Application of Neural Networks for Gesture Recognition

Mohammad Kamali Moghaddam

Electrical Department, Ghuchan University of advance technology
MohammadKamaliMoghaddam@chmail.ir

Abstract: Using orientation histograms a simple and fast algorithm will be developed to work on a workstation. It will recognize static hand gestures, namely, a subset of American Sign Language (ASL). Previous systems have used datagloves or markers for input in the system. A pattern recognition system will be using a transform that converts an image into a feature vector, which will then be compared with the feature vectors of a training set of gestures. The final system will be implemented with a Perceptron network.

Keywords: Neural Networks for Gesture Recognition

1-   Introduction

Since the introduction of the most common input computer devices not a lot have changed. This is probably because the existing devices are adequate. It is also now that computers have been so tightly integrated with everyday life, that new applications and hardware are constantly introduced. The means of communicating with computers at the moment are limited to keyboards, mice, light pen, trackball, keypads etc

These devices have grown to be familiar but inherently limit the speed and naturalness with which we interact with the computer. As the computer industry follows Moore's Law since middle 1960s, powerful machines are built equipped with more peripherals. Vision based interfaces are feasible and at the present moment the computer is able to "see". Hence users are allowed for richer and user-friendlier man-machine interaction. This can lead to new interfaces that will allow the deployment of new commands that are not possible with the current input devices.

Recently, there has been a surge in interest in recognizing human hand gestures. Handgesture recognition has various applications like computer games, machinery control (e.g. crane), and thorough mouse replacement. One of the most structured sets of gestures belongs to sign language. In sign language, each gesture has an assigned meaning (or meanings).

Computer recognition of hand gestures may provide a more natural-computer interface, allowing people to point, or rotate a CAD model by rotating their hands. Hand gestures can be classified in two categories: static and dynamic. A static gesture is a particular hand configuration and pose, represented by a single image. A dynamic gesture is a moving gesture, represented by a sequence of images. We will focus on the recognition of static images.

Interactive applications pose particular challenges. The response time should be very fast. The user should sense no appreciable delay between when he or she makes a gesture or motion and when the computer responds. The computer vision algorithms should be reliable and work for different people.

There are also economic constraints: the vision-based interfaces will be replacing existing ones, which are often very low cost. A hand-held video game controller and a television remote control each cost about $40. Even for added functionality, consumers may not want to spend more. When additional hardware is needed the cost is considerable higher.

Academic and industrial researchers have recently been focusing on analyzing images of people. While researchers are making progress, the problem is hard and many present day algorithms are complex, slow or unreliable. The algorithms that do run near real-time do so on computers that are very expensive relative to the existing hand-held interface devices.

2-   Literature review

Excellent work has been done in support of machine sign language recognition by *Sperling and Parish*, who have done careful studies on the bandwidth necessary for a sign conversation using spatially and temporally sub-sampled images. Point light experiments (where "lights" are attached to significant locations on the body and just these points are used for recognition), have been carried out by *Poizner*. Most systems to date study isolate/static gestures. In most of the cases those are finger spelling signs.

2-1-   Object Recognition

In some interactive applications, the computer needs to track the position or orientation of a hand that is prominent in the image. Relevant applications might be computer games, or interactive machine control. In such cases, a description of the overall properties of the image, may be adequate. Image moments, which are

fast to compute, provide a very coarse summary of global averages of orientation and position. If the hand is on a uniform background, this method can distinguish hand positions and simple pointing gestures.

The large-object-tracking method makes use of a low-cost detector/processor to quickly calculate moments. This is called the artificial retina chip. This chip combines image detection with some low-level image processing (named artificial retina by analogy with those combined abilities of the human retina). The chip can compute various functions useful in the fast algorithms for interactive graphics applications.

2-2-　　Shape recognition

Most applications, such as recognizing particular static hand signal, require a richer description of the shape of the input object than image moments provide. If the hand signals fell in a predetermined set, and the camera views a close-up of the hand, we may use an example-based approach, combined with a simple method top analyze hand signals called orientation histograms.

These example-based applications involve two phases; training and running. In the training phase, the user shows the system one or more examples of a specific hand shape. The computer forms and stores the corresponding orientation histograms. In the run phase, the computer compares the orientation histogram of the current image with each of the stored templates and selects the category of the closest match, or interpolates between templates, as appropriate. This method should be robust against small differences in the size of the hand but probably would be sensitive to changes in hand orientation.

3-　　Proposed method

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown in fig(2). There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this *supervised learnin*g (training method studied in more detail on following chapter), to train a network.

Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems.

Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. The supervised training methods are commonly used, but other networks can be obtained from *unsupervised training* techniques or from direct *design* methods. Unsupervised networks can be used, for instance, to identify groups of data. Certain kinds of linear networks and Hopfield networks are designed directly. In summary, there are a variety of kinds of design and learning techniques that enrich the choices that a user can make.
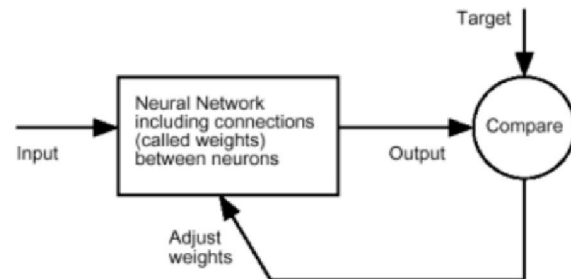

Figure 1: Neural Net block diagram

The field of neural networks has a history of some five decades but has found solid application only in the past fifteen years, and the field is still developing rapidly. Thus, it is distinctly different from the fields of control systems or optimization where the terminology, basic mathematics, and design procedures have been firmly established and applied for many years.

3-1- Neuron Model

A neuron with a single scalar input and no bias is shown on the left below.

The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w, to form the product wp, again a scalar. Here the weighted input wp is the only argument of the transfer function f, which produces the scalar output a. The neuron on the right has a scalar bias, b. You may view the bias as simply being added to the product wp as shown by the summing junction or as shifting the function f to the left by an amount b. The bias is much like a weight, except that it has a constant input of 1. The transfer function net input n, again a scalar, is the sum of the weighted input wp and the bias b. This sum is the argument of the transfer function f. Here f is a transfer function, 15 typically a step function or a sigmoid function, that takes the argument n and produces the output a. Examples of various transfer functions are given in the next section. Note that w and b are both *adjustable* scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior.

