Chapter 5

Neural Networks in Face Localization

5.1 Introduction

In computer vision applications based on color-segmentation there is a common problem: the sensitivity to changes in the intensity of light. This is a result of the inability of some algorithms to tolerate variations presented in the color hue which correspond, in fact, to the same object.

Jepson, McKenna and Raja [215]-[217] have already discussed color segmentation using Gaussian distributions and mixture models. Isard [218] have employed color information in particle filtering. Recently, Perez [219] introduced an approach that also uses color histograms and a particle filtering framework for multiple object tracking. The two independently proposed methods differ in the initialization of the tracker, the model update, the region shape and the observation of the tracking performance. Bradski [220] modified the mean-shift algorithm (Camshift) which operates on probability distributions to track colored objects in video frame sequences. The color image data has to be represented as a probability distribution; normally it is used color histograms to accomplish this.

Learning Vector Quantization (LVQ) networks learn to recognize groups of similar input vectors in such a way that neurons physically near to each other in the neuron layer respond to similar input vectors. The learning is supervised, the inputs vectors into target classes are chosen by the user. In this thesis a new color-segmentation algorithm based on LVQ is presented. It involves neural networks that operate directly on the image pixels with a decision function.

In comparison, our LVQ algorithm has the advantage of working only with image pixels, without using any dynamic model or probability distribution, that which means processing speed and implementation easiness. LVQ algorithm has been applied to spotting and tracking human faces, and shows more

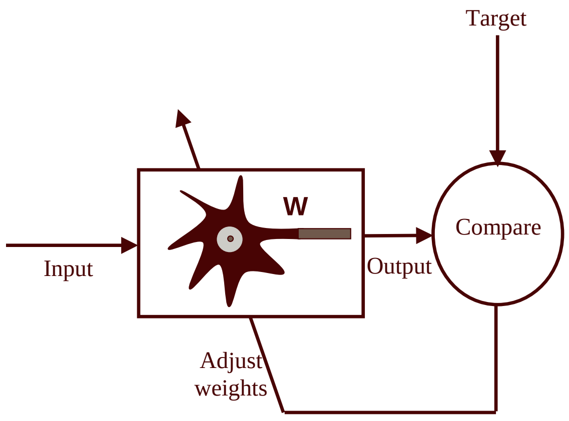


Figure 5.1: Supervised learning in Neural Networks

robustness than other standard algorithms as Camshift for the same task.

5.2 Neural Networks

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological systems. As in nature, the network function is determined by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (**W** 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 the figure 5.1. 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 learning*, to train a network.

Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as on line or adaptive training.

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

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.

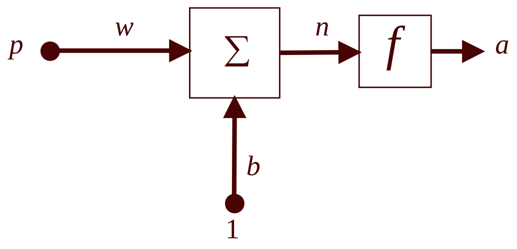


Figure 5.2: Simple neuron model

5.2.1 Simple neuron model

In a neuron 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 has too a scalar bias, b. We 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, a representation of this model is in the figure 5.2.

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, typically a *step function* or a *sigmoid function* (although exist other), which takes the argument n and produces the output a.

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.

5.2.2 Network Architecture.

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers.

A one-layer network with R input elements and S neurons is shown in the figure 5.3. In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith neuron has a summer that gathers its weighted inputs and bias to form its own scalar output n_i . The various n_i taken together form an S-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. We show the expression for a at the bottom of the figure. Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., R and S). A layer is not constrained to have the number of its inputs equal to the number of its

