Object Recognition: A Shape-Based Approach using Artificial Neural Networks

Jelmer de Vries
University of Utrecht
Department of Computer Science
0024260
jpvries@students.cs.uu.nl
Supervisor: Marco Wiering
marco@cs.uu.nl

Abstract

In this document we will introduce a novel artificial intelligence approach to object recognition. The approach is shape-based and works towards recognition under a broad range of circumstances, from varied lighting conditions to affine transformations. The main emphasis is on its neural elements that allow the system to learn to recognize objects about which it has no prior information. Both supervised techniques, in the form of feed-forward networks with error back-propagation, and unsupervised techniques, in the form of a self-organizing map, are employed to deal with the interpretation of objects in images.

To make the system complete, a full account is given of the necessary image processing techniques that are applied to the images to make recognition possible. These techniques include the extraction of shapes from images and the creation of descriptors that overcome difficulties of affine transformations.

The aim has been to create a complete object recognition system. This document gives an in-depth account about the choices made and the results the system has obtained. The objective of this work has been to create a new system for artificial object recognition that forms a good basis for further development.

Contents

Chapter 1

Introduction

Humans make object recognition look trivial. We can easily identify objects in our surroundings, regardless of their circumstances, whether they are upside down, different of color or texture, partly occluded, etc. Even objects that appear in many different forms, like vases, or objects that are subject to considerable shape deviations, such as trees, can easily be generalized by our brain to one kind of object.

Many different theories have been developed on how humans perform object recognition. Most of them are based on logical reasoning and on clear abstractions, and sound very plausible. Yet to go from human object recognition to computerized object recognition is a large step. The theories of human object recognition often do not touch on the lower levels of vision processing, i.e. they may assume that extracting an object from its surroundings is a given and that decomposition of this object into different entities is something that happens naturally. Sometimes some abstract principles are introduced that describe the workings, but the actual (biological) implementation from top to bottom is hardly ever touched upon. However, shapes we can easily extract from their surroundings, to a computer will be no more than a clutter of pixels. These can not always easily be separated from the random pixels surrounding it. A lot of processing can help to extract some information on the shapes and objects in a picture, but the level of abstraction humans achieve can not be matched by the image processing techniques of today.

In this thesis we will make an attempt to go all the way from the lowest layers of processing to allowing the computer to generalize different views of the same object into one category. The emphasis will be on the last part, the artificial intelligence (AI), as this is the focus of this work. However, as we will be creating an entire application that works from top to bottom we will discuss all techniques and give a full overview of the process. Of course the results are still far from perfect. However, we hope this document shows the possibilities of this approach and the lines along which further development could take place.

1.1 Objective

In this thesis we look at the difficult task of object recognition. Object recognition can be done employing a neural system that incorporates aspects of human object recognition, together with classical image processing techniques. We will first look at work that has already been done in the field of object recognition and AI. Then we will try a new approach to solving the problem of object recognition. Basically what we want to research is: *to what extent can a neural system combined with a shape-based approach be used to recognize objects efficiently and effectively?*

The main reason for our interest in object recognition stems from the belief that generalization is one of the most challenging, but also most valuable skills a computer can have. Since the area of vision probably depends on generalization more than any other area, this makes it a challenging endeavor.

Below we have listed the general characteristics of the research done for this thesis

- The ultimate goal of object recognition is to be able to recognize an object no matter what the circumstances (background, lighting, occlusion, etc.) As this is not trivial to achieve, certainly not without making any reservations, we will try a step by step approach, moving from simple shape recognition to more complex object recognition.
- As object recognition involves a lot more than just building a neural system other techniques are also discussed in this document. Since these other techniques are not always at the center of this research, these discussions will not always have the same depth.
- The created adaptive systems are tested as thoroughly as possible. Due to constraints of computational power and time sometimes tests have been less extensive in order to make way for testing more variations.
- The system will be tested using different kinds of tests. Simple tests are introduced to check parts of the system. In the end a final test with pictures of actual objects will show the workings of the complete system.
- All of the work described has been implemented and tested in Java. A brief explanation of the workings of the program can be found in appendix A.

1.2 Difficulties

The difficulties of object recognition are extensive. To avoid too general a discussion we will mainly look at them here in a light that makes sense when working with neural systems. Shapes can differ in appearance for several reasons. The most important reason is the difference in perspective we can have on a shape, i.e. shapes can be viewed from different angles and positions possibly making the shape appear bigger, upside down, tilted etc. But we also have to deal with shapes that, even though they might represent the same concept, actually are different. No two trees will ever look exactly the same, yet in the process of recognition we do want to label them using a single classifier. In this section we will give a quick overview of the most important difficulties we have to deal with when working with the visual aspects of object recognition.