Thus, we can train the network to do a particular job by adjusting the weight or bias parameters, or

perhaps the network itself will adjust these parameters to achieve some desired end. All of the neurons in the program written in MATLAB have a bias.

However, you may omit a bias in a neuron if you wish. A

3-2- Supervised Learning

Supervised learning is based on the system trying to predict outcomes for known examples and is a commonly used training method. It compares its predictions to the target answer and "learns" from its mistakes. The data start as inputs to the input layer neurons. The neurons pass the inputs along to the next nodes. As inputs are passed along, the weighting, or connection, is applied and when the inputs reach the next node, the weightings are summed and either intensified or weakened. This continues until the data reach the output layer where the model predicts an outcome. In a supervised learning system, the predicted output is compared to the actual output for that case. If the predicted output is equal to the actual output, no change is made to the weights in the system. But, if the predicted output is higher or lower than the actual outcome in the data, the error is propagated back through the system and the weights are adjusted accordingly. This feeding errors backwards through the network is called "back-propagation." Both the Multi-Layer Perceptron and the Radial Basis Function are supervised learning techniques. The Multi-Layer Perceptron uses the back-propagation while the Radial Basis Function is a feed-forward approach which trains on a single pass of the data.

3-3-*Advantages of Neural Computing*

There are a variety of benefits that an analyst realizes from using neural networks in their work. Pattern recognition is a powerful technique for harnessing the information in the data and generalizing about it. Neural nets learn to recognize the patterns which exist in the data set.

The system is developed through learning rather than programming. Programming is much more time consuming for the analyst and requires the analyst to specify the exact behavior of the model. Neural nets teach themselves the patterns in the data freeing the analyst for more interesting work.

Neural networks are flexible in a changing environment. Rule based systems or programmed systems are limited to the situation for which they were designed--when conditions change, they are no longer valid. Although neural networks may take some time to learn a sudden drastic change, they are excellent at adapting to constantly changing information.

Neural networks can build informative models where more conventional approaches fail. Because neural networks can handle very complex interactions they can easily model data which is too difficult to model with traditional approaches such as inferential statistics or programming logic.

Performance of neural networks is at least as good as classical statistical modeling, and better on most problems. The neural networks build models that are more reflective of the structure of the data in significantly less time.

Neural networks now operate well with modest computer hardware. Although neural networks are computationally intensive, the routines have been optimized to the point that they can now run in reasonable time on personal computers. They do not require supercomputers as they did in the early days of neural network research.

3-4- *Single Perceptron*

The perceptron is a program that learns **concepts**, i.e. it can learn to respond with *True* (1) or *False* (0) for inputs we present to it, by repeatedly "studying" examples presented to it.

The structure of a single perceptron is very simple. There are two inputs, a bias, and an output. A simple schematic diagram is shown in fig(5).

Note that both the inputs and outputs of a perceptron are binary - that is they can only be 0 or 1.
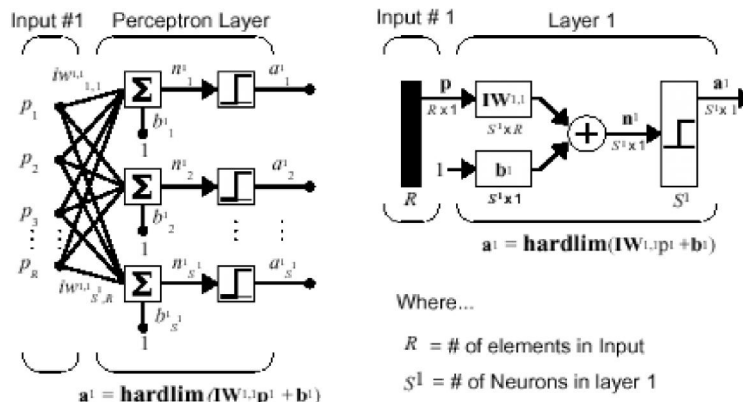

$$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$$

$$\mathbf{a}^1 = \mathbf{hardlim}(\mathbf{IW}^{1,1}\mathbf{p}^1 + \mathbf{b}^1)$$

Where...

R = # of elements in Input

S^1 = # of Neurons in layer 1

Figure 2: Perceptron Schematic Diagram

Each of the inputs and the bias is connected to the main perceptron by a weight. A weight is generally a real number between 0 and 1. When the input number is fed into the perceptron, it is multiplied by the corresponding weight. After this, the weights are all summed up and fed through a hard-limiter. Basically, a hard-limiter is a function that defines the threshold values for 'firing' the perceptron. For example, the limiter could be:

For example, if the sum of the input multiplied by the weights is -2, the limiting function would return 0. Or if the sum was 3, the function would return 1.

Now, the way a perceptron learns to distinguish patterns is through modifying its weights

- the concept of a learning rule must be introduced. In the perceptron, the most common form of learning is by adjusting the weights by the *difference between the desired output and the actual output*. Mathematically, this can be written:

Learning on a perceptron is guaranteed, as stated by the Perceptron Convergence Theorem which states that if a solution can be implemented on a perceptron, the learning rule will find the solution in a finite number of steps.

Proof of this theorem can be found in *Minsky and Papert's* 1989 book, *Perceptrons.*

3-5- The Perceptron Convergence Algorithm

For the development of the error-correction learning algorithm for a single-layer perceptron, we will work with the signal-flow graph shown in fig(6). In this case the threshold $\theta(n)$ is treated as a synaptic weight connected to a fixed input equal to -1.
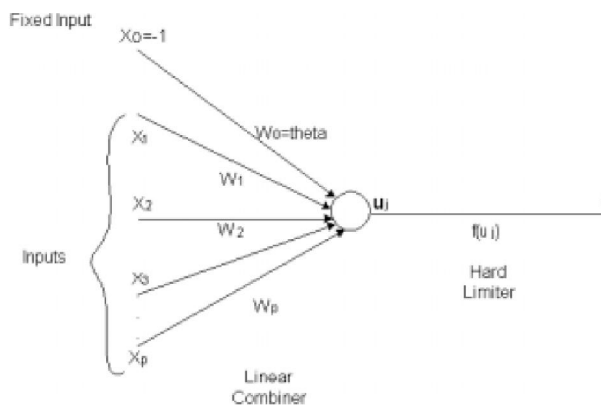

Figure 3: Perceptron Signal Flow Graph

We may then define the (p+1)-by-1 input vector:
$$x(n) = [-1, x_1(n), x_2(n),\ldots, xp(n)]^T$$
Correspondingly we define the (p+1)-by-1 weight vector:
$$w(n) = [\theta(n), w_1(n), w_2(n),\ldots, wp(n)]^T$$
Below are some variable and parameters used in the convergence algorithm
$\theta(n)$=threshold

$y(n)$ = actual response
$d(n)$ = desired response
η = learning rate parameter, $0< \eta <1$
So lets see the 4-step algorithm in greater detail:

Step 1: Initialization

Set w(0)=**0.** Then perform the following computations for time $n=1,2,\ldots.$

Step 2: Activation

At time n, activate the perceptron by applying continuous-valued input vector $x(n)$ and desired response $d(n)$.

