification module. The main requirement for the feature extraction algorithm is invariantion to image distortion and different types of writing. There are two different approaches for combining a classifying artificial neural network with feature extraction.

Implicit feature extraction

One approach is called implicit feature extraction. The simple feature extraction algorithm is combined with complex ANN. Features are extracted implicitly inside the neural network.

Advantages

1. Easy to implement. No need to invent and support a complex algorithms for feature extraction, the work is done by the neural network.

2. Flexibility. If we find that network is unable to read some type of text in the image, we can just add it to the training data and restart the learning process, instead of reimplementing the complex algorithm.

Disadvantages

1. Requires a lot of training material for ANN. To make the network understand the glyph distortion invariants we need a complex training data which represents all possible distortions and different writing properties.

2. Requires a complicated ANN structure with great number of neurons and layers.

3. The quality of the work of OCR system can not be proven mathematically.

We can judge the quality only by the system work results.

Explicit feature extraction

Second approach is called explicit feature extraction. In this approach the features of character glyph are extracted by the concrete explicitly defined algorithm.

Advantages

1. This approach is not tied to classifying ANN. Any kind of classification algorithm may be used with explicit feature extractor. For example it may be a statistically comparing nearest-neighbor classifier.

2. Requires simpler ANN as a classifier, which is easier to teach with smaller training data.

3. It is possible to find the exact reason of the failure by debugging the algorithm.

Disadvantages

1. It is quite hard to found and implement the algorithm which can equally represent the same character glyph independently from font or handwriting specific properties.

Because of the difficulty of achievement of the representation invariance property of feature extraction method by any kind of analytic and theoretical means it is often pursued heuristics and more or less intuitive procedures.

Algorithms

I choose to use some kind of mix of both feature extraction approaches, mentioned above. My idea is to explicitly use the feature extraction algorithm, which is able to partly reduce the differences between fonts and handwritings. This allows me to fill up the lack of

training data.

Image region density

The first idea that comes to mind is to use pixel values as a receptors of the artificial eye. To make the number of receptors constant we can break the image into a number of rectangle regions and calculate the average color density of pixels in each region. The implementation of this algorithm can be found in class ***ee.ttu.ocr.SquareRegionDensityEye***.

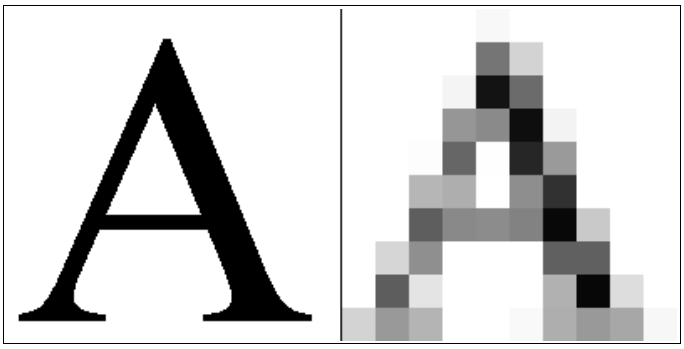


Figure 6: Region density

This algorithm is easy to implement, but it is very ineffective when used as explicit feature extraction method. Density of regions is not resistant to glyph distortion. Another disadvantage of this method is that number of receptors is too large. Most receptors represent the background and does not contain any information about the character glyph. Though this algorithm is often used with implicit feature extraction approach.

Random line receptors

Algorithm is based on the conception presented in the project of Andrew Kirillov [2]. This is a some kind of attempt to reduce the disadvantages of the previously described region density extractor. The main idea is the same: calculate average density of some regions, but in this case region shapes and positions should be optimal for the specific set of characters and their image representations. The optimal set of receptors is defined only once, so it can be done manually, but in this work the automatic algorithm is used. Its implementation can be found in Java class ***ee.ttu.ocr.RandomReceptorEye***.

To simplify the implementation line shaped receptors were used. The most complex

calculations are done before start of training process. Precalculation steps are:

1. Generate a number of random lines and make sure that they are not coincide with each other.

2. Calculate the quality of each line receptor. See page 24 for quality calculation algorithm description.

3. Throw away receptors with poorest quality.

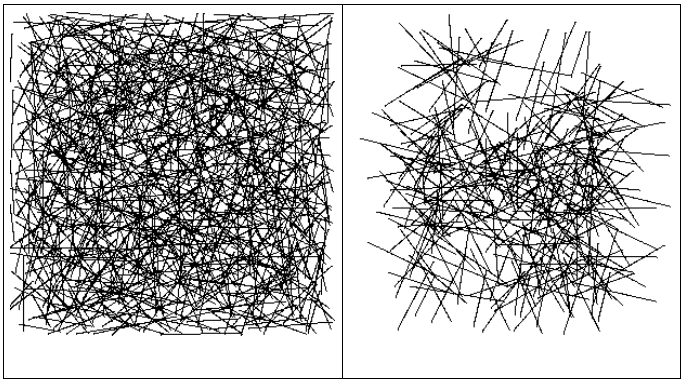


Figure 7: Receptors optimization

In Figure 7 you may see an example of receptors optimization. The initial state of feature extractor is on the left picture – 1000 receptors are generated randomly. Usability of each receptor is calculated for alphabet that contains letters 'A', 'S', 'D', 'F', 'G' and 'H', with 4 different glyph variants for each letter. On the right picture only 200 receptors with best usability values are left, as you may see receptors some kind of follow the shapes of glyphs.

Image region contour direction

Algorithm is based on the work of Carlos Perez and Enrique Vidal [3]. This is also a kind of image region density (See page 21) method extension. Same as in named method we divide the image into number of static regions, rectangles for example. But instead of calculating average density of the region color we calculate the average direction of lines inside the region. This algorithm is implemented by ***ee.ttu.ocr.LocalContourDirectionEye*** class.

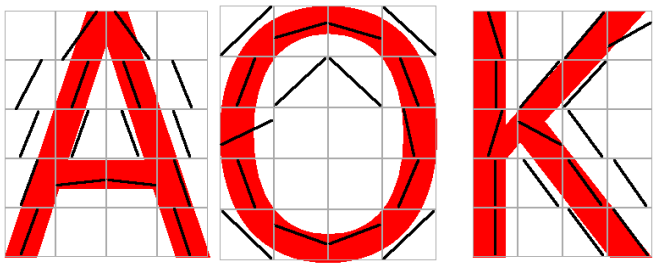


Figure 8: Local contour direction

As we can see from Figure 8, some regions may be totally blank, this means that they don't have a contour direction, and others can have only one tiny pixel inside them. To avoid the rapid change of receptions indications with small changes of glyph shape we can combine contour direction with region density.

Another problem of this method is angle representation. Representation of value is usable when it is unique and continuous. This means that sensor value should change smoothly with small changes of direction angle and each sensor value should represent only one unique angle value. If we use a sine or cosine representations of the angle we loose uniqueness, absolutely different angles can be represented by one cosine value $\cos(\pi/2)=\cos(-\pi/2)$. Angle representation $\arctan(k)$ lacks continuousness, nearly vertical lines can have a very different representations, like 0.1 and $\pi-0.1$ radians.