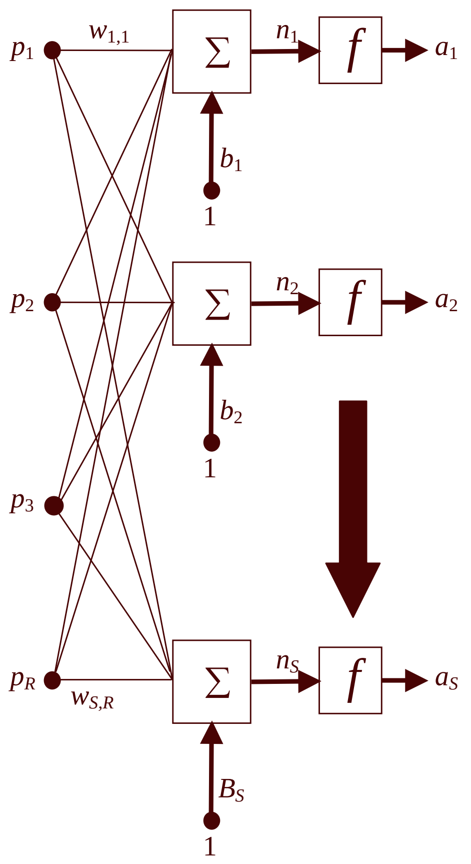


Figure 5.3: A one-layer network with R input elements and S neurons.

neurons.

The input vector elements enter the network through the weight matrix **W**. Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in say that the strength of the signal *from* the second input element *to* the first (and only) neuron is $w_{1,2}$.

The S neuron R input one-layer network also can be drawn in abbreviated notation as is shown in the figure 5.4. Here **p** is an R length input vector, **W** is an S x R matrix, and **a** and **b** are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function boxes.

A network can have several layers. Each layer has a weight matrix **W** (although in occasions $\mathbf{W}^1$is called **IW** because is in contact with the inputs of the net), a bias vector **b**, and an output vector **a**. To distinguish between the weight

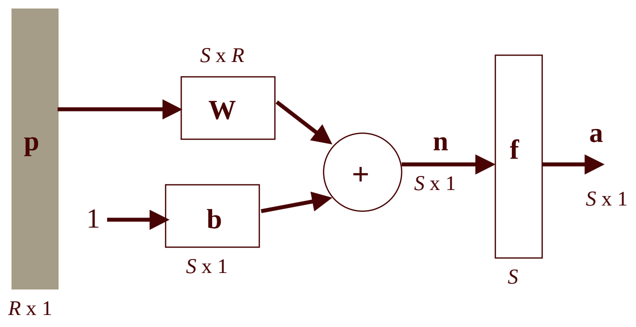


Figure 5.4: Abbreviated notation of one-layer network.

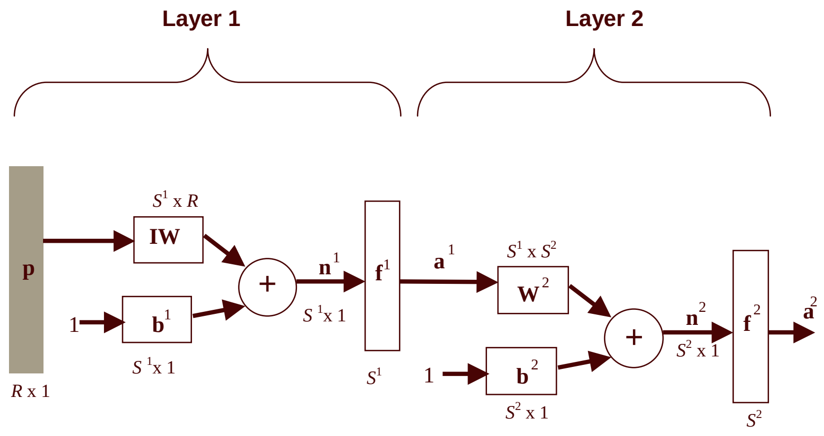


Figure 5.5: Two-layer network.

matrices, output vectors, etc., for each of these layers in the figures, we append the number of the layer as a superscript to the variable of interest. We can see the use of this layer notation in the two-layer network shown in the figure 5.5.

5.2.3 Learnig Algorithm.

We define a *learning rule* as a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules fall into two broad categories: supervised learning, and unsupervised learning.