Step 3: Computation of Actual Response

Compute the actual response of the perceptron:
$y(n)$ =sgn[$w^T(n)x(n)$]
The linear output is written in the form:
$u(n)$ =wT$(n)x(n)$
where :
sgn(u) = +1 if u>0
sgn(u) = -1 if u<0

Step 4: Adaptation of Weight Vector

$w(n+1)$ =$w(n)$ +$\eta[d(n) -y(n)]x(n)$
where $d(n)$= +1 if x(n) belongs to class C_1
where $d(n)$= -1 if x(n) belongs to class C_2

Step 5

Increment time *n* by one unit and go back to step 2

Perceptrons can only solve problems where the solutions can be divided by a line (or hyperplane) - this is called linear separation. To explain the concept of linear separation further, let us look at the function shown to the left. The function reads *'x1 and (not x2)'.* Let us assume that we run the function through a perceptron, and the weights converge at 0 for the bias, and 2, -2 for the inputs. If we calculate the *net* value (the weighted sum) we get:
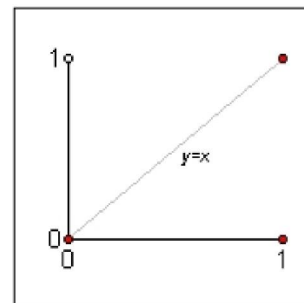
Table 1: 'x1 and not x2

Figure 4: Linear Separable

So the perceptron correctly draws a line that divides the two groups of points. Note that it doesn't only have to be a line, it can be a hyperplane dividing points in 3-D space, or beyond. This is where the power of perceptrons lies, but also where its limitations lie. For example, perceptrons cannot solve the XOR binary functions.

4-Approach

The starting point of the project was the creation of a database with all the images that would be used for training and testing.

The image database can have different formats. Images can be either hand drawn, digitized photographs or a 3D dimensional hand. Photographs were used, as they are the most realistic approach.

Images came from two main sources. Various ASL databases on the Internet and photographs I took with a digital camera. This meant that they have different sizes, different resolutions and some times almost completely different angles of shooting. Images belonging to the last case were very few but they were discarded, as there was no chance of classifying them correctly. Two operations were carried out in all of the images. They were converted to grayscale and the background was made uniform. The internet databases already had uniform backgrounds but the ones I took with the digital camera had to be processed in *Adobe Photoshop*.

Drawn images can still simulate translational variances with the help of an editing program (e.g. *Adobe Photoshop*).

The database itself was constantly changing throughout the completion of the project as it was it that would decide the robustness of the algorithm. Therefore, it had to be done in such way that different situations could be tested and thresholds above which the algorithm didn't classify correct would be decided.

The construction of such a database is clearly dependent on the application. If the application is a crane controller for example operated by the same person for long periods the algorithm doesn't have to be robust on different person's images. In this case noise and motion blur should be tolerable. The applications can be of many forms and since I wasn't developing for a specific one I have tried to experiment for many alternatives. We can see an example below. In the first row are the training images. In the second the testing images.

Train image 1 Train image 2 Train image 3
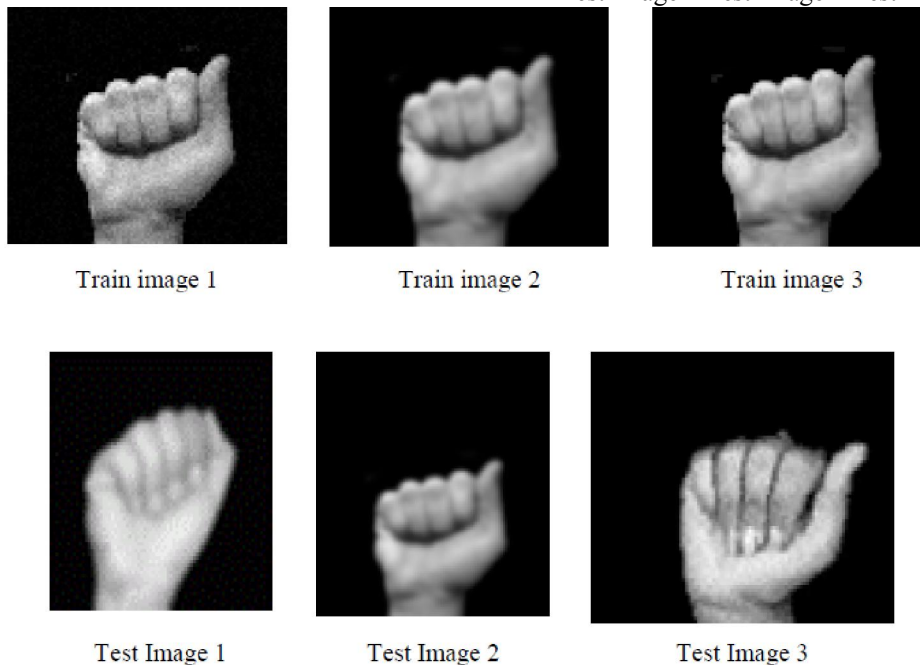Test Image 1 Test Image 2 Test Image 3

Train image 1   Train image 2   Train image 3

Test Image 1   Test Image 2   Test Image 3

Figure 5: Train – Test images

For most of the gestures the training set originates from a single gesture. Those were enhanced in *Adobe Photoshop* using various filters. The reason for this is that I wanted the algorithm to be very robust for images of the same database. If there was a misclassification to happen it would be preferred to be for unknown images.

The final form of the database is this.

Train set:

Eight training sets of images, each one containing three images. Each set originates from a single image for testing.

Test Set:

The number of test images varies for each gesture. There is no reason for keeping those on a constant number. Some images can tolerate much more variance and images from new databases and they can be tested extensively, while other images are restricted to fewer testing images.