This pair of expressions meets the above mentioned requirements.

$$f_1=\sin(2\cdot\arctan(k))=\frac{2k}{1+k^2} \qquad f_2=\cos(2\cdot\arctan(k))=\frac{1-k^2}{1+k^2}$$

f_1 first direction feature

f_2 second direction feature

k straight line angle coefficient $y=k\cdot x+a$

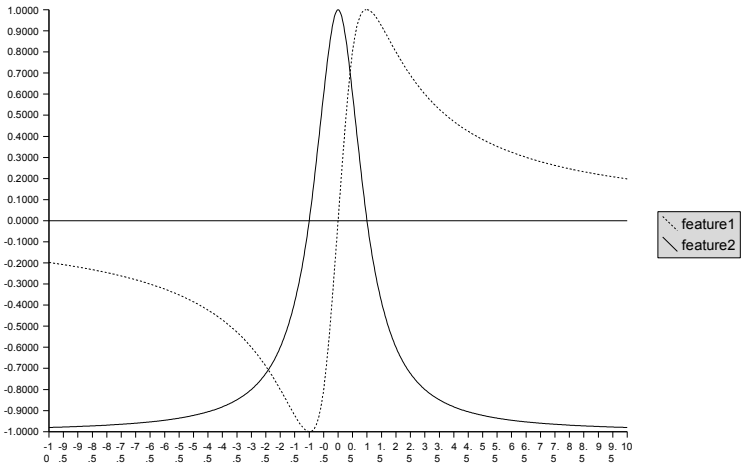


Figure 9: Contour direction feature plot

Both features are continuous, their graphs tend to the same value at the both sides of infinity.

First feature: $\lim_{k\to\infty} f_1 = \lim_{k\to\infty} \frac{2k}{1+k^2} = 0 \qquad \lim_{k\to-\infty} f_1 = \lim_{k\to-\infty} \frac{2k}{1+k^2} = 0$

Second feature: $\lim_{k\to\infty} f_2 = \lim_{k\to\infty} \frac{1-k^2}{1+k^2} = -1 \qquad \lim_{k\to-\infty} f_2 = \lim_{k\to-\infty} \frac{1-k^2}{1+k^2} = -1$

The combination of these two parameters is unique for every k .

In combination with density we have 3 receptors per region. Unlike region density feature extractor, which receptors number equals number of image regions, this eye have 3 times more receptors.

Quality measurement of feature extraction algorithms

This quality measurement algorithm is used for optimization in random line feature extractor (page 21) and for impartial efficiency comparison of different feature extractors. It is implemented by ***ee.ttu.ocr.teaching.EyeQualityInspector*** Java class.

Usability of extracted features can be measured by two main properties: feature should be common for all glyph variants of one character and vise-versa it's value should be

different for various characters. To measure these properties we can use entropy.

$$\eta = e_o \cdot (1 - e_i)$$

- η relative quality coefficient
- e_o outer entropy is calculated on values which receptor receives on different characters of alphabet.
- e_i inner entropy is calculated on values which receptor receives on different glyphs of the same character.

Of course we have a number of inner and outer entropies for a single receptor, so we use an average value.

Entropies are calculated using the correlative entropy calculation method.

$$e = -\ln \frac{\sum_{i,j}^{n} 1(|a_i - a_j| - \xi)}{n(n-1)}$$

- e correlative entropy
- $a_1, a_2 \ldots a_n$ values of the studied parameter
- $1()$ Heaviside function
- n number of studied values
- ξ coefficient that depends on number of values their sparseness, it should be somewhere near the average distance between values

Correlative entropy is an entropy estimation, it aspires to the real entropy with infinite number of parameter values. $E = \lim_{\substack{n \to \infty \\ \xi \to 0}} e$

See Figure 10 for Java implementation of algorithm described above.

```
// From EyeQualityInspector.java
/**
  * Measure a relative usability of the eye for recognizing the given course
  * Usability measurement algorithm is based on entropy of eye receptor values.
  *
  * @param eye
  * @param course
  * @return array of receptor usabilities
  * @throws OCRTeachingException
```

```
  */
public static float[] getEyeReceptorUsabilities(Eye eye, OCRTeachingCourse course)
                                                                    throws OCRTeachingException {
  Map<Character, List<float[]>> eyeReceptorValues = getEyeReceptorValues(eye, course);
  float[] result = new float[eye.getReceptorsCount()];
  for(int receptorIndex = 0; receptorIndex < eye.getReceptorsCount(); receptorIndex++) {
    float avgInnerEntropy = getReceptorInnerEntropy(eyeReceptorValues, receptorIndex);
    float avgOuterEntropy = getReceptorOuterEntropy(eyeReceptorValues, receptorIndex);
    result[receptorIndex] = avgOuterEntropy * (1-avgInnerEntropy);
  }
  return result;
}

private static float getReceptorOuterEntropy(Map<Character, List<float[]>> eyeReceptorValues,
                                                                    int receptorIndex) {
  float sumOuterEntropy = 0;
  for (int i=0; i< characterImagesCount; i++) {
    float[] values = new float[eyeReceptorValues.size()];
    int j=0;
    for (Iterator<Character> it = eyeReceptorValues.keySet().iterator(); it.hasNext(); j++) {
      values[j] = eyeReceptorValues.get(it.next()).get(i)[receptorIndex];
    }
    sumOuterEntropy += MathEx.correlativeEntropy(0.2f, values);
  }
  return sumOuterEntropy/characterImagesCount;
}

private static float getReceptorInnerEntropy(Map<Character, List<float[]>> eyeReceptorValues,
                                                                    int receptorIndex) {
  characterImagesCount = 999;
  float sumInnerEntropy = 0;
  for (Character character : eyeReceptorValues.keySet()) {
    List<float[]> receptorValues = eyeReceptorValues.get(character);
    if (characterImagesCount > receptorValues.size()) {
      characterImagesCount = receptorValues.size();
    }
    float[] values = new float[receptorValues.size()];
    int i = 0;
    for (ListIterator<float[]> iterator = receptorValues.listIterator(); iterator.hasNext();
                                                                                  i++) {
      values[i] = iterator.next()[receptorIndex];
    }
    sumInnerEntropy += MathEx.correlativeEntropy(0.2f, values);
  }
  return sumInnerEntropy/eyeReceptorValues.size();
}

// From MathEx.java
/**
 * Calculate the entropy of the array values using correlative integral
 * @param distance
 * @param values
 * @return correlative entropy
 */
public static float correlativeEntropy(float distance, float[] values) {
  int result = 0;
  for (int i=0; i<values.length; i++) {
    for (int j=i+1; j<values.length; j++) {
      if (java.lang.Math.abs(values[i]-values[j]) < distance) {
        result++;
      }
    }
  }
  return (float)-Math.log(2*(float)result/(values.length*(values.length-1)));
}
```

Figure 10: Feature extraction quality measurement algorithm

Quality of implemented algorithms

Usability of 3 different feature extraction algorithms were measured for character

database containing 520 images (See page 38). Algorithms were inspected with different initialization parameters. The best results of each algorithm are in the table below.

Algorithm	**Optimal parameters**	**Quality coefficient**
Random receptors	Initial receptors count: 500 Minimal quality coefficient: 0.4 Number of receptors: 132	0.455
Region density	Grid width: 2 Grid height: 9 Region overlapping: -10% Number of receptors: 18	0.369
Region contour direction	Grid width: 3 Grid height: 9 Region overlapping: -15% Number of receptors: 81	0.251

It came out that Random receptor feature extractor is best according to the used quality measurement algorithm. It is not surprising because it was optimized by the same algorithm. Theoretically the quality of this extractor can by set to any number by adjusting the initial parameters, but it may be dangerous because in case of large minimal quality coefficient parameter algorithm may choose to recognize only parts of the image which are easiest to recognize. In other words it will be an image data loss.

The surprising thing is that a dummy region density algorithm shows better results than its extended version. Another surprising thing is that positive overlapping parameters made quality coefficient to fall down, but negative values of this parameter vise-versa make the result better.