In *supervised learning*, the learning rule is provided with a set of examples (the *training se*t) of proper network behavior where is an input to the network, and is the corresponding correct (*target*) output. As the inputs are applied to

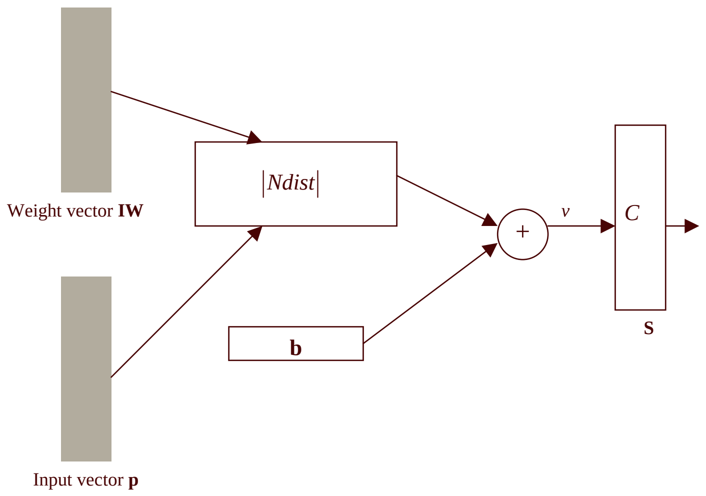


Figure 5.6: Architecture of a competitive network

the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets.

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization.

5.3 Competitive Networks

Competitive Networks [77] learn to classify input vectors according to how they are grouped in the input space. They differ from another networks in that neighboring neurons learn to recognize neighboring sections of the input space. Thus, competitive layers learn both the distributions and topology of the input vectors which they are trained on.

The architecture for a competitive network is shown in Fig. 5.6. The $|Ndist|$ box in the figure accepts the input vector **p** and the input weight matrix **IW** and produces a vector having **S** elements. The elements are the negative of the distances between the input vector **p** and the vector **IW**. The net value v of the competitive layer is computed by finding the negative distance between input vector **p** and the weight vector **IW** and then adding the biases **b**. If, all biases are zero, the maximum net input that a neuron can have is 0. This occurs when the input vector **p** equals the neuron's weight vector.

The competitive transfer function C accepts a net value v and returns neu-

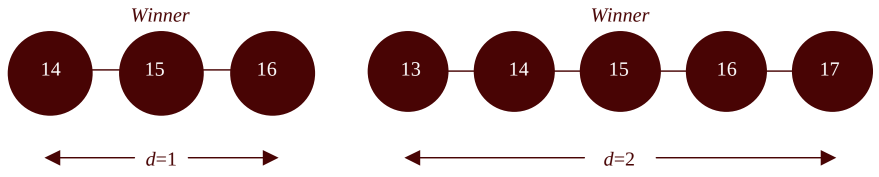


Figure 5.7: $Left$, one dimensional neighborhood of radius $d = 1$. $Right$, neighborhood of radius $d = 2$.

rons outputs of 0 for all neurons except for the $winner$, the neuron associated with the most positive element of the input v. Thus, the winner's output is 1.

The weights of the winning neuron are adjusted with the Kohonen learning rule. Supposing that the i^{th} neuron wins, the elements of the i^{th} row of the input weight matrix and all neurons within a certain neighborhood $Ni(d)$ of the winning neuron are adjusted as shown the Eq.5.1.

$$_i\mathbf{IW}^{1,1}(q) =_i \mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) -_i \mathbf{IW}^{1,1}(q-1)) \tag{5.1}$$

Here α is the learning rate and $Ni(d)$ contains the indices for all of the neurons that lie within a radius d of the i^{th} winning neuron. Thus, when a vector **p** is presented, the weights of the winning neuron and its closest neighbors move toward **p**. Consequently, after many presentations, neighboring neurons will have learned vectors similar to each others. The winning neuron's weights are altered proportional to the learning rate. The weights of neurons in its neighborhood are altered proportional to half the learning rate. In this thesis, the learning rate and the neighborhood distance (used to determine which neurons are in the winning neuron's neighborhood) are not altered during training.