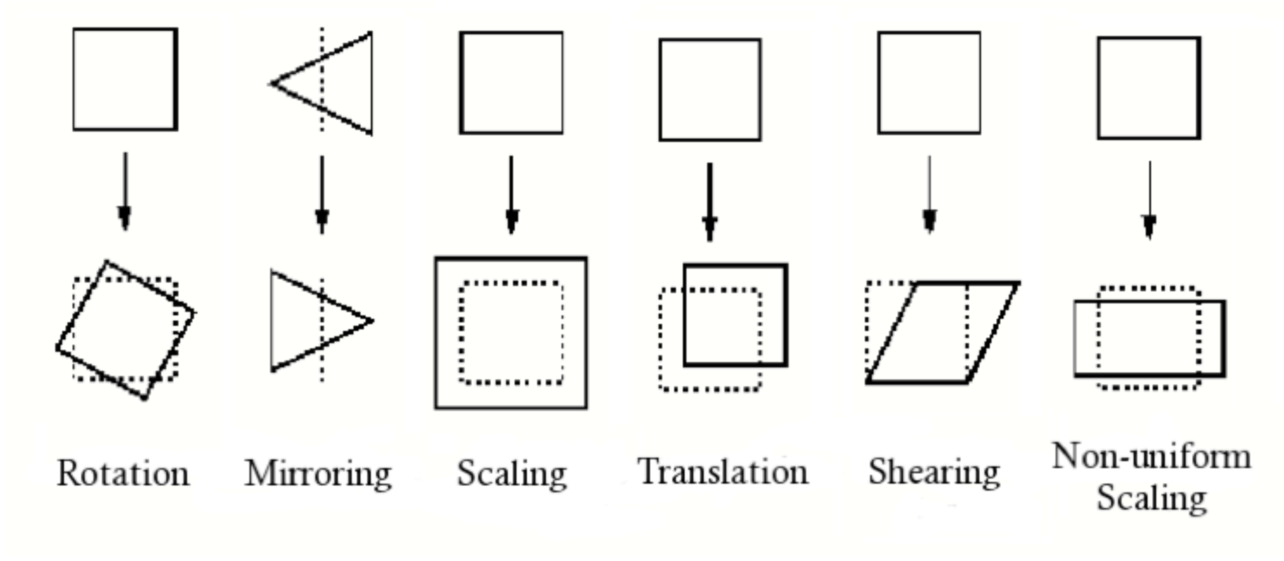


Figure 1.1: Examples of all the affine transformations for a simple geometric shape.

1.2.1 Rotation, Mirroring, Translation and Scaling

A basic problem in recognizing a shape is caused by the fact that it can be in any kind of position, rotated, mirrored, etc. Such transformations belong to the group of affine transformations, which includes any kind of transformation that preserves distance ratios as well as co-linearity. To a neural system these transformations can quickly lead to difficulties recognizing the object. Examples of these transformations can be seen in figure 1.1.

- *Rotation*

  Objects appearing under different angles have very different representations when it comes to their pixels. Without taking this into account in some way, a neural network will incorrectly recognize identical patterns as different ones, depending on rotation.

- *Mirroring*

  Similar to the possibility of an object appearing rotated, it can also be flipped, i.e. in a mirrored position. This creates a problem similar to that of rotation.

- *Scale invariance*

  Different scalings of a shape create another problem of the same class. Shapes can appear in different sizes for many reasons. It can, for instance, be due to the perspective or their actual size can differ. Even though humans have little problem dealing with these and the previous transformations they are far from trivial to a computer.

- *Translation*

  The problem of translation is the problem of an object appearing in a different place in the image. Although this is not one of the most difficult problems to overcome, it certainly should not be ignored when creating an object recognition system.

- *Shearing and non-uniform scaling*

  There are also more difficult affine transformations mainly shearing and non-uniform scaling (see figure 1.1). Small transformations can generally be handled by having good shape descriptors in combination with the generalizing power of the learning system. Larger transformations are harder to accommodate, since there comes a point where it is questionable whether we are still dealing with the same shape.

Even though small shifts along these lines do not directly lead to problems it is almost impossible for a basic neural network to accommodate all these effects. This would require the network to have seen each object, in every possible place, under every possible rotation, in every possible size, etc.

It can even be debated whether achieving perfect invariance on the earlier mentioned points is always a good thing. Perfect invariance on all these points would for instance also mean that b is equal to d (*mirroring*), and that d is equal to p (*rotation*). Furthermore, scaling invariance could cause us to ignore important information. For example, the size relative to other stimuli can be useful in the process of deciding in what category a white circle could fall, otherwise ranging anywhere from a golf-ball to a volleyball (*scaling*).

Of course a long discussion could be held on all these annoying little complications on the road to achieving invariance. As we will later see, however, even though important information will be lost in obtaining invariance, this information will not be discarded when we design our full object recognition system, where we will work with invariant shapes as well as mutual relations between shapes.

1.2.2 Occlusion

A difficulty that is harder to remedy than those mentioned up till now is that of occlusion. Here we are not dealing with information that is offered in a different way, but with information that is simply not there. It will in some cases even be supplemented by extra information, not dealt with before. Occlusion is difficult to remedy when dealing with the recognition of an object as a whole, i.e. a situation where the object is for instance offered directly to a neural network. Making a system more robust can be achieved by introducing partial recognitions. This can be done by segmenting the object in such a way that loose features can be computed, so that when part of the object is missing, the features that are fully intact can still serve as a basis for recognition.