Practical tests show about the same results (See page 47), so the quality measurement results can be considered to be adequate.

Limitations

Described quality measurement algorithm may show an inadequate results. The algorithm takes into account only individual receptors, not being aware of what number of receptors is required for good recognition and how the image area is covered by receptors.

6.Classification

In classification step the sequence of numbers, returned by feature extractor is converted to computer interpretation of concrete character, for example ASCII code.

Statistical pattern classification

One of the approaches of pattern classification is to use statistical comparison between input data and data from database. Nowadays this approach is used more rarely than artificial neural networks, however, there is no conclusive evidence about superiority of one over another [4].

Statistical approach is not a part of the given thesis.

Artificial neural network pattern classification

Artificial neural network is an attempt to simulate a human brain. The idea is to create a number of simple and independent data processing items, called neurons, and connect them to each other so that they become a complex network. Same as a human brain ANN is able to learn. In learning process each item adjusts its internal state independently according to some defined law. The first connectionism hypothesis that later formed the ANN technology was proposed by Friedrich Hayek in 1952 [5].

There are a lot of different ANN types. At first neural networks are classified by the type of neuron connections. The network type, which is used in this thesis, is called feed-forward, in this type of networks the output of any neuron never returned to this neuron directly or through any number of other neurons. Another type is called recurrent or feedback neuron network, in this type of network neurons can be connected anyhow.

Feed-forward neural network

Feed-forward neural network usually consists of several layers: one input layer, one output and a number of internal layers. Often input layer contains the biggest number of

neurons and output layer the least. Input layer collects the input data and passes it via nodes to the first internal layer, first input layer passes it to the second and so on until data reaches the output.

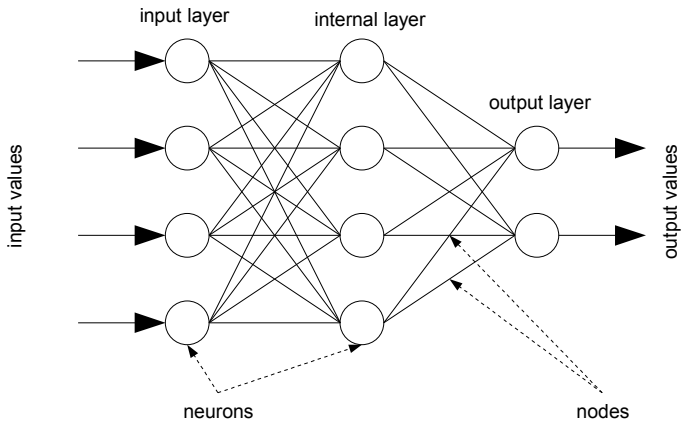


Figure 11: Feed-forward neural network

Neuron

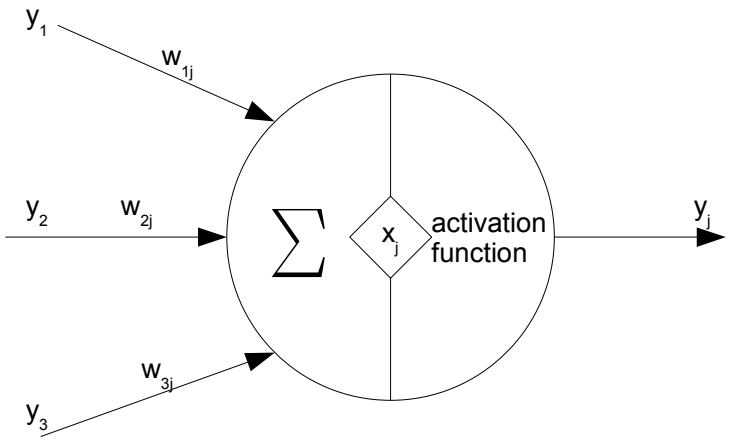


Figure 12: Artificial neuron

x_j Neuron internal value

y_j Neuron output

w_{ij} Node weight

A simple data processing item – neuron consists of two parts. First part collects data from the input neurons and calculates the neuron internal value.

$$x_j = \sum_i y_i \cdot w_{ij}$$

Second part is an activation function, which processes the internal value of the neuron and forms output.

$$y_j = f(x_j)$$

See page 34 for detailed description of different activation functions.

While passing through node from one neuron to another, signal is distorted by it's weight. The input values, which are received by neurons of the next layer from one neuron are the output value of this neuron multiplied by node weight.

```
public class Neuron implements Serializable {
     protected Float cachedOutput;

     protected ArrayList<Node> inputNodes;
     protected ActivationFunction activationFunction;

     public Neuron(ActivationFunction activationFunction) {
            this.activationFunction = activationFunction;
     }

     /**
      * Get output of the neuron.
      * If the internal value is not yet set, this method activates the whole network
      *                                                                          recursively
      * to get the internal value.
      * @return neuron value
      */
     public float getOutput() {
            if (cachedOutput == null) {
                  float value = 0;
            for (Node inputNode : inputNodes) {
                value += inputNode.getOutput();
            }
                  cachedOutput = activationFunction.calculate(value);
            }
            return cachedOutput;
     }

     /**
      * Sets the internal value of the neuron. This should be used only for input neurons.
      * @param value
      */
     public void set(Float value) {
            cachedOutput = activationFunction.calculate(value);
     }

     public void addInput(Neuron neuron) {
            if (inputNodes == null)
                  inputNodes = new ArrayList<Node>();
            inputNodes.add(new Node(neuron));
     }
```

```
	public void clear() {
		cachedOutput = null;
	}

	public void randomizeInputNodes() {
	   for (Node node : inputNodes) {
		node.randomize();
	   }
	}

	public class Node implements Serializable {
		private Neuron inputNeuron;
		protected float weight;

		public Node(Neuron inputNeuron) {
			this.inputNeuron = inputNeuron;
			randomize();
		}

		public float getWeight() {
			return weight;
		}

		public Neuron getInputNeuron() {
			return inputNeuron;
		}

		public float getOutput() {
			return inputNeuron.getOutput()*weight;
		}

		public void randomize() {
			weight = Random.next();
		}
	}
}
```

Figure 13: Java implementation of neuron

Learning

There are two main approaches of ANN training: Supervised and Unsupervised training.

In Supervised training network is provided with both input and output data. Network then processes inputs and compares results with outputs trying to be very close to desired output data.

In Unsupervised learning network is only provided with input data. Network tries to adapt the changing input without knowing the required result.

In this work Supervised training approach is used. Artificial neural network learning process is practically a process of node weight adjustment. Weights are adjusted according to learning patterns. Learning patterns is a pair of data sequences – input sequence and a desired output data sequence. Network output error is a difference between real output

and desired output pattern. The goal of learning process is minimizing the output error.

The overall error of the network output can be calculated like this

$$E = \frac{1}{2} \sum_i (y_i - d_i)^2$$

E	overall network output error
$y_1, y_2 \dots y_n$	real network outputs
$d_1, d_2 \dots d_n$	desired network outputs

Genetic learning algorithm

One of the network learning approaches is adjusting node weights genetically. In each learning iteration (generation) nodes are adjusted randomly. If output error is bigger then in previous generation, new network is destroyed, otherwise it is used as a base on the next generation. Process is repeated until output error reaches some acceptable level. Despite the simplicity of this algorithm, it is very ineffective.

Back-propagation learning algorithm

Another approach, which is used in this work is called Back-propagation. It is first described by Paul Werbos in 1974 [6]. The idea of this approach is to calculate the error of output value of each neuron starting from output network layer and back propagating it to the previous layer. So that each neuron may adjust its nodes based on the known self error value. See ***ee.ttu.ann.BackPropagationLearningNetwork*** class for implementation.