To illustrate the concept of neighborhoods, consider the Fig. 5.7. At left is shown a one dimensional neighborhood of radius $d = 1$ around neuron 15, at right is shown a neighborhood of radius $d = 2$.

These neighborhoods could be written as:

$$N_{15}(1) = (14, 15, 16), N_{15}(2) = (13, 14, 15, 16, 17)$$

5.4 Learning Vector Quantization Networks

An LVQ network [78] has first, a competitive layer and second, a linear layer. The competitive layer learns to classify input vectors like the networks of the last section. The linear layer transforms the competitive layer's classes into target classifications defined by the user. We refer to the classes learned by the competitive layer as $subclasses$ and the classes of the linear layer as $target$ $classes$. Both the competitive and linear layers have one neuron per class.

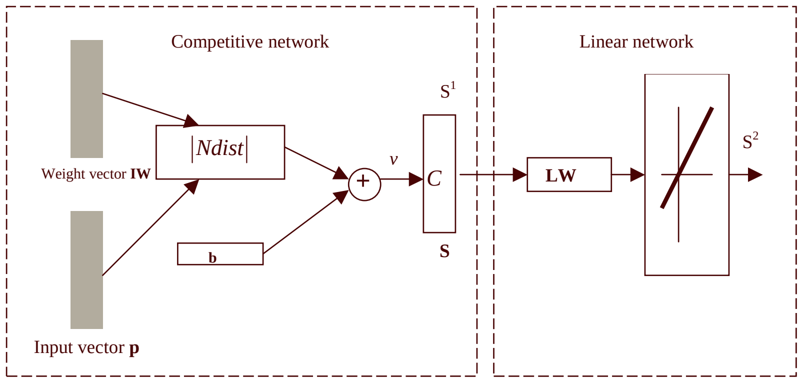


Figure 5.8: Schematic representation of the LVQ net

Thus, the competitive layer can learn $\mathbf{S}^1$ classes. These, in turn, are combined by the linear layer to form $\mathbf{S}^2$ target classes. The LVQ network architecture is shown in the Fig. 5.8.

Summarizing, the LVQ network allows to order classes learned by the competitive network in a final vector more appropriate for the color-segmentation.

5.5 Architecture of the color segmentation System

The core of the algorithm is a LVQ network whose inputs are connected directly with the pixel-vector of the image $\mathbf{I}$ (in RGB format) and its outputs are connected directly to the decision function $\mathbf{f_d}$, which produces an output of 1 or 0 depending of if the color corresponds to the object to be segmented forming in this way a new vector image $\mathbf{I'}$. The Fig. 5.9 shows a scheme of the segmentator.

Considering that the LVQ net is configured with N output neurons, then it would be possible train the competitive net to learn the configuration space and the color pixels topology, described as the vector $\mathbf{p}$ with elements $\mathbf{p}_R$, $\mathbf{p}_G$ and $\mathbf{p}_B$ coming from the image. For the supervised training of the linear net, we suppose that the image $\mathbf{I}$ contains an Object O to be segmented, being $\mathbf{p_O}$ a pixel corresponding to the object, we train the linear network in such a way that the class $\mathbf{S}^2$ (of this pixel) corresponds to the assigned (in a supervised way) for the neuron $N/2$ of the linear network.

The idea is that the color to be segmented is located halfway of the network, while the similar colors are located in the neighboring neurons. In such a way that if exists a vector $\mathbf{p_1}$ that belongs to the object but for the illumination conditions and noise could be classified in the neighborhood of the neuron N/2.