44

Train Set sample

The databases are shown in more detail in the results section where all the training and testing images are listed.

The system could be approached either in high or low-level. The former would employ models of the hand, finger, joints and perhaps fit such a model to the visual data. This approach offers robustness, but at the expense of speed.

A low-level approach would process data at a level not much higher than that of pixel intensities.

Although this approach would not have the power to make inferences about occluded data, it could be simple and fast. The pattern recognition system that will be used can be seen in Fig (9). Some transformation T, converts an image into a feature vector, which will be then compared with feature vectors of a training set of gestures.

Table 2: Train Set sample

0	Original	Blur radius (pixels)	Noise
1	✓	✗	✗
2	✗	0.5	✗
3	✗	✗	10 Uniform

1	Original	Blur radius (pixels)	Noise
1	✓	✗	✗
2	✗	0.4	✗
3	✗	✗	10 Gaussian

H	Original	Blur radius (pixels)	Noise
1	✓	✗	✗
2	✗	0.6	✗
3	✗	✗	6 Gaussian

We will be seeking for the simplest possible transformation T, which allows gesture recognition.

Histogram orientation has the advantage of being robust in lighting change conditions. If we follow the pixel-intensities approach certain problems can arise for varying illumination. Taking a pixel-by-pixel difference of the same photo under different lighting conditions would show a large distance between these two identical gestures. For the pixel-intensity approach no transformation T has been applied. The image itself is used as the feature vector. In Fig (10) we can see the same hand gesture under different lighting conditions.

Another important aspect of gesture recognition is translation invariance. The position of the hand within the image should not affect the feature vector. This could be enforced by forming a local histogram of the local orientations. This should treat each orientation element the same, independent of location. Therefore, orientation analysis should give robustness in illumination changes while histogramming will offer translational invariance. This method will work if examples of the same gesture map to similar orientation histograms, and different gestures map to substantially different histograms.
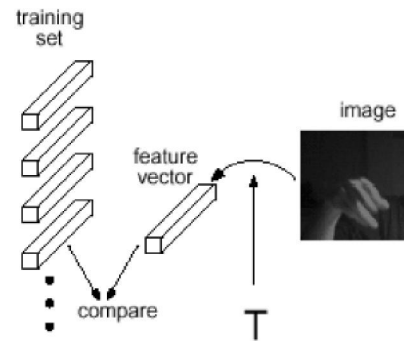

Figure 6: Pattern Recognition System

Figure 7: Illumination Variance

4-1 Orientation Histograms

We want gestures to be the same regardless of where they occur with the images boarders. To achieve this we will ignore position altogether, and tabulate a histogram of how often each orientation element occurred in the image. Clearly, this throws out information and some distinct images will be confused by their orientation histograms. In practice, however, one can choose a set of training gestures with substantially different orientation histograms from each other.

One can calculate the local orientation using image gradients. I used two 3 – tap x and y derivative filters. The outputs of the x and y derivative operators will be dx and dy. Then the gradient direction is **atan (dx, dy).** I had decided to use the edge orientation as the only feature that will be presented to the neural network. The reason for this is that if the edge detector was good enough it would have allowed me to test the network with images from different databases. This would lead though to testing the algorithm with only similar images. Apart from this the images before resized should be of approximately the same size. This is the size of the hand itself in the canvas and not the size of the canvas. Once the image has been processed the output will be a single vector containing a number of elements equal to the number of bins of the orientation histogram.

Figure 8 shows the orientation histogram calculation for a simple image. Blurring can be used to allow neighboring orientations to sense each other.

4-2- Operation

The program can be 'divided' in 6 steps. Lets examine them one by one.

Step1

The first thing for the program to do is to read the image database. A *for* loop is used to read an entire folder of images and store them in MATLAB's memory. The folder is selected by the user from menus. A menu will firstly pop-up asking you whether you want to run the algorithm on test or train sets. Then a second menu will pop-up for the user to choose which ASL sign he wants to use.

Step2

Resize all the images that were read in Step1 to 150x140 pixels. This size seems the optimal for offering enough detail while keeping the processing time low.

Step3.

Next thing to do is to find the edges. As mentioned before 2 filters were used.

Step 4

Dividing the two resulting matrices (images) *dx* and *dy* element by element and then taking the atan ($\tan^{-1}$). This will give the gradient orientation.
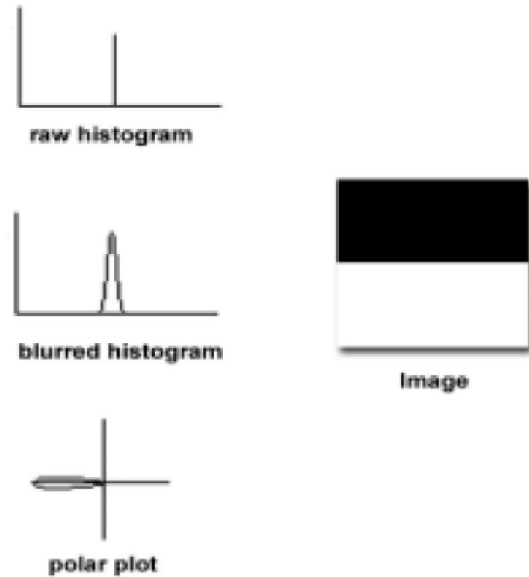

Figure 8 : Orientation histogram

Step 5

Then the MATLAB function *im2col* is called to rearrange the image blocks into columns. This is not a necessary step but it has to be done if we want to display the orientation histogram. *Rose* creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range. Each group is shown as one bin.

Step 6

Converting the column matrix with the radian values to degrees. This way we can scan the vector for values ranging from 0o to 90o . This is because for real elements of X, atan(X) is in the range .

This can also be seen from the orientation histograms where values come up only on the first and last quarter.

Determining the number of the histogram bins was another issue that was solved by experimenting with various values. I have tried with 18 20 24 and 36 bins. What I was looking for was the differentiation (or not) among the images. At the same time I was thinking of the neural network itself as this vector would be the input to the network. The smaller the vector the faster the processing. Finally, the actual resolution of each bin was set to 10o , which means 19 bins.

The algorithms development was organized having in mind MATLAB weaknesses. The major one is speed. MATLAB is perfect for speeding up the development process but it can be very slow on execution when bad programming practices have been employed.

Nested loops slow down the program considerably. It is probably because MATLAB is built

on loops. Therefore, unnecessary back-tracking was avoided and even some routines were written in full instead of using *for* loops. The code is not much in any case.