The back-propagation algorithm consists of four steps

1. Compute the output errors of layer neurons

$$eo_i = \frac{\delta E}{\delta y_i} = y_i - d_i$$

2. Calculate the input errors

$$ei_i = \frac{\delta E}{\delta x_i} = \frac{\delta E}{\delta y_i} \cdot \frac{\delta x_i}{\delta x_i} = eo_i \cdot f^{-1}(y_i)$$

where $f^{-1}()$ is a derivative of the activation function

3. Calculate the node weights errors

$$ew_{ij} = \frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta x_i} \cdot \frac{\delta x_i}{\delta w_{ij}} = ei_i \cdot y_j$$

4. Compute the output errors of the previous layer

$$eo_j = \frac{\delta E}{\delta y_j} = \sum_i \frac{\delta E}{\delta x_i} \cdot \frac{\delta x_i}{\delta y_j} = \sum_i ei_i \cdot w_{ij}$$

Steps are repeated starting from step 2. for all layers until the input layer is reached. Weight error values ew_{ij} are used to adjust the weights of the nodes.

```
public class BackPropagationLearningNeuron extends Neuron {

     float errorOutput;

     public BackPropagationLearningNeuron(ActivationFunction activationFunction) {
            super(activationFunction);
     }

     /**
      * Backpropagate the error to the input neurons<br/>
      * and adjust the input node weights.<br/>
      * <p/>
      * This method should be called when error neuron value is completely set.
      */
     public void learn() {
        if (inputNodes == null)
            return;
        for (Neuron.Node node : inputNodes) {
            BackPropagationLearningNeuron nodeInputNeuron = node.getInputNeuron();
            float errorInput = errorOutput*activationFunction.calculateDerivative(cachedOutput);
            nodeInputNeuron.increaseError(errorInput*node.getWeight());
            float errorWeight = errorInput*nodeInputNeuron.getOutput();
            ((Node)node).adjustWeight(errorWeight);
        }
        errorOutput = 0;
     }

     public void increaseError(float incError) {
            errorOutput += incError;
     }

     public List<Neuron.Node> getInputNodes() {
            return inputNodes;
```

```
    }

    public class Node extends Neuron.Node {
        public Node(Neuron inputNeuron) {
            super(inputNeuron);
        }

        public void adjustWeight(float errorWeight) {
            weight -= errorWeight;
        }
    }

}
```

Figure 14: Java implementation of back-propagation learning neuron

Activation functions

Activation function is used to compute the neuron output value. Properties of this function directly influence the properties and behavior of the network: its learning ability, complexity etc.

Theoretically any function can be used as an activation function, but there are ones that are used more often mostly because of calculation convenience.

Sigmoid function

This function is very often used with back propagation learning algorithm.

$$f(x)=\frac{1}{1+e^{-\alpha x}}$$

Derivative	$f^{-1}(y)=\alpha y(1-y)$
Value range	(0 : 1)

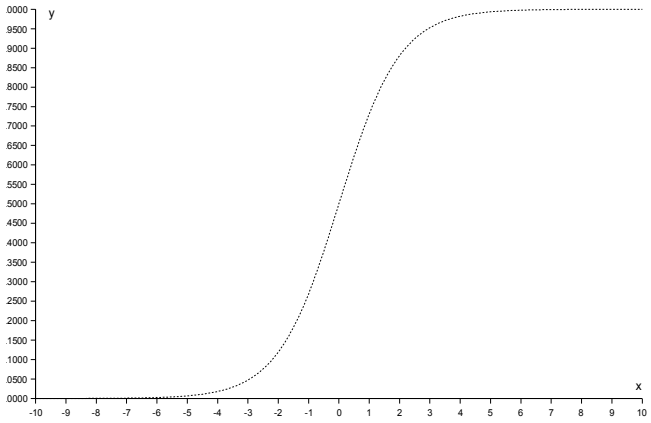


Figure 15: Graphical presentation of the sigmoid function

In Figure 16 you may see how network error decreases in learning process. Sigmoid

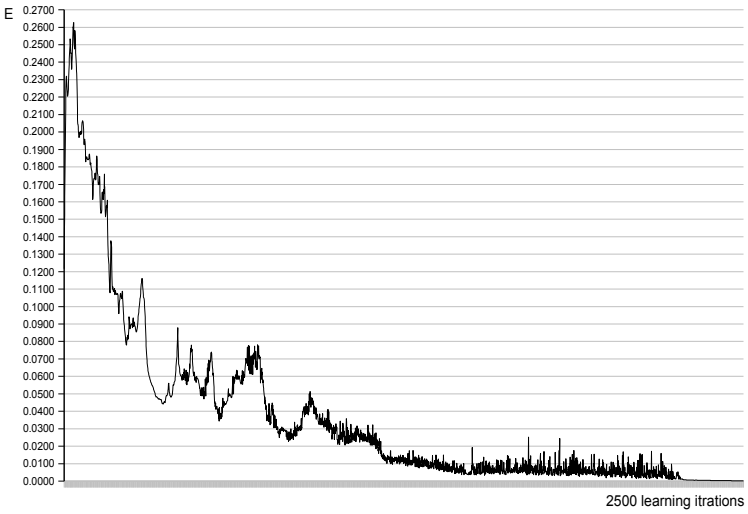


Figure 16: Learning process of the network based on the sigmoid function

function based network learning process is not very smooth, the error value jumps up and down before reaching the required level very close to zero.

This function Java implementation is in **ee.ttu.math.SigmoidFunction** class.

Bipolar sigmoid function

This is a sigmoid function that is symmetric relatively to X axis. Implementation can be found in ***ee.ttu.math.BipolarSigmoidFunction***.

$$f(x)=\frac{1}{1+e^{-\alpha x}}-\frac{1}{2}$$

Derivative $$f^{-1}(y)=\alpha\left(\frac{1}{4}-y^2\right)$$

Value range (-0.5 : 0.5)

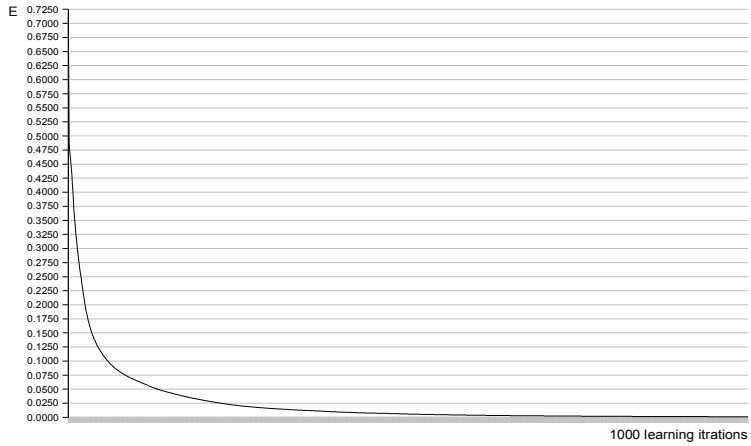


Figure 17: Learning process of the network based on the bipolar sigmoid function

Unlike with sigmoid function, bipolar sigmoid function based network decreases its error almost perfectly smoothly (Figure 17) and with equal conditions it reaches the appropriate error level about two times faster.