The classification achieved by the LVQ network gives a vector of elements categorized by $\mathbf{S}^2$ of N elements corresponding to the N classes. Each element

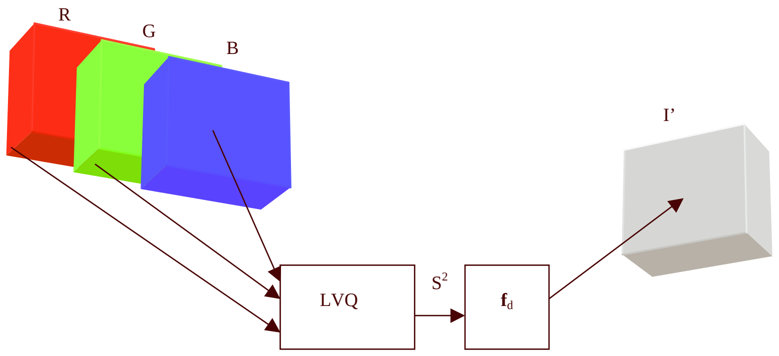


Figure 5.9: Architecture of the color segmentation system

of the $\mathbf{S}^2$ vector could have 2 possible values, 1 or 0 and only an element from each vector could be 1, while the other elements will be 0. Then for the object to be segmented, the activation of the neurons is concentrated in the middle of the vector, that is the neurons nearest to the $N/2$ will have a bigger possibility to be activated than the other ones corresponding to the color to be segment.

Considering the previous problem is necessary to define a function $\mathbf{f_d}$ able to define neuron´s density which will be taken to consider if a pixel corresponds or not to the color to be segmented, this function will be called in this thesis decision function. It is possible to formulate many functions which could solve the decision problem satisfactorily, however the Gaussian function has been chosen by its well-known properties, although it is possible to use other, including non-simmetrical distributions. The Fig. 5.10 shows graphically the function used.

The equation 5.2 shows mathematically this function where $\mathbf{g}$ is the number of the activated neuron, μ is $N/2$ and σ is the standard deviation. Therefore, $\mathbf{f_d}$ has only a calibration parameter represented by σ which determines the generalization capacity of the complete system. Thus, if the value of σ is chosen small enough, the segmentation capacity will be more selective than in the case of a bigger σ. For the final result the decision function was evaluated with a threshold of 0.7.

$$\mathbf{f_d}(\mathbf{g}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(- \frac{(\mathbf{g}-\mu)^2}{2\sigma^2} \right) \tag{5.2}$$

5.6 Implementation

The implementation is divided in two parts, the net training and the segmentator structure.

In the first part, a net of 30 neurons is chosen for the competitive layer and

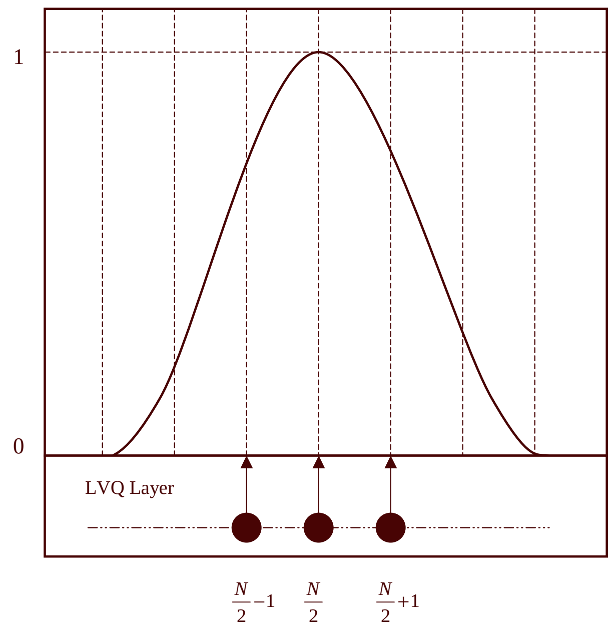


Figure 5.10: Decision function model

also for the linear layer, then a frame of the image is taken containing the object whose color will be segmented, then a pixel corresponding to the color of this object is chosen and the training of the LVQ net is made it, allowing that the pixel-class corresponds to the 15^{th} net class ($N/2$). During the training it is used a learning rate α of 0.1 and a distance radius $Ni(d)$ of 3. This conditions make sure that the winner weights are affected in a reason of 0.1 while three neurons of its neighbors (in both ways) are affected in a reason of 0.05. The process of training was made using the neural networks toolbox of Matlab.