1.2.3 Multiple objects

Just about the hardest problem of object recognition in real images is that the objects are not likely to appear alone, making it very hard to recognize them, since separation of objects in an image is not a trivial task. This is even harder when two objects are touching or overlapping, possibly letting them be identified as one. Such a problem, of separation of objects and clustering of various elements, by itself justifies a full research project and we have no other choice then to give this issue here a lower priority and refer the research to future work.

1.2.4 Lighting

A perennial problem in working with images is the various lighting conditions a picture can have been taken under. The effect of lighting on colors is extremely large, and especially in color based recognition is considered to be one of the most difficult problems. Human eyes within a broad range automatically adjust for brightness. A computer does not have that capability. This greatly reduces its ability to recognize colors. Good examples are the very popular soccer robots, the Aibos. In most cases they are programmed to find their location in the field by identifying colors of objects around them. Every new match to be played by these robots is preceded by a large operation of re-calibrating them to new lighting conditions. This takes a lot of precious time (it can easily take an hour) and can be troublesome because the time constraints often compromise the quality. Since we work with shapes instead of colors we are less dependent on these changing colors, but unfortunately this does not mean that lighting is not a problem for us. Mechanisms like edge extraction are sensitive to lighting conditions. Edges tend to disappear when lighting becomes dim and the difference in brightness of pixels tends to decrease. And brighter images can on their part lead to the extraction of too much edge information, as with a brighter image small color changes can become more visible.

1.3 Practical application

The main motivation for the research lies with the challenge of generalization and discovering what the limits and possibilities are. But apart from the theoretical point of view there also is great practical use when it comes to object recognition. The motivation for wanting to build a system that can learn to recognize objects mainly stems from a desire to make the computer able to interact with the real world. Many theories are written on the capabilities of robots, but as for now robots still have little notion of what is going on in the outside world. An object recognition system should (in the future) allow a robot to process the vision input around it and assign meaning to what it sees.

Currently this is still a far-fetched goal, but steps can be taken which also in the near future could already produce useful results.

1.3.1 Content based image retrieval

Due to the growing presence of media on the computer text-based searches are often no longer sufficient for finding what we need. An area that is now quickly gaining importance is Content Based Image Retrieval (CBIR). Its goal is to provide in the growing need to make media, and specifically images searchable. Most work in this field is done by indexing large databases by supplying images with a characteristic representation of its contents that is both compact and highly descriptive. Even though CBIR is not the same as object recognition, both processes require techniques for describing image content. In an attempt to create standards for this field MPEG-7 [MPE01] has been created. Further in this document we will return to this standard and its implication for us.

1.3.2 Security

With increasing calls for security at places such as airports, government buildings, public transportation and public institutions, computer recognition has become an invaluable tool. Even though most of its applications are now still directed to face recognition, there is certainly also potential for object recognition to improve security.

Two examples of how object recognition can be used to improve security are:

- Airport scanners. Checking of luggage still depends heavily on human observation. Lack of concentration can let dangerous objects slip past security. Having an automated system could be a useful tool for assisting people that have to do this work.

- Security in banks. Cameras detecting the presence of guns and other arms can help alarm staff about people carrying suspicious items. Techniques for doing this are still under development, but are expected to become operative in the near future.

1.3.3 Robotics

The current line of robots is not yet at a level where it is concerned much with reasoning about the specifics of its surroundings. However, as progress in this field continues, interaction with the surroundings will become an increasingly prominent part of research. For now, however, only simple object recognition is used in some cases. Robots participating in the popular robot soccer competitions (`http://www.robocup.org/`) only need to recognize landmarks and the ball to be able to determine their place on the field, as well as the best way to achieve their goals. Depending on the power of the robot's computer this recognition is done on the basis of different features. Robots in the Aibo-league, that have limited processing power, generally use color to determine what a goal, ball or landmark is. However, as mentioned in section 1.2.4 this is far from ideal. Larger soccer robots, carrying bigger computers, can employ more specific methods. But these robots generally do not have the same flexibility as the Aibos on the soccer-field. This illustrates the importance of having efficient mechanisms for recognition, which surely is something that can be achieved using neural networks.

1.4 Approach

We will now give a brief introduction of the system we are proposing in this document. The process starts with the decomposition of objects into entities that will then serve as the basis for our recognition system. Broadly speaking our system can be divided in three steps:

1. *Preprocessing*

   Our system first processes images with preprocessing steps to extract entities that can be used for recognition by a neural system. This is done because the system for recognition we are proposing mainly relies on the fact that an object can be decomposed into many shapes that can be extracted from an image. These shapes will then be the basis for our adaptive system.