Hyperbolic tangent function

Implemented by ***ee.ttu.math.HyperbolicTangentFunction*** Java class.

$$f(x)=\tanh(\alpha x)=\frac{e^{\alpha x}-e^{-\alpha x}}{e^{\alpha x}+e^{-\alpha x}}$$

Derivative $$f^{-1}(y)=\alpha(1-y^2)$$

Value range (-1 : 1)

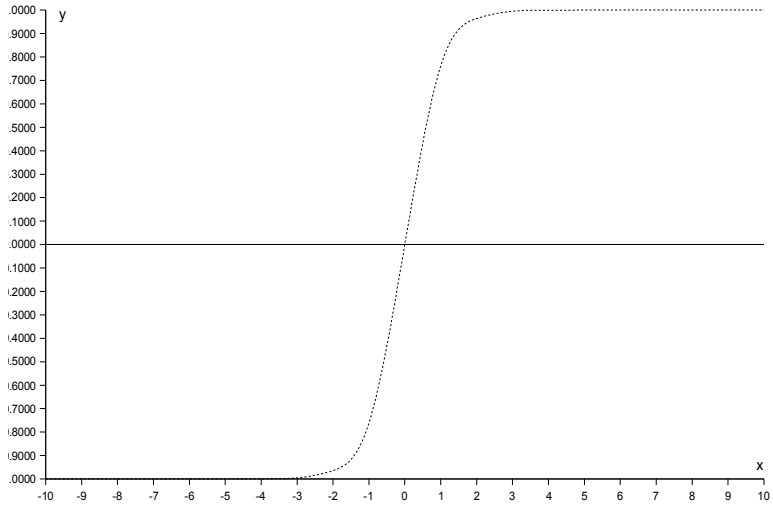


Figure 18: Graphical presentation of the hyperbolic tangent function

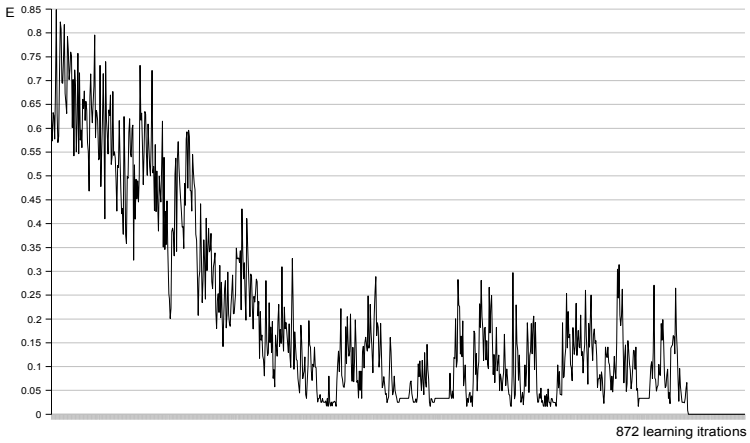


Figure 19: Learning process of the network based on the hyperbolic tangent function

Network based on hyperbolic tangent function makes network to learn very fast, but at the same time it makes the error value to change very rapidly in the learning process, what makes the network somewhat unreliable: sometimes error value does not reach the required level at all.

7.Character database

Character database is the data which is used to teach network to understand the characters. Some institutions manage the databases of images of handwritten characters and sell the to the companies which create a commercial OCR products. Like for example NIST [7] character database contains over 800000 images and can be purchased from their website for $90.

I've created my own character database. It contains only capital letters with 20 different glyphs of each character, 8 of which are my own handwritings and 12 are generated by computer. 520 images overall.

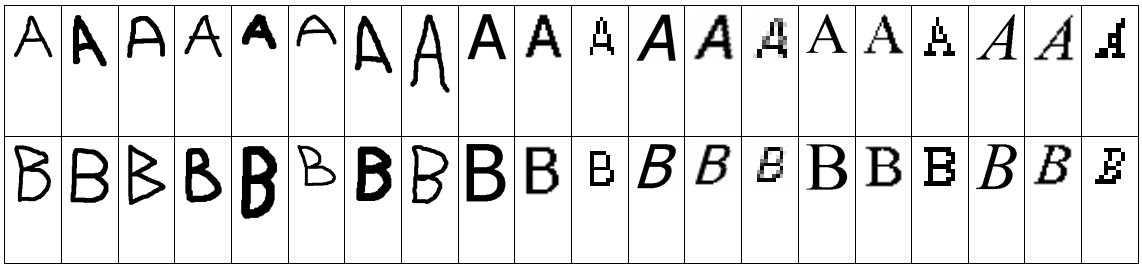


Figure 20: Character database example

I am going to release my character image set as an open database that may be used for other open-source character recognition projects. The idea is to create a common format and a good interface so that anybody could easily add their handwritings to the database. Though it is still have to be done.

8.Implementation

All algorithms described in this thesis are implemented in programming language Java using object oriented approach. The implementation consists of 36 classes and 2179 lines of code.

The structure of the OCR system is made so that all its parts are functioning independently from each other, using a common interfaces to interact. This allows any part to be painlessly replaced or changed without touching the rest of the system.

Usage

From users point of view system consists of two main tools: network training tool and character recognition tool. To create a functional OCR user should follow 4 simple steps.

1. Create a character database. Database is just a set of image files in PNG format, file names consist of the character which is represented by the image with "png" extension. Like A.png, B.png, C.png etc. Each image file may contain any number of character glyph variants.

2. Configure the network teaching tool. Set all the required parameters: number of layers in the network, number of neurons in each layer, activation function, network learning rate, feature extraction algorithm and its specific parameters.

3. Run the network training tool and wait for some time until the network error value reaches the appropriate level. 10^{-5} - 10^{-10} seems to be an appropriate value. This step may take a lot of time: up to several days, depending on the computer power, network complexity and character database size. (If initial parameters are not suitable, the error value may not reach the low values at all. If learning process is stuck it is worth returning to the step 2 and for example increase number of layers in the network) When training process is finished, the result is saved into a file, which contains all information about feature extractor, neural network and alphabet.

4. Now the character recognition tool can be used to open the file which was saved by training tool and reuse the stored information.

Any number of neural networks can be stored and loaded in the same program at the same time, this allows to easily make recognition competitions among different algorithms.

Storage

The file format, used to store the training results is a Java specific object serialization format. This format is very flexible and it is not specific to a file storage, objects can be stored to a database or streamed through the network etc.

But this format has its disadvantages: stored objects can not be loaded if object classes were changed, in other words neural network trained with one version of the library can not be used by other versions.

UML diagrams

Sequence diagram

In Figure 24 is provided a sequence diagram of the network creation and character recognition process. It represents steps 2-4 of the previous section of this thesis.

As user initializes the training process, ***OCRTeacher*** cycles through the characters provided by ***OCRTeachingCourse*** and allows ***LearningNeuralNetwork*** to learn from them. At the end of the training process, result is stored by ***OCRSerializer***.

Recognition process starts with reading the saved parameters of neural network and the Eye, after which same instances of **LearningNeuralNetwork** and Eye are used by ***OCR*** to perform the recognition.