The same techniques were put in practice for the following program as well.

4-3 Linear Classification

Linear networks can be trained to perform linear classification with the function **train**.

This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs. Then the network is adjusted with the sum of all these corrections. We will call each pass through the input vectors an *epoch*. This contrasts with **adapt**, which adjusts weights for each input vector as it is presented.

Adapt is another function in MATLAB for training a neural network. I was using this at the first stages when I was using a back-propagation network. Their main difference is that with **train** only batch training (updating the weights after each presenting the complete data set) can be used, while with **adapt** you have the choice of batch and incremental training (updating the weights after the presentation of each single training sample). **Adapt** supports far less training functions. Since I didn't have a very good reason to go for incremental training I decided to use **train** which is more flexible.

Finally, train applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error.

**4-4- Input , Target , Test and Output files**

The input, test and target vectors are saved on the hard drive in text files. All data is stored in a single column. MATLAB can tell where one vector ends and another starts simply by writing so in the *fscanf* command as shown below.

TS1 = fscanf(fid,'%f',[**19**,inf]);

fid = fopen('target.txt','rt');

Formatting those text files can be time consuming but once the training and target files are ready the test files can be created when needed. The output is displayed on the command line. The error is calculated by subtracting the output A from target T. Then the sum-squared error is calculated. Below some error graphs of the network training are shown.
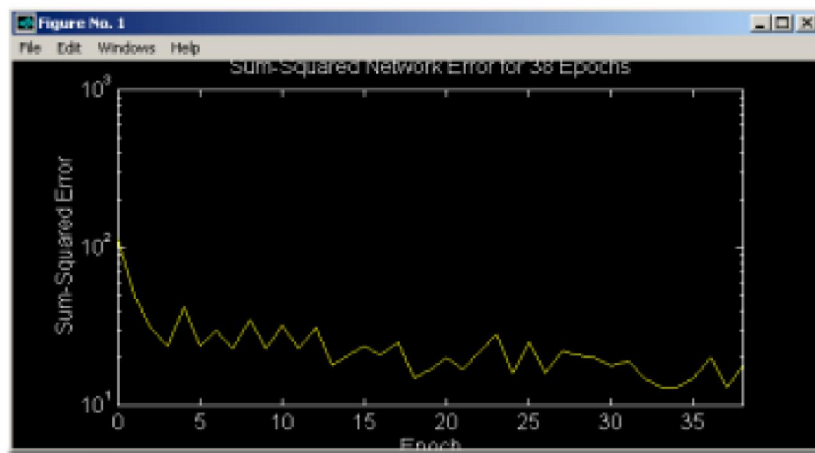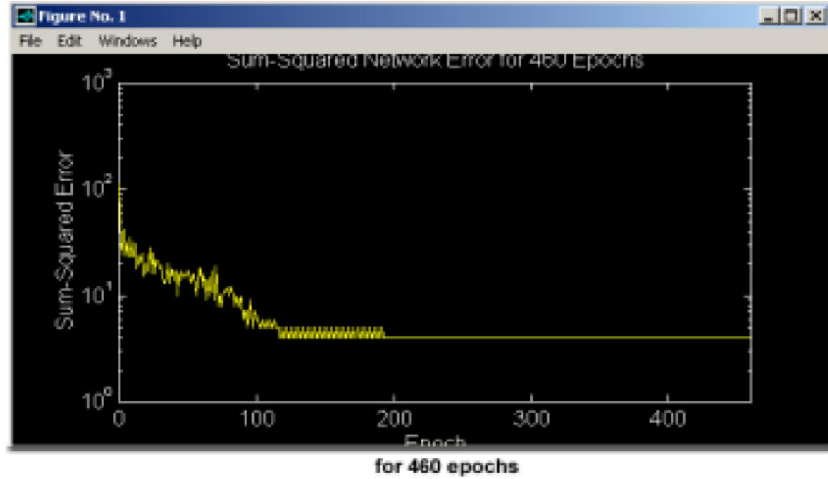

for 38 epochs
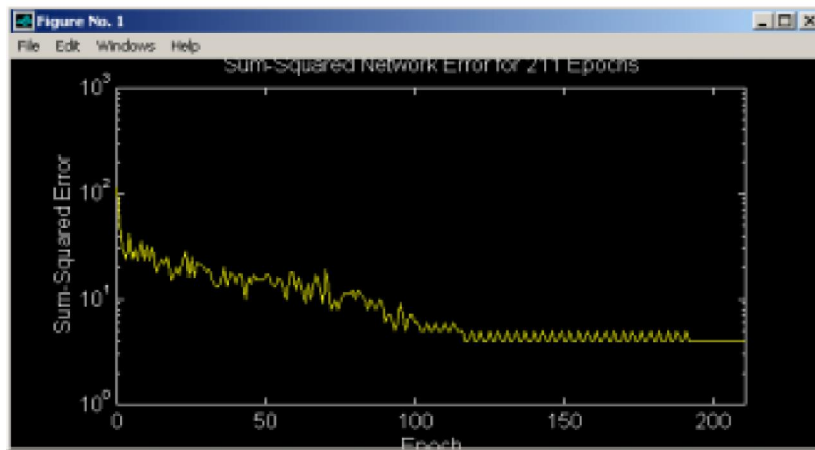
for 211 epochs

for 460 epochs
Figure 9: Train Error

Here is the error for the first 20 epochs of training
Training starts
[W2,b2,epochs,errors] = trainp (W2,b2,A1,T,TP);
TRAINP: 0/500 epochs, SSE = 86.
TRAINP: 1/500 epochs, SSE = 42.
TRAINP: 2/500 epochs, SSE = 24.
TRAINP: 3/500 epochs, SSE = 64.
TRAINP: 4/500 epochs, SSE = 42.
TRAINP: 5/500 epochs, SSE = 21.
TRAINP: 6/500 epochs, SSE = 46.
TRAINP: 7/500 epochs, SSE = 28.
TRAINP: 8/500 epochs, SSE = 21.
TRAINP: 9/500 epochs, SSE = 20.
TRAINP: 10/500 epochs, SSE = 36.
TRAINP: 11/500 epochs, SSE = 27.
TRAINP: 12/500 epochs, SSE = 20.
TRAINP: 13/500 epochs, SSE = 24.
TRAINP: 14/500 epochs, SSE = 22.
TRAINP: 15/500 epochs, SSE = 28.
TRAINP: 16/500 epochs, SSE = 23.
TRAINP: 17/500 epochs, SSE = 23.
TRAINP: 18/500 epochs, SSE = 15.
TRAINP: 19/500 epochs, SSE = 14.
TRAINP: 20/500 epochs, SSE = 19.
The error is constantly reducing until it converges to 0. When it does it will stop and start the testing process. In the case of reaching the 500 predetermined epochs without coming down to 0 it will test the network as is.