2. *Descriptor creation*

   After having extracted the shapes the descriptors are created. This is another central issue that will be researched in this thesis. These descriptors are made specifically to be invariant representations entailing as much important information of the shape as possible. They are created in such a way that they can be fed into a neural network and allow for easy categorization.

3. *Learning and recognition*

   The adaptive system is a combination of both feed-forward neural networks and a self-organizing map (SOM). The training of the system is performed in several phases, both supervised and unsupervised. Broadly speaking our adaptive approach can still, as most adaptive systems, be divided in two phases. First, there is the learning phase, which is the phase where the system processes training data and adapts itself to their specifics. Second there is the recognition phase. Here the system's recognition capabilities are tested, when confronted with images which it has not encountered before.

The entire system from top to bottom is shown in figure 1.2

1.5 Overview

In the following chapter, chapter 2, some basic principles of human recognition and related work on artificial object recognition will be discussed. Chapter 3 describes all the preprocessing that is done, and explains how we extract shapes from an image. Chapter 4 discusses the creation of shape descriptors that can be used as inputs for a neural network, without having to worry about the transformations of the shape. The adaptive system is described in chapter 5 together with a full account of the techniques used to create the system. The results of the system are discussed in chapter 6 where we test all the aspects of the created program. Finally, in chapter 7, we discuss the results achieved and the kind of additions that can be made in the future.

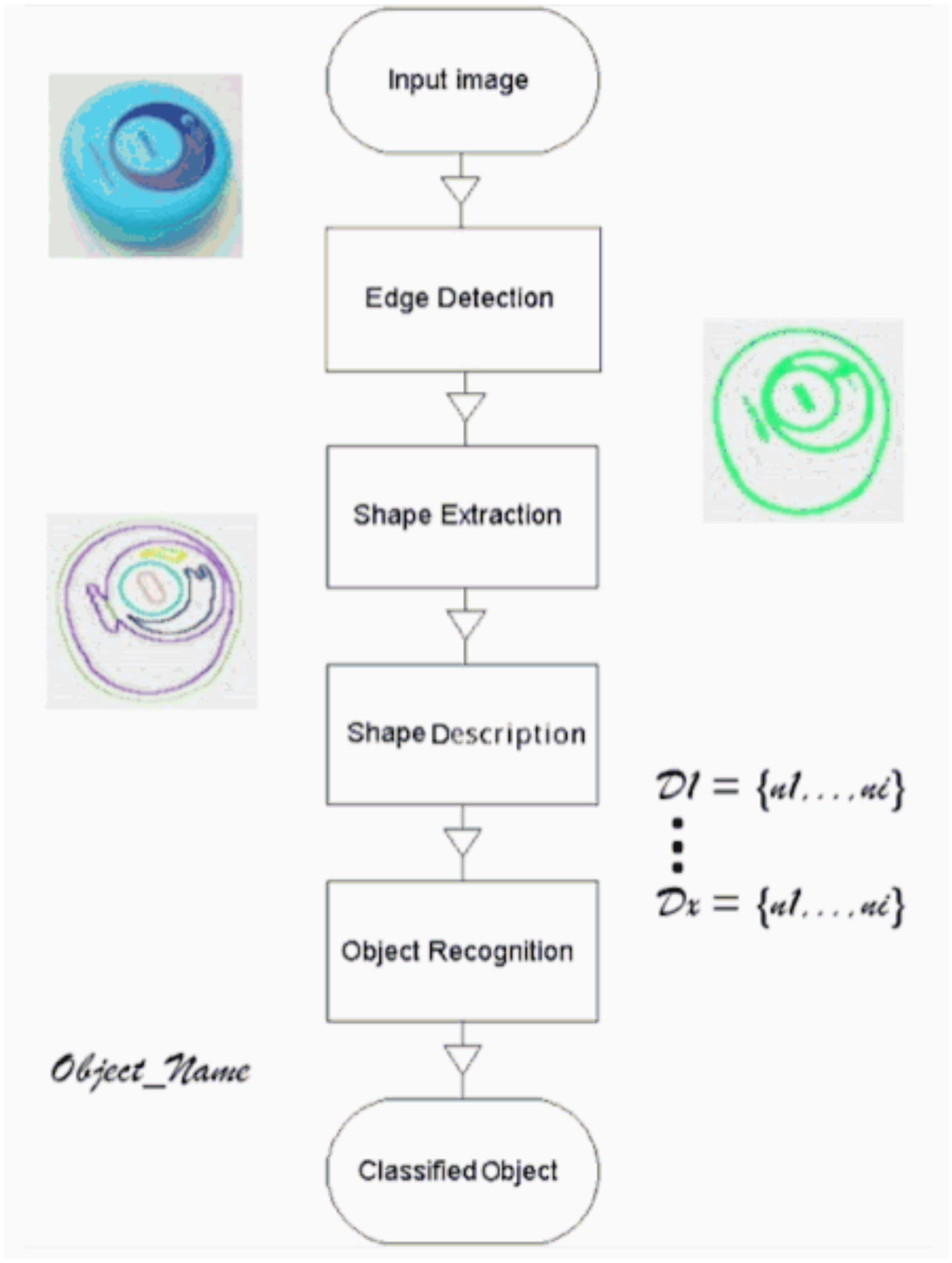


Figure 1.2: An abstract overview of the system we are proposing in this document

Chapter 2

Artificial Intelligence and Object Recognition

Object recognition is an extremely complex problem that will, no matter what the approach, at least for now be bound by its high computational costs. One benefit of trying to tackle object recognition from the AI point of view is that such an approach, be it at times less exact, can often lead to a significant reduction in computational costs. Furthermore, since object recognition is a task we humans are particularly good at, it can certainly be beneficial to look at object recognition from a cognitive point of view.