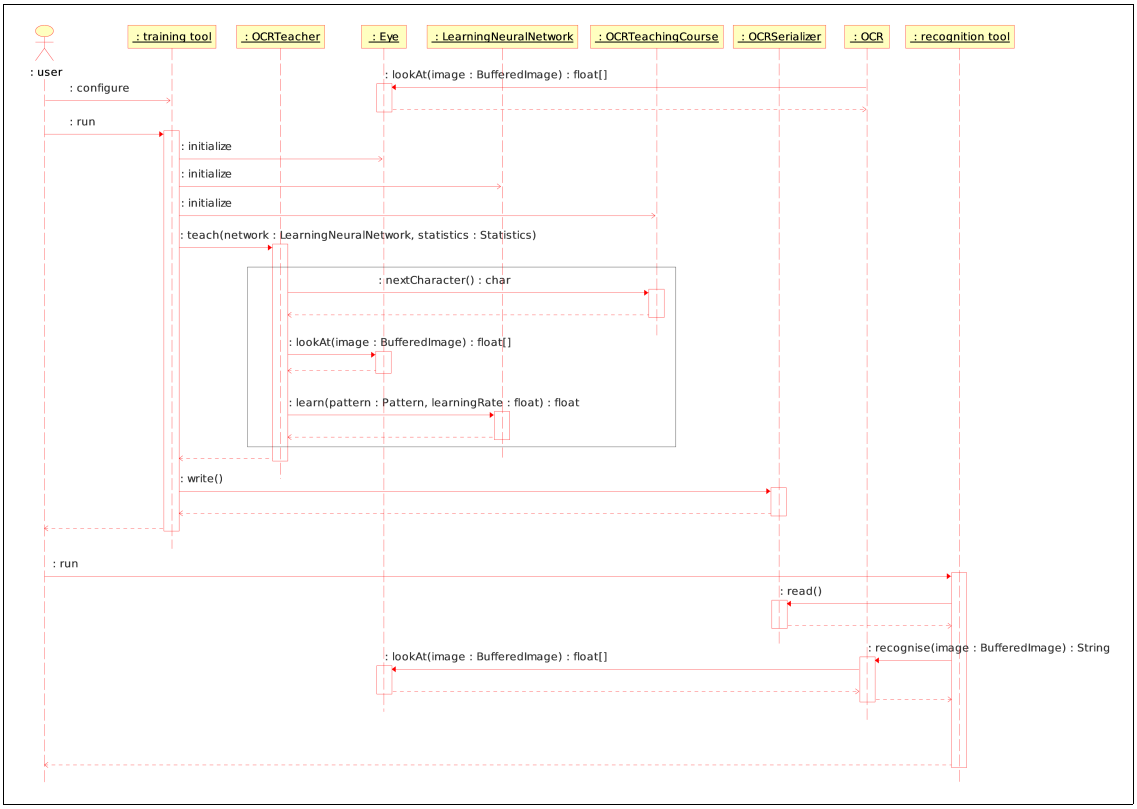


Figure 21: Neural network creation and character recognition sequence diagram

Class diagrams

In this section are presented class diagrams which describe relations among main classes of the project.

Figure 22 describes stricture of the neural network itself. The right part of the diagram is a simple neural network which is unable to learn and can be only used for recognition. Main interface ***NeuralNetwork*** is implemented by ***NeuralNetworkImpl*** class containing ***Neuron*** objects inside it.

On the left side of the diagram presented classes which are responsible for network learning ability. Main interface ***LearningNeuralNetwork*** extends ***NeuralNetwork*** and has one implementation: ***BackPropagationLearningNeuralNetwork*** which is a realization of ANN learning approach described in Classification section of the given thesis (see page

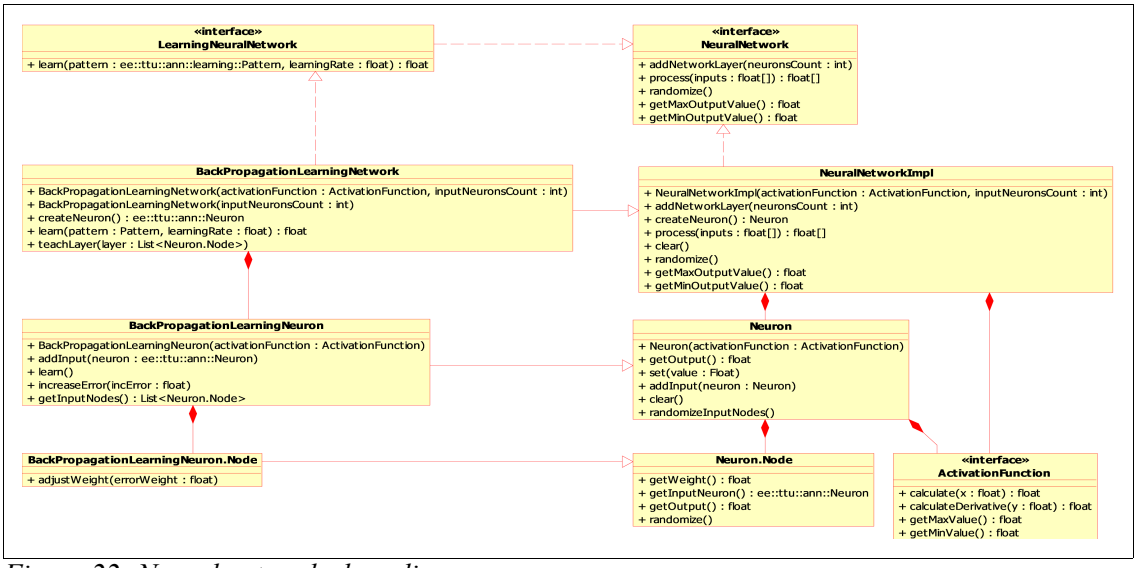


Figure 22: Neural network class diagram

28).

In Figure 23 is shown that we have 3 different realizations of feature extractor. The common interface, called ***Eye***, is implemented by ***SquareRegionDensityEye***, ***LocalContourDirectionEye*** and ***RandomReceptorEye***. Descriptions of there implementations can be found in Feature extraction section on the page 19.

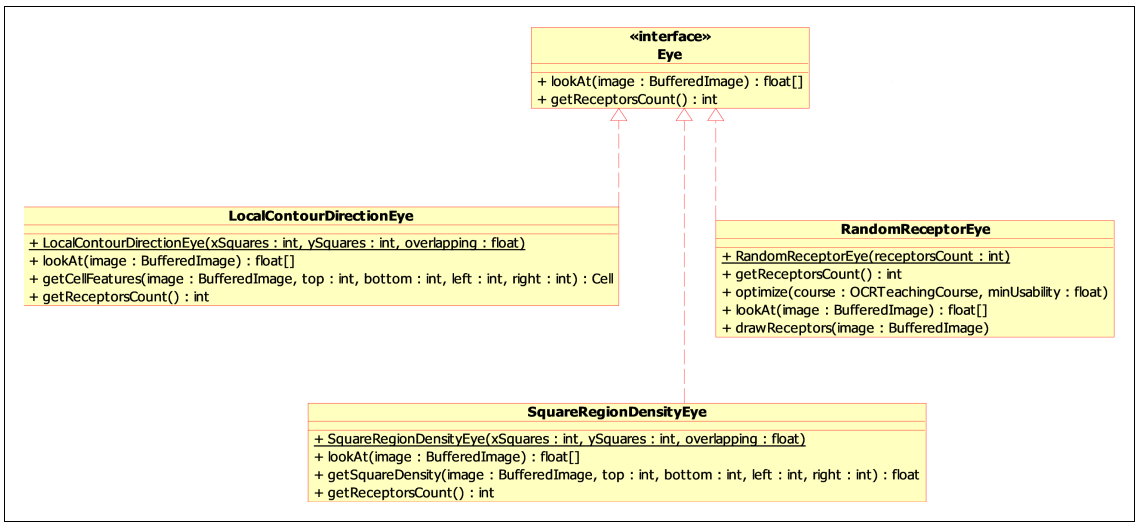


Figure 23: Feature extraction class diagram

Activation function implementations structure is described in Figure 24. Descriptions of

these functions can be found on page 34.