5-   Results
Here are some tables displaying the results obtained from the program. Sign images of the same letter are grouped together on every table. The table gives us information about the pre-processing operations that took place (i.e. blurring, noise, translation) and also if the image belongs to the same database with the training images.

The amount of each filter is also recorded so we can estimate the maximum values of noise the network can tolerate. This of course varies from image to image. The result also varies for every time the algorithm is executed. The variance is very small but it is there.

So we cannot easily draw conclusions and set a certain threshold above which we can tell that the network will not classify correctly. It all comes down to the application again.

Form of results:
The results come out of the network in column format. Each column is a classified image vector. The position of the '1' in the vector among the '0s' indicates which sign it is.

Therefore there should be only one '1' in every vector, but this is not always the case. As you will see from the tables below there are situations that the perceptron cannot converge to a single solution but it gives two possible solutions. In almost all of those cases one of the classifications is correct. There are few others though that the vector is not classified at all.

This is a table with all the target vectors that have used to train the network and confirm correct classification or not.

Table 3: Target Vectors

0	1	5	A	H	L	V	W
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Below you can see some result vectors as they come out of MATLAB: This is a test set for '*L*'.

a2 =

0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 1 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Therefore the 1st, 4th, 5th, 7th, 8th, 10th, 11th and12th vectors are correctly classified. The rest are classified as either '*L*' or '*A*'.

All the filtering operations were performed in *Adobe Photoshop 5.0.* hence the amount of each one was measured.

_ Blur – Measured in pixel radius

_ Motion Blur – Like taking photo of moving car. Specify angle and blur radius

_ Noise - Either Uniform or Gaussian (stronger)

_ Same Image – From the same image database

_ Translated – move the signaling hand in the canvas to test translational invariance

_ Classified

_ : correctly classified

_ : not classifed

W : classified as '*W*' (wrongly)

L – a : classified as either an '*L*' or '*A*' (Upper case correct)

I would like to clarify here that for any test set, image that is noted as coming from other databases is the only one. Therefore every image that is indicated so represents a specific database on each own.

Below are the result tables. They start with digit '*0*' and continue with the rest of the digits and letters.

Testing – **0 (Zero)**
Figure 10: Zero
Image Blur(Radius) Noise Same Image Translated Classified

Testing – **1 (One)**
Figure 11: One
Image Blur Noise Same Image Translated Classified

Table 4: Zero Test Results

Image	Blur(Radius)	Noise	Same Image	Translated	Classified
1	-	-	NO	-	✓
2	-	18 u	Yes	-	✓
3	-	25 u	Yes	-	✓
4	-	25 g	Yes	-	✓
5	-	28 g	Yes	Yes	✓
6	1.0	-	Yes	-	✓
7	1.2	-	Yes	-	✓
8	1.4	-	Yes	-	✓
9	1.8	-	Yes	Yes	✓
10	-	15 u	Yes	-	✓
11	-	8 g	Yes	-	✓
12	0.8	-	Yes	-	✓
13	1.0	-	Yes	Yes	✓
14	2.0	-	Yes	-	✗
15	1.0	-	NO	-	1
16	-	-	NO	-	✓
17	-	-	NO	-	1

For '0' the classification error is very small. For the 5th image the amount of noise (28 gaussian) is very high but it still classifies correctly. It will not tolerate blurring above a pixel radius of 2.0 though.

On the other hand translation doesn't seem to cause any problems.

Table 5: One Test Results

Image	Blur	Noise	Same Image	Translated	Classified
1	Sharpen	-	Yes	Yes	✓
2	Sharpen More	-	Yes	-	✗
3	1.2	-	Yes	-	✓
4	1.5	-	Yes	-	0
5	2.0	10 g	Yes	Yes	5
6	-	15 g	Yes	-	✓
7	-	20 g	Yes	-	0
8	-	22 u	Yes	-	✓
9	-	25 u	Yes	-	0
10	1.0	-	Yes	-	✓
11	-	20 u	Yes	-	✓
12	Motion blur 90degrees, 2	-	NO	Yes	✓
13	-	-	NO	Yes	✓
14	-	-	NO	-	0

Sharpening causes plenty of distortion on the images, especially on the edges. That is way not much sharpening will be tolerated. We can see from the table above that for great amounts of noise or blurring the perceptron classifies '*1*' signs as '*0s*'.

On image 5 a combination of blurring and gaussian noise is causing the network to classify the sign completely wrongly.

Testing – L
Figure 12: L
Image Blur Noise Same Image Translated Classified

Table 6: L Test Results

Image	Blur	Noise	Same Image	Translated	Classified
1	-	12 g	Yes	-	✓
2	-	15 g	Yes	-	L-a
3	-	20 g	Yes	-	L-a
4	-	10 u	Yes	-	✓
5	-	15 u	Yes	Yes	✓
6	-	20 u	Yes	Yes	L-a
7	1.2	-	Yes	-	✓
8	1.5	-	Yes	-	✓
9	2.0	-	Yes	-	✓
10	-	5 g	Yes	Yes	✓
11	-	10 g	Yes	-	L-a
12	0.7	-	Yes	-	✓
13	0.9	-	Yes	-	✓
14	-	-	NO	-	✓

'*L*' classification is pretty good apart from confusion with '*A*'. Below you can see the two signs in their original form. Comparing images in their raw form with their vectors representations is not very safe. Roughly though we may be able to see where the problem starts from. In this case the fist and finger arrangements are quite similar apart from the index finger. Fingers are critical because they generate many edges.

6- Conclusion

The idea of the project got started from a *McConnel's* idea of orientation histograms. Many researchers found the idea interesting and tried to use it in various applications. From hand recognition to cat recognition and geographical statistics. My supervisor and I had the idea of trying to use this technique in conjunction with Neural Networks. In other approaches of pattern recognition that orientation histograms have

been used different ways of comparing and classifying were employed. Euclidean distance is a straight forward approach to it. It is efficient as long as the data sets are small and not further improvement is expected.