In the second part(the segmentator implementation), a decision function with parameters μ =15 and σ =3 was added to the net (above) generated by the training. The complete system was coded in C++ and tested on a PCx86 at 900 MHz with 128 MBytes RAM, operating in real time on an image of 352x288 pixels surrendered by an USB-Webcam.

To inspect the segmentator robustness, it was tested on face localization. The segmentator gives an image with a value of 1 in those points that belong to the segmented object while in other cases this is 0. This is caused as a result that the object to be segmented in this case, will have an appearance of a white fleck. Thus, to find the object position it will be necessary to calculate the object centroid (5) through the zero moment (3) and first degree moment order (4).

$$\mathbf{M_{00}} = \sum_x \sum_y I(x,y), \tag{5.3}$$

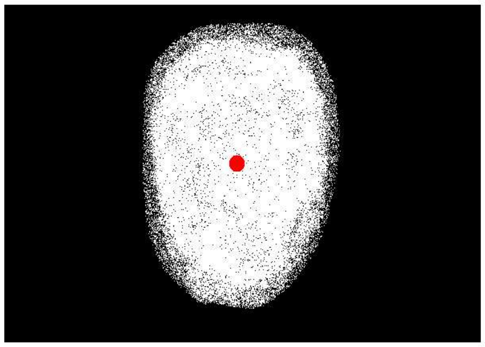


Figure 5.11: Center coordinates $\mathbf{x_c}$ and $\mathbf{y_c}$.

$$\mathbf{M_{10}} = \sum_x \sum_y xI(x,y), \mathbf{M_{01}} = \sum_x \sum_y yI(x,y), \tag{5.4}$$

$$x_c = \frac{\mathbf{M_{10}}}{\mathbf{M_{00}}}, y_c = \frac{\mathbf{M_{01}}}{\mathbf{M_{00}}}, \tag{5.5}$$

where $\mathbf{M_{00}}$ represents the zero degree moment while of $\mathbf{M_{10}}$ and $\mathbf{M_{01}}$ means the first degree moments of $\mathbf{x}$ and $\mathbf{y}$ respectively, while $\mathbf{x_c}$ and $\mathbf{y_c}$ represent the center coordinates. The Fig. 5.11 shows this process.

5.7 Neuronal Network Creation and Training in Matlab

5.7.1 Creating a NN in Matlab

Competitive layers and self organizing maps can be created with the Matlab commands *newc* and *newsom*, respectively. Also is possible specify different topologies for the original neuron locations with the Matlab functions *gridtop*, *hextop* or *randtop*.

In the neural networks toolbox from Matlab, there are four distinct ways to calculate distances from a particular neuron to its neighbors. Each calculation method is implemented with a special function.

In this thesis was used the function *dist*. It calculates the Euclidian distance from the home neuron to any other neuron.

Suppose that we want to create a network having input vectors with two elements that fall in the range 0 to 2 and 0 to 1 respectively. Further suppose that we want to have 10 neurons in a linear manner 1-by-10 network. The code to obtain this network is:

```
net=newsom([0 2; 0 1],[1,10]);
```

5.7.2 Training a NN in Matlab

Learning, in a self-organizing map, occurs for one vector at the time, independent of whether the network is trained directly (with the Matlab function $trainr$) or wether it is trained adaptively (with the Matlab function $trans$). In either case, $learnsom$ is the self-organizing map, weight learning function in Matlab.

First, the neuron identifies the winning neuron, the weights of the winning neuron and then the neighboring neurons are moved closer to the input vector at each learning step using the self-organizing learning map (Matlab function $learnsom$). The parameters can to be modified with the values

```
LP.order_lr.
LP.order_steps.
LP.tune_nd.
```

Learning rate, Nummer of steps and neighborhood distance, respectively.