In our opinion the most interesting question in AI has always been how to generalize. Humans have the great ability to generalize from one situation to another, even if nothing is exactly the same. How do we know when two situations require the same approach, and when they don't? Where lies our ability to see whether things are similar or not?

2.1 Human object recognition

Much of the power of AI stems from cloning human behavior. Therefore, learning more about how recognition in the brain works can help us in developing our solution. Despite the fact that what we know is still limited, we can use a lot of what is known in our program. Gaps left because of what is not yet known about human vision are filled by more standard solutions.

In the first part of this chapter we will briefly discuss one prominent theory on human object recognition. We take a look at Biederman's theory of *Recognition By Components* that discusses the recognition on the basis of *geons*, a selection of entities that make up everything we see.

2.1.1 Recognition by components

One attempt to understand human object recognition is the theory of *Recognition By Components* (RBC) [Bie87]. The basis for this theory is that our primary way of classifying objects is by identifying their components and the relational properties among these components, rather than by features as texture or color.

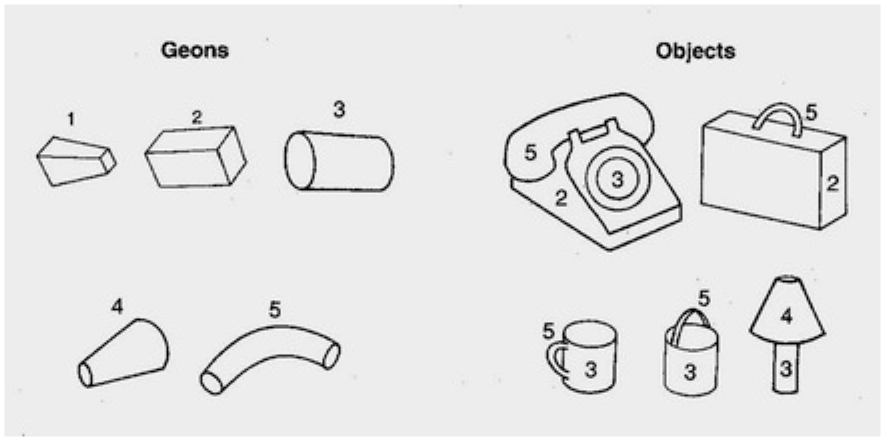


Figure 2.1: A set of geons, and some of the objects that can be composed by them. *Source* [Bie87]

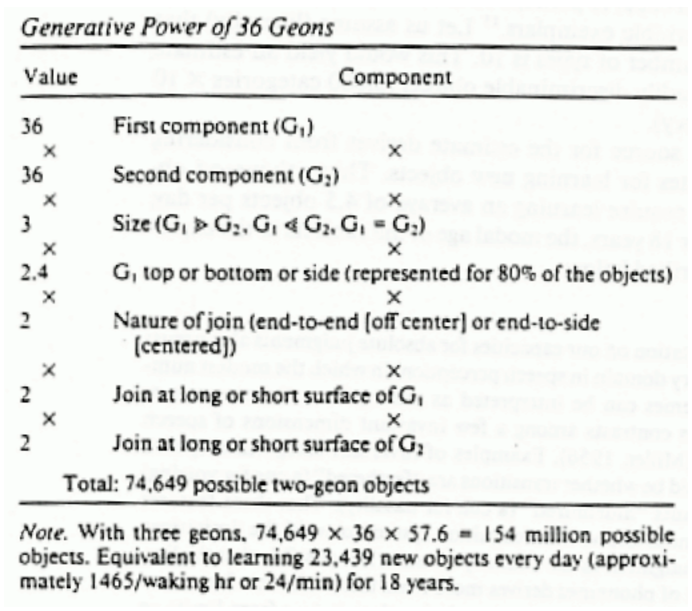
The components Biederman introduces are volumetric geometrical entities that are referred to as geons. Every object can be described in terms of these geons. In figure 2.1 some of Biederman's geons are shown, in combination with some objects that can be composed by them. In total Biederman specified 36 geons that should be sufficient to define every object. Important in RBC is that an object is described not only by the presence of some of these geons, but also by their relations, i.e. the way in which they are connected to one another. In figure 2.1, for instance, we see a bucket and a cup that are both made up by the same two geons. The fact that one geon is on the side of one object, while it is on top of the other object makes it possible for us to distinguish them. To show that 36 geons can be more than sufficient to create all the objects around us we have inserted figure 2.2 from Biederman's article that shows that with just three geons already 154 million objects can be created.