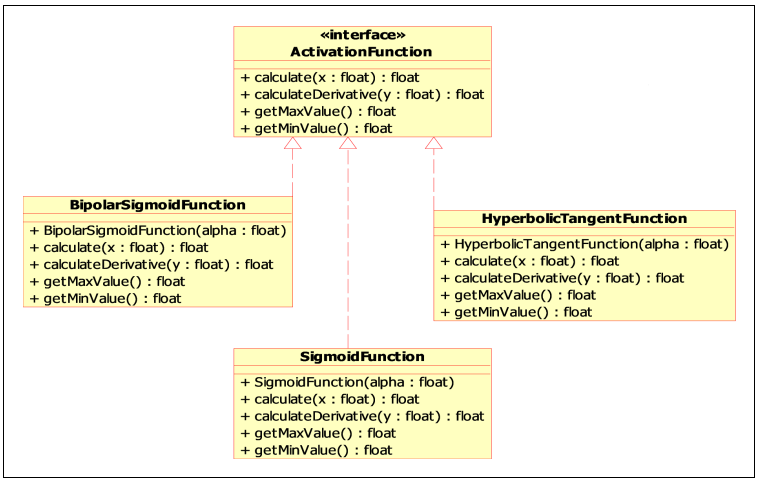


Figure 24: Activation function class diagram

Figure 25 describes the ANN teacher structure. then main class ***OCRTeacher*** operates with ***LearningNeuralNetwork***, ***Eye***, ***OCRTeachingCourse*** and ***Statistics*** objects in the neural educational process. ***OCRTeachingCourse*** is a container for character database and it is responsible for generation of teaching patterns. Statistics is a simple callback interface which is used by the program which initializes the teaching process, to receive some progress information.

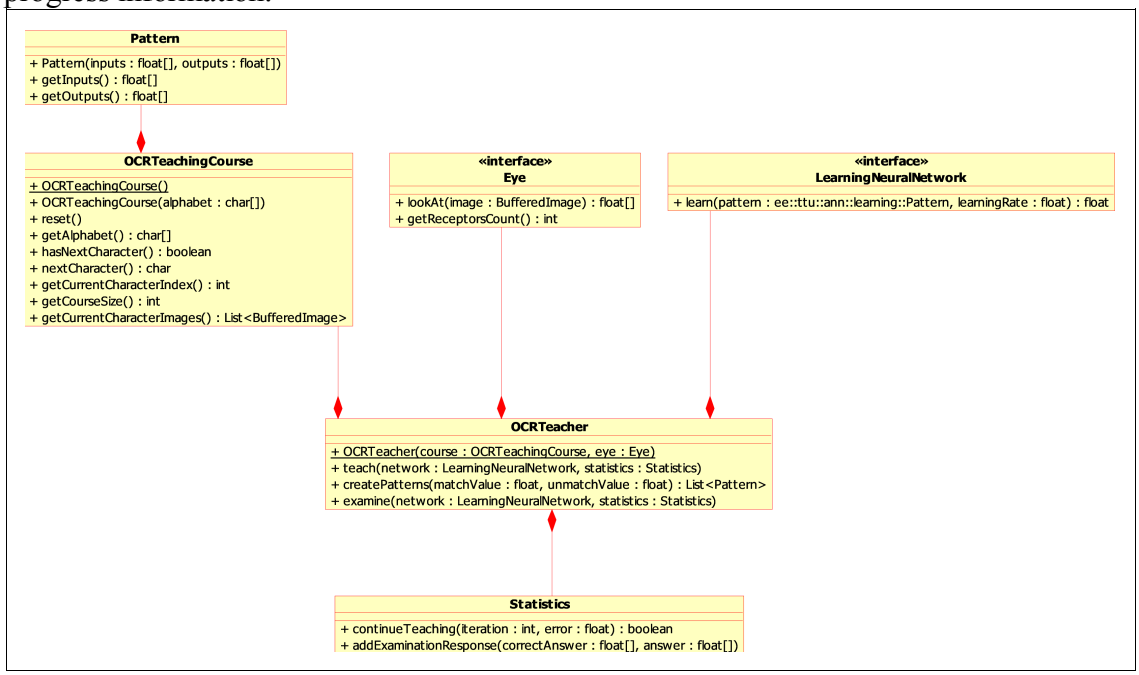


Figure 25: Network teacher class diagram

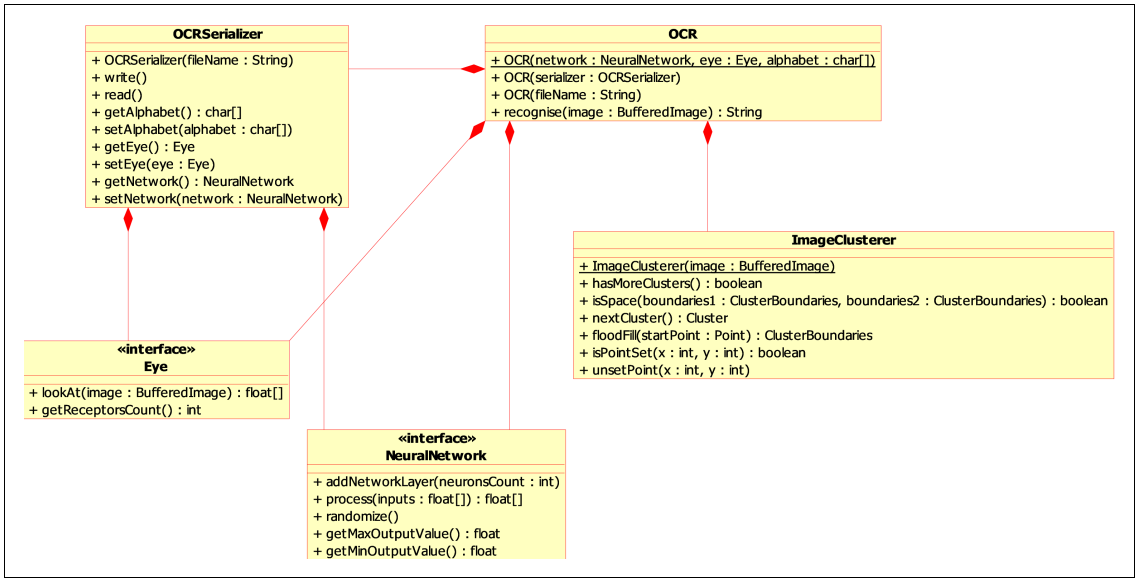


Figure 26: OCR class diagram

Last diagram (Figure 26) shows the character recognizer. Man class ***OCR*** contains one simple method *recognise()* which takes the image object as a parameter and returns the recognized text as a string. ***OCR*** class operates with ***Eye***, ***NeuralNetwork*** (note that in this case network has no learning ability because it is not required), ***ImageClusterer*** and ***OCRSerializer***. ***ImageClusterer*** is a realization of clusterization approach described on page 15. ***OCRSerializer*** is a class responsible for storing and loading the neural network from file, so ANN which was educated by ***OCRTeacher*** can be saved to file and reused later.

IDE

Eclipse integrated development environment was used in this project. Eclipse is one of the most popular Java development environments at the moment, it is free and open source. Eclipse can be freely downloaded from it's official website [8].