Thus, we can train the network for 100 epochs, a learning rate of 0.5 and neighborhood distance of 1. The code is:

```
net.trainParam.epochs=100;
net.trainParam.order_lr=0.05;
net.trainParam.tune_nd=1;
net=train(net,P);
```

5.7.3 Creating and training the Linear Layer

Linear layers can be created with the Matlab command $newlin$. For example, we can create a network with two elements that be in the range -1 to 1 and one output, like is show in the followings commands:

```
net=newlin([-1 1; -1 1], 1);
```

Linear networks can be trained to perform linear classification with the Matlab function $train$. This function applies to each vector a set of input vectors and calculates the network weights and bias increments due each of the inputs, according to $learnp$ Matlab function.

5.7.4 Auxiliary Functions

For the image capture the VFM utility was used, this Matlab-tool captures images(in RGB format) in a $m*n*3$ matrix(been mxn the dimensions of the image and the index 3 the corresponding index for R, G and B). This data matrix is represented in a integer format and it should be converted to a float point matrix(**double**). An example code for an image capture is shown.

```
vfm Im=vfm ('grab',1);
```

the captured image will be store in **Im**. **Im** should be transform for the training in a data vector of mxn type (R, G and B color channels) thats is performance by the next commands:

```
R=Im(:,:,1);
V=Im(:,:,2);
A=Im(:,:,3);
[m,n]=size(R);
index=0;
for a1=1:m
for an=1:n
data=R(a1,an);
data1=V(a1,an);
data2=A(a1,an);
index=index+1;
VR(index)=data;
VV(index)=data1;
VA(index)=data2;
end
end
VP=[VR;VV;VA];
VP=double(VP);
```

Thus, from **VP** (the training vector) is possible to perform the training. The following functions sequens make that:

```
net=newsom([0 255;0 255;0 255],30);
net.trainParam.epochs=500;
net.trainParam.order_lr=0.1;
net.trainParam.tune_nd=3;
net=train(net,VP);
```

Ones, that the competitive neuronal network has been trained than it is trivial to find a point and their correct identification in the data vector and then make able the supervised training of the lineal network(using the Matlab function $learnp$). During the training, practically any parameters combination give good results.

It is possible that the data obtained by the neuronal network could be interpreted as an image. To prevent this, a function was created, this function allows to modify the vector format to a mxn image format, the following commands show that:

```
index=0;
for a1=1:m
for an=1:n
index=index+1;
data=Image(index);
MR(a1,an)=data;
end
end
```

where Image is the data vector obtained by the neural network.

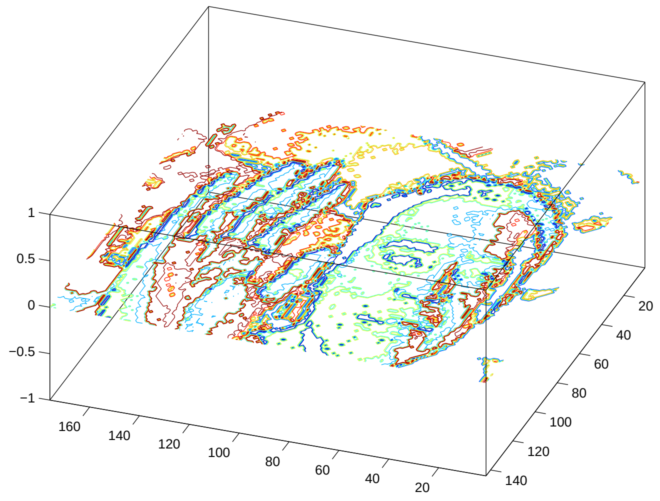
5.8 Conclusions.

In this section the segmentator results obtained in 3 different parts are presented.

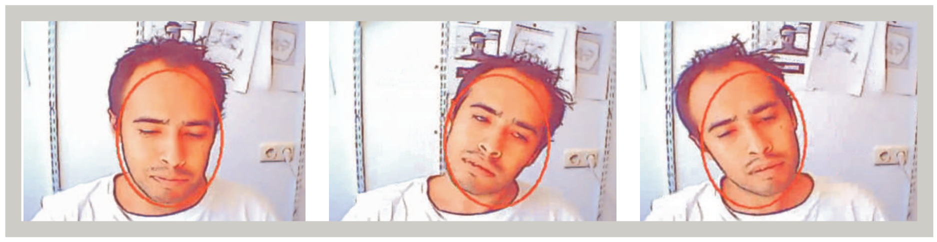
- In the learning of the LVQ network was proved the net classification robustness and its capacity to organize topologically, which shows excellent results, being able to perform the pixel classification of the image even in critical cases with very few neurons. The Fig. 5.12(a) shows the obtained classification results from an image, using a LVQ network with only 10 neurons.