Even though Biederman doesn't include color or texture in object recognition he also doesn't fully ignore them. He reserves them not for object recognition but for the recognition of mass nouns. Mass nouns indicate those entities that are not clearly defined by boundaries like objects are, such as water, sand and grass. These mass nouns can not be quantified in a direct way, i.e. we do not speak of two sands or three grass, but simply of sand and grass.

In order to find evidence for his RBC theory, Biederman tested human recognition, mainly by using response times (RT) as empirical evidence. His tests, that varied visibility of geons and their connections, clearly support his theory. RBC is at the moment still one of the more promising descriptions of higher level human recognition.

For us the most important conclusions drawn in Biederman's research are:

- Recognition is based on decomposition of an object in geons.
- Recognition is more likely to be based on shape information than on color and texture.
- Recognition utilizes the relation among entities, in the form of orientation, distance, size and connections.


Generative Power of 36 Geons

Value	Component
36	First component (G_1)
×	×
36	Second component (G_2)
×	×
3	Size ($G_1 \gg G_2$, $G_1 \ll G_2$, $G_1 = G_2$)
×	×
2.4	G_1 top or bottom or side (represented for 80% of the objects)
×	×
2	Nature of join (end-to-end [off center] or end-to-side [centered])
×	×
2	Join at long or short surface of G_1
×	×
2	Join at long or short surface of G_2
	Total: 74,649 possible two-geon objects

Note. With three geons, 74,649 × 36 × 57.6 = 154 million possible objects. Equivalent to learning 23,439 new objects every day (approximately 1465/waking hr or 24/min) for 18 years.

Figure 2.2: The calculation of the total number of shapes that can be represented using three geons *Source* [Bie87]

- Intersections of lines making up geons are important for recognition, more so than having long untouched lines fully intact.

2.2 Neural networks

By implementing object recognition using neural networks we hope to create a system that is able to generalize well among objects, that is adaptable so that it can easily learn to recognize new objects and that is limited in its computational cost so that it can easily be incorporated into existing systems.

We have chosen neural networks for this task because of their favorable properties that make them an excellent choice for object recognition. The most important of these properties are:

- *Generalization*

  Small distortions can be handled easily, a necessity for object recognition. Accounting for small changes under different conditions (lighting, rotation, etc.) can to a certain extent be left up to training the network under as many conditions as possible.

- *Expandability*

  Another great benefit of a neural network is that it can easily be expanded. In order for it to learn new objects there is no need to start all over and redefine distance measures and distributions. Learning a different set of objects will require hardly any change to the structure of the program.

- *Representing multiple samples*

  A class of objects can easily be represented by multiple samples under multiple conditions. Because a neural network incorporates in its structure what it learns, the recognition of an object becomes a single step. In this way there is no need for multiple comparisons as in many conventional systems. The network determines in one single step to what class the object belongs.

- *Memory Saving*

  An advantage stemming from the previously mentioned characteristic is that there is no need to store all the standard images to be used for comparison. Once a network is trained properly it contains the necessary information and the image data becomes expendable and can be removed from memory.

2.3 Related work

A lot of neural techniques were originally developed for visual pattern recognition. Feed-forward networks like Adaline [WH60] and the Perceptron [Ros58] were two early versions that served this objective. But also one of the first recurrent networks, the Hopfield network [Hop82], as well as the network that is commonly referred to as its extension, the Boltzmann machine [AHS85], were developed for this task.

A general cause for this large variety of neural solutions is probably the nature of the problem. The required amount of generalization over so many different factors that is needed to work with visual data, makes it almost impossible to use raw computation as a solution. Moreover, the problems of vision are far from an exact science, making it extremely hard to lay out the boundaries for recognizing objects in the complex field we are working in. This makes it necessary to reach out to solutions that deal with the problem in another way than using standard heuristics.

Much work has been done in the field of object recognition specifically as well as in describing image content more generally. In this section we will briefly talk about some of the existing methods of object recognition as well as those other applications that deal with the description of image content that might be useful to us. The methods we will talk about are examples of AI approaches to the problem. First we will introduce an artificial technique performing CBIR and then we will look at an artificial object recognition solution.

2.3.1 Content based image retrieval and the PicSOM

MPEG-7

The earlier mentioned field of CBIR is gaining attention quickly. In order to make the comparison of images possible all images need to be described in similar terms. This is why the MPEG-7 [MPE01] standard has been developed. The MPEG-7 standard, also known as the *Multimedia Content Description Interface*, is an attempt to standardize the description of image content. The descriptors it outlines can be used to describe both still images and

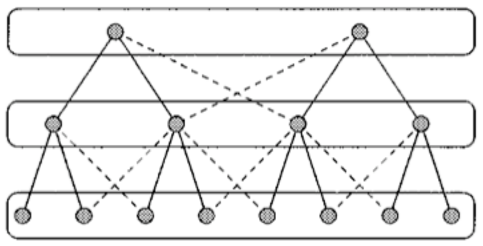


Figure 2.3: An abstract sketch of a Tree Structured SOM. *Source* [LKO02]

other multimedia, like moving images and sound. The hope of its creators is that it will have a similar impact as previous MPEG standards, which have been adopted widely.