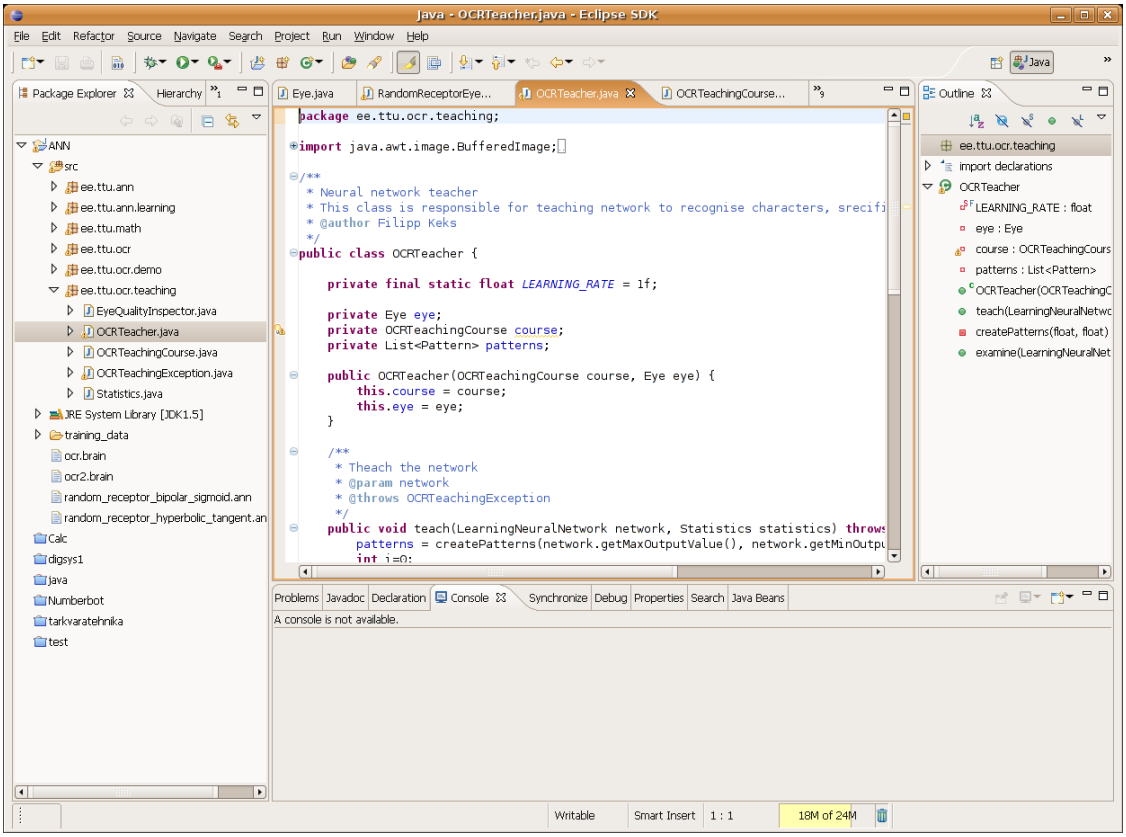


Figure 27: Eclipse in action

Source code

The source code of this project is freely available for anybody to download on the SourceForge – a popular open-source software development portal [9]. Project is registered under name ***ocr4j***, which stands for Optical Character Recognition For Java.

License

Source code is released under LGPL (Lesser General Public License) open-source license. Which means that anybody can use or modify my program freely and all additions and modifications will be also available under LGPL license. My source code can not be a part of any commercial product, but can be used in commercial products as a library without modifications. More detailed description of LGPL license can be found on the GNU official website [10].

Website

The project has it's own website http://ocr4j.sourceforge.net/

Website contains a demo applet which allows visitors to draw characters and try how the library works without downloading and compiling the source code.

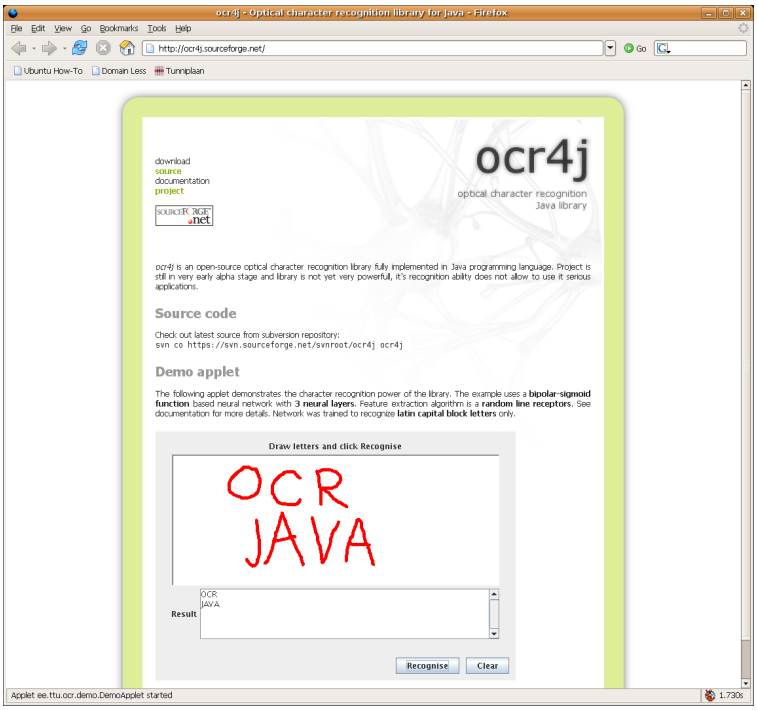


Figure 28: Project website look

9.Results

I've tried different combinations of algorithms described in this thesis and the best combination was Random receptor feature extraction algorithm with Bipolar sigmoid activation function based neural network with **315 neurons** divided into **4 layers**. This combination was the fastest learning and most accurate in image recognition.

Neural network training process took about **5 minutes** on **AMD Athlon 3800** processor. The final error value was near to 10^{-8} . I've also performed some test to measure the quality of character recognition:

Examination data	**Number of samples**	**Mistakes**	**Quality**
Same data that was used in network training process	520	0	100%
Handwritten characters that were never seen by the network before	130 (5 variants for each letter)	18	86%
Printed characters of fonts, styles and sizes that were not used in training process	936 (36 images of each letter)	19	98%

The recognition quality is somewhat not bad, but it is still not on the industrial level. Classifier makes mistakes sometimes even when image content is obvious for human eye. The reasons for this are: very limited character database and algorithms limitations.

Other things which library still lacks:

- Current clusterization algorithm is not able to find characters that consist of several separate parts like "i" or "j".
- Ability to recognize handwritings with connected characters.
- Noise reduction.
- Neural network training speed optimization (Using for example the NOVEL optimization method [11])

10.Bibliography

[1] Artificial Intelligence Tutorial Review - Eyal Reinhold, Johnathan Nightingale 1999 http://www.psych.utoronto.ca/~reingold/courses/ai/

[2] Neural Network OCR - Andrew Kirillov 2005 http://www.codeproject.com/csharp/neural_network_ocr.asp

[3] Simple and Effective Feature Extraction for Optical Character Recognition - Juan-Carlos Perez, Enrique Vidal

[4] Machine Learning, Neural and Statistical Classification - D. Michie, D.J. Spiegelhalter, C.C. Taylore 1994 http://www.amsta.leeds.ac.uk/~charles/statlog/

[5] The Sensory Order - Friedrich Hayek 1952

[6] The Roots of Backpropagation - Paul Werbos 1974

[7] National Institute of Standards and Technology - http://www.nist.gov

[8] Eclipse IDE official website - http://www.eclipse.org

[9] SourceForge open-source software development portal - http://www.sourceforge.net

[10] GNU Lesser General Public License - http://www.gnu.org/licenses/lgpl.html

[11] Global Optimization for Neural Network Training - Yi Shang, Benjamin W. Wah 1996 University of Illinois