Another advantage of using neural networks is that you can draw conclusions from the network output. If a vector is not classified correct we can check its output and work out a solution.

As far as the orientation algorithm is concerned it can be further improved. The main problem is how good differentiation one can achieve. This of course is dependent upon the images but it comes down to the algorithm as well. Edge detection techniques are keep changing while line detection can solve some problems. One of the ideas that I had lately is the one of tangents but I don't know if it is feasible and there is not time of developing it.

To say that I have come to robust conclusions at the end of the project is not safe. This is possible only for the first part of the project. Regardless of how many times you run the program the output vector will always be the same. This is not the case with the perceptron. Apart from not being 100% stable there are so many parameters (e.g. number of layers, number of nodes) that one can play with that finding the optimal settings is not that straight forward. As mentioned earlier it all comes down to the application. If there is a specific noise target for example you can work to fit this specifications.

My major goal was speed and the avoidance of special hardware. This was achieved although it would be faster if written in C / C++ but the complexity of the design and implementation would have been much higher. MATLAB is slower but allows its users to work faster and concentrate on the results and not on the design.

References:
1. Christopher M. Bishop, "Neural networks for Pattern Recognition" Oxford, 1995.
2. William T. Freeman, Michael Roth, "Orientation Histograms for Hand Gesture Recognition" *IEEE Intl. Wkshp. On Automatic Face and Gesture Recognition, Zurich, June, 1995.*
3. Maria Petrou, Panagiota Bosdogianni, "Image Processing, The Fundamentals", Wiley_ Vladimir I. Pavlovic, Rajeev Sharma, Thomas S Huang, " Visual Interpretation of Hand Gestures for Human-Computer Interaction : A review" *IEEE Transactions of pattern analysis and machine intelligence, Vol 19, NO 7, July 1997*
4. Srinivas Gutta, Ibraham F. Imam, Harry Wechsler, " Hand gesture Recognition Using Ensembles of Radial Basis Functions (RBF) Networks and Decision Trees" *International*

Journal of Pattern Recognition and Artificial Intelligence, Vol 11No.6 1997.

5. Simon Haykin, "Neural Networks, A comprehensive Foundation", Prentice Hall_ Duane Hanselman, Bruce Littlefield, "Mastering MATLAB, A comprehensivetutorial and reference", Prentice Hall

APPENDIX
Part1 : Image Processing Program

```
%%%%Klimis Symeonidis - Msc signal Processing
    Communications - Surrey
Univertsity%%%%%
%%%Orientation Histograms%%%%%
clc
% Select from menu Test or Train Sets
F = MENU('Choose a database set','Test Set','Train
    Set');
if F==1
%%% Select Test Set
K = MENU('Choose a file','Test A','Test V','Test
    W','Test 0','Test
1','Test From other DBs');
if K == 1
loop=4
for i=1:loop
string = ['test\a\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif K == 2
loop=7
for i=1:loop
string = ['test\v\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif K == 3
loop=5
for i=1:loop
string = ['Test\W\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif K == 4
loop=5
for i=1:loop
string = ['test\0\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif K == 5
loop=4
for i=1:loop
string = ['test\1\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif K == 6
loop=13
for i=1:loop
string = ['test\otherdb\' num2str(i) '.tif'];
2
Rimages{i} = imread(string);
end
end
end;
%%% Select Train Set
if F==2
loop=3 %Set loop to 3. All train sets have 3 images
L = MENU('Choose a file','Train A','Train V','Train
    W','Train
0','Train 1');
if L == 1
for i=1:loop
string = ['train\a\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif L == 2
for i=1:loop
string = ['train\v\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif L == 3
for i=1:loop
string = ['train\W\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif L == 4
for i=1:loop
string = ['train\0\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
elseif L == 5
for i=1:loop
string = ['train\1\' num2str(i) '.tif'];
Rimages{i} = imread(string);
end
end
end
% Resize all images to 150x140
T{i}=imresize(Timages{i},[150,140]);
x = [0 -1 1]; %x-derivative filter
y = [0 1 -1]'; %y-derivative filter
% returns only those parts of the convolution that can
    be computed
without assuming that the array A is zero-padded
dx{i} = convn(T{i},x,'same');
% returns only those parts of the convolution that can
    be computed
3
without assuming that the array A is zero-padded
dy{i} = convn(T{i},y,'same');
% divide the two matrices element by element to find
    gradient
orientation
gradient{i} = dy{i} ./dx{i};
```

```
theta{i} = atan(gradient{i}); %find the atan of
    'gradient'
% turn the matrix into a column vector
cl{i}= im2col(theta{i},[1 1],'distinct');
% convert radians to degrees
N{i} = (cl{i}*180)/3.14159265359;
% read and store the orientation magnitude every 10
    degrees
c1{i}=(N{i}>0)&(N{i}<10);
s1{i}=sum(c1{i});
c2{i}=(N{i}>10.0001)&(N{i}<20);
s2{i}=sum(c2{i});
c3{i}=(N{i}>20.0001)&(N{i}<30);
sum(c3{i});
s3{i}=sum(c3{i});
c4{i}=(N{i}>30.0001)&(N{i}<40);
sum(c4{i});
s4{i}=sum(c4{i});
c5{i}=(N{i}>40.0001)&(N{i}<50);
sum(c5{i});
s5{i}=sum(c5{i});
c6{i}=(N{i}>50.0001)&(N{i}<60);
sum(c6{i});
s6{i}=sum(c6{i});
c7{i}=(N{i}>60.0001)&(N{i}<70);
sum(c7{i});
s7{i}=sum(c7{i});
c8{i}=(N{i}>70.0001)&(N{i}<80);
sum(c8{i});
s8{i}=sum(c8{i});
c9{i}=(N{i}>80.0001)&(N{i}<90);
sum(c9{i});
s9{i}=sum(c9{i});
c10{i}=(N{i}>90.0001)&(N{i}<100);
sum(c10{i});
s10{i}=sum(c10{i});
c11{i}=(N{i}>-89.9)&(N{i}<-80);
sum(c11{i});
s11{i}=sum(c11{i});
4
c12{i}=(N{i}>-80.0001)&(N{i}<-70);
sum(c12{i});
s12{i}=sum(c12{i});
c13{i}=(N{i}>-70.0001)&(N{i}<-60);
sum(c13{i});
s13{i}=sum(c13{i});
c14{i}=(N{i}>-60.0001)&(N{i}<-50);
sum(c14{i});
s14{i}=sum(c14{i});
c15{i}=(N{i}>-50.0001)&(N{i}<-40);
sum(c15{i});
s15{i}=sum(c15{i});
c16{i}=(N{i}>-40.0001)&(N{i}<-30);
sum(c16{i});
s16{i}=sum(c16{i});
c17{i}=(N{i}>-30.0001)&(N{i}<-20);
sum(c17{i});
s17{i}=sum(c17{i});
c18{i}=(N{i}>-20.0001)&(N{i}<-10);
sum(c18{i});
s18{i}=sum(c18{i});
c19{i}=(N{i}>-10.0001)&(N{i}<-0.0001);
sum(c19{i});
s19{i}=sum(c19{i});
D{i}= [s1{i} s2{i} s3{i} s4{i} s5{i} s6{i} s7{i} s8{i}
    s9{i} s10{i}
s11{i}  s12{i}  s13{i}  s14{i}  s15{i}  s16{i}  s17{i}
    s18{i} s19{i}];
end;
end;
end;
%close the waiting bar
close(w);
5
```