The MPEG-7 descriptors are made for indexing shapes, which is not exactly what we want for object recognition. Nevertheless, we are still able to benefit from them. There are several ways of describing a shape included in the standard. Among these there is, for instance, the use of curvature scale space [AMK96] for describing a shape. We will use a version of this later to describe shapes present in the objects.

PicSOM

The PicSOM [LKO02] is a CBIR application that allows the user to find images similar to query images in an iterative process. The PicSOM system is driven by two main principles, *Query by Pictorial Example* (QBPE) and *Relevance Feedback* (RF). As the basis for finding similar images, a tree-structured SOM (TS-SOM) is used, which consists of several SOMs layered on top of each other as seen in 2.3. Each subsequent layer is more detailed than the previous layer. The system has a number of these SOMs running in parallel that have each been trained using image data extracted with different feature extraction techniques.

A query of this PicSOM will take place as follows. Using the principle of QBPE the user selects images from a set of images that is uniformly selected from the SOM that he considers to represent best what he is looking for. In an iterative process, RF, the user keeps getting new pictures that are the result of his query, of which he can indicate whether they are relevant or irrelevant. Places on the SOM where mainly relevant pictures are found are given positive weights, while those where mainly irrelevant pictures are found are given negative weights. Using these weights new pictures are selected for the user to choose from. By continuing the process of RF the user should be able to find a satisfactory selection of pictures.

Since the general workings of the SOM will be explained in section 5.1.3 we will not go into detail here. Interesting is, however, that in an effort to save time a tree structured SOM is used instead of a standard single layer SOM. This reduces the complexity of searching the SOM from the regular $O(n)$ to $O(log(n))$, where n is the number of nodes on the map. This is a considerable gain given the amount of data that needs to be processed. The tree structured SOM is equal to the SOM but built up of several layers, that increase in dimensionality as they lay deeper in the structure. The layers are trained from small to large, where each search for a best-matching unit (BMU) is restricted to an area found under the BMU, for the same image, of the layer above.

The results of the PicSOM are rather good. Special tests were devised by creating sets of similar images, and this showed its ability to find pictures in categories such as faces, sunsets and horses. The PicSOM is available online and can be found at: `http://www.cis.hut.fi/picsom/ftp.sunet.se` where users can experience the process of the PicSOM for themselves.

2.3.2 Object recognition

The PicSOM is an impressive application of artificial intelligence and image content identification. However, there are also a lot of object recognition solutions. Here we will discuss one of the more famous ones that uses more than the standard three layers to give the system more power.

NeoCognitron

One early attempt to perform pattern recognition is the Cognitron proposed in [Fuk75]. This was a self-organizing network that learns to distinguish between patterns. The drawback of the system is, however, that it is very sensitive to shifts in position and distortions of the pattern. In order to overcome these shortcomings, the NeoCognitron was introduced [FM82], a newer version much more robust to distortions and shifts.

The NeoCognitron is a multilayered self-organizing network composed of four different types of neurons. First there are the simple (S) cells and the complex (C) cells, which are both excitatory neurons. Among these, in smaller numbers, there are also inhibitory versions of both neurons present, the simple inhibitory (V_S) cell and complex inhibitory (V_C) cell. Using these neurons a multi-layered network is created.

The network starts with a layer of input *photocells* after which an even number of layers is inserted, alternatively containing either only simple (type S and V_s) or only complex cells (type C and V_c). Between these layers (not within) are the so called interconnections which allow activations to be propagated through the layers. The interconnections going from simple to complex layers are fixed and stay the same during the organization process. The connections to the simple layers are updated to allow the network to recognize patterns.

The activation (w) for a simple cell is calculated as follows[1]:

$$w = \varphi[\frac{1 + \sum_{v=1}^{N} a(v) \cdot u(v)}{1 + b \cdot v} - 1] \tag{2.1}$$

Here $a(v)$ are the weights of the incoming nodes from the previous layer and $u(v)$ the corresponding activations. Furthermore b is the weight coming from a single inhibitory input and v is its activation.

Function $\varphi[\cdot]$ is defined as:

$$\varphi[x] = \begin{cases} x & \text{if } x \geq 0; \\ 0 & \text{if } x < 0. \end{cases} \tag{2.2}$$

[1]We have taken a somewhat simpler version of the actual formula to keep the explanation from being too complex here

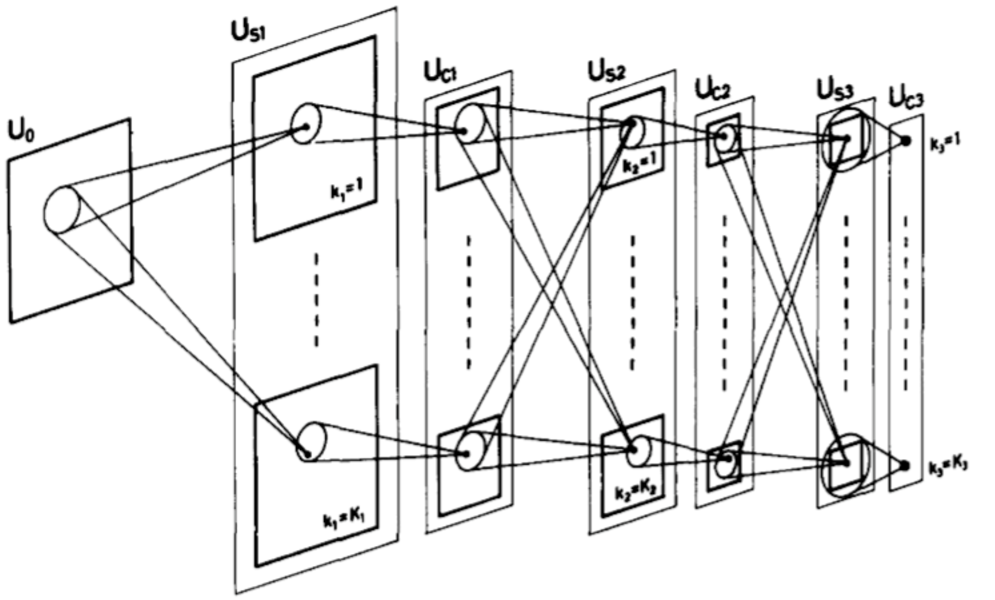


Figure 2.4: A visualization of the connections between the layers in the NeoCognitron. The larger thin boxes represent layers, while the smaller thick boxes represent planes. *Source* [FM82]

In a similar manner the activation of the complex cell is calculated, now however the φ formula is substituted by the ψ which is defined as follows.

$$\psi[x] = \begin{cases} \frac{x}{x+\alpha} & \text{if } x \geq 0; \\ 0 & \text{if } x < 0. \end{cases} \tag{2.3}$$

Here α is a constant greater than zero. The inhibitory cells have just one kind of input, making the activation of simple cells a weighted mean of the inputs. The activation for complex cells is similar but a bit more complex, it is:

$$w = \sqrt{\sum_{v=1}^{N} c(v) \cdot u^2(v)} \tag{2.4}$$

Here $c(v)$ are the weights of the incoming connections.

The idea behind the system is that each layer of nodes represents less detailed features at each level. Looking at figure 2.4 we see that the first layer is large and contains detailed planes (the thick lined subsets in the layers) linked to the input. Moving forward through those layers shows a decrease in detail. However, with each node receiving input from a circular field from the previous layer the space represented by the node (its receptive field) becomes larger as we progress through the layers.

A more practical sketch of its workings can be seen in figure 2.5, where, for instance, nodes in the first layer might represent just the corner, intersection or maybe a small piece of line of the letter A while the last layer's nodes each represent an entire pattern.

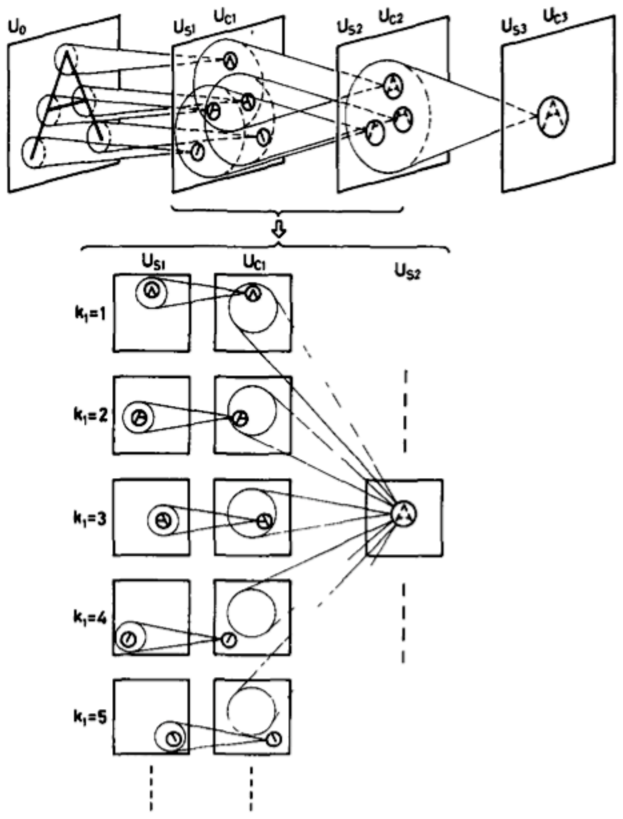


Figure 