- The system was tested to track a human head showing very good results. The system performance proposed in this work compared with some other algorithms [75,76] was good with the advantage that is computationally more efficient, being better for applications where it is required, besides of the object localization, also to perform complex calculations, for example the dynamics of a robot structure (that could supports the vision system). The Fig. 5.12 (b). shows a tracked secuence in the final application.

- The influence of the σ parameter in the decision function was demonstrate, and for this case values of σ in the interval of 3 at 5 gave good results. It can also be said that the robustness of the system can improve if it were possible to implement an algorithm to adapt the σ parameter. In the Fig. 5.13 (a)-(b) shows the results obtained with different values of σ.



(a)



(b)

Figure 5.12: (a) Classification performed by a LVQ network with 10 neurons. (b) Final secuence of the segmentator used for face tracking

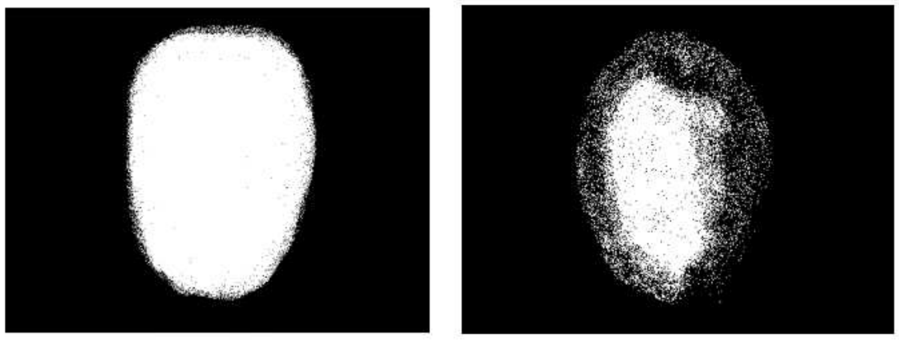


Figure 5.13: Results considering (a) $\sigma = 4$ and (b) $\sigma = 1$.

- The robustness to the illumination changes of the LVQ algorithm was compared with the standard algorithm Camshift. For the comparison test, the algorithms were allowed to run until "track failure" to measure basic tracking performance. The “track loss” definitions used in the comparisons were proposed in [221]; a track is determined to be lost when either the estimate is sufficiently far from the true track (the "norm test") or the true measurement does falls outside the gate (the "acceptance test") for five consecutive time steps. The “track-lifetime” is defined as the time step at which the track was lost; the track lifetimes for the comparison were averaged across all trials. The figure 5.14 shows the results obtained in the comparison. Both algorithms were tested considering 100 luxes as initial point (that is the average illumination of an office), in this point the average track lifetime is infinite for both. From that initial point value measurements were made in both directions, showing that the LVQ algorithm hat a better robustness, particularly in low illuminations.

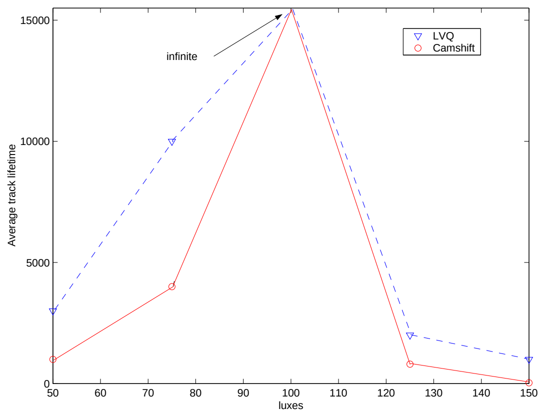


Figure 5.14: Results obtained in the comparison LVQ and Camshift.