Part2 : Neural Network Program

```
%%%%Klimis Symeonidis - Msc signal Processing
    Communications - Surrey
Univertsity%%%%%
%%%Perceptron   network   for   hand   gesture
    classification%%%%%
% Turn on echoing of commands inside the script-file.
echo on
% Clear the workspace (all variables).
% clear all
% load perf24
% Clear command window.
clc
%
    ===================================
    ======================
%
    ===================================
    ======================
%   CLASSIFICATION   WITH   A   2-LAYER
    PERCEPTRON:
% The first layer acts as a non-linear preprocessor for
% the second layer. The second layer is trained as
    usual.
pause % Strike any key to continue...
clc
% DEFINING THE CLASSIFICATION PROBLEM
% ===================================
% A matrix P defines 24 19-element input (column)
    vectors:
% There are 3 examples of % each character, 8
    characters, so 3 x 8
= 24 input
% patterns.
% A matrix T defines the categories with target
    (column)
% vectors. There are 3 numerals and 5 characters so, 8
    target
```

vectors in total.
pause % Strike any key to continue...
clc
% Open the files with the input vectors.
fid = fopen('train8.txt','rt');
P1 = fscanf(fid,'%f',[19,inf]);
P=P1;
fid = fopen('testA.txt','rt');
TS1 = fscanf(fid,'%f',[19,inf]);
fid = fopen('test0.txt','rt');
TS2 = fscanf(fid,'%f',[19,inf]);
fid = fopen('test5.txt','rt');
TS3 = fscanf(fid,'%f',[19,inf]);
fid = fopen('testL.txt','rt');
TS4 = fscanf(fid,'%f',[19,inf]);
fid = fopen('testV.txt','rt');
TS5 = fscanf(fid,'%f',[19,inf]);
fid = fopen('testW.txt','rt');
TS6 = fscanf(fid,'%f',[19,inf]);
6
fid = fopen('testH.txt','rt');
TS7 = fscanf(fid,'%f',[19,inf]);
fid = fopen('test1.txt','rt');
TS8 = fscanf(fid,'%f',[19,inf]);
fid = fopen('testGB.txt','rt');
TS9 = fscanf(fid,'%f',[19,inf]);
% Open the file with the target vectors.
fid = fopen('target8.txt','rt');
T = fscanf(fid,'%f',[8,inf]);
%clc
% DEFINE THE PERCEPTRON
% =======================
% P has 19 elements in each column,
% so each neuron in the hidden layer
% needs 19 inputs.
%R1;
% To maximize the chance that the preprocessing layer
% finds a linearly separable representation for the
% input vectors, it needs a lot of neurons.
% After trying a lot of different network architectures,
% it has been found that the optimal number of neurons for
% the hidden layer is 85.
S1 = 85;
% T has 5 elements in each column,
% so 5 neurons are needed.
S2 = 5;
% INITP generates initial weights
% and biases for the network:
% Initialize pre-processing layer.
[W1,b1] = initp(P,S1);
% Initialize learning layer.
[W2,b2] = initp(S1,T);
pause % Strike any key to train the perceptron...
clc
% TRAINING THE PERCEPTRON

% =======================
% TRAINP trains perceptrons to classify input vectors.
% The first layer is used to preprocess the input vectors:
7
A1 = simup(P,W1,b1);
% TRAINP is then used to train the second layer to
% classify the preprocessed input vectors A1.
% The TP parameter is needed by the TRAINP function
% to define the number of epochs used.
% The first argument is the display frequency and
% the second is the maximum number of epochs.
TP = [1 500];
pause % Strike any key to start the training...
%Delete everything and also reset all figure properties,
%except position, to their default values.
clf reset
%Open a new Figure (graph window), and return
%the handle to the current figure.
figure(gcf)
%Set figure size.
setfsize(600,300);
% Training begins...
[W2,b2,epochs,errors] = trainp(W2,b2,A1,T,TP);
% ...and finishes.
pause % Strike any key to see a plot of errors...
clc
% PLOTTING THE ERROR CURVE
% =======================
% Here the errors are plotted with
% respect to training epochs:
ploterr(errors);
% If the hidden (first) layer preprocessed the original
% non-linearly separable input vectors into new linearly
% separable vectors, then the perceptron will have 0 errors.
% If the error never reached 0, it means a new
% preprocessing layer should be created
% (perhaps with more neurons).
pause % Strike any key to use the classifier...
clc
% USING THE CLASSIFIER
% ===================
% IF the classifier WORKED we can now try to classify
8
% the input vectors we like using SIMUP. Lets try the
% input vectors that we have used for training.
% Create a menu, so the user can select a test set.
K = MENU('Choose a file resolution','Test A','Test 0','Test 5','Test
L','Test V','Test W','Test H','Test 1','Test GB');
% According to the choice use the appropriate variables.

```
if K == 1
TS = TS1;
elseif K == 2
TS = TS2;
elseif K == 3
TS = TS3;
elseif K == 4
TS = TS4;
elseif K == 5
TS = TS5;
elseif K == 6
TS = TS6;
elseif K == 7
TS = TS7;
elseif K == 8
TS = TS8;
elseif K == 9
TS = TS9;
else
P = 0;
R1 = 0;
end
a1 = simup(TS,W1,b1); % Preprocess the vector
a2 = simup(a1,W2,b2) % Classify the vector
echo offdisp('End of Hand Gesture Recognition')
```

9/17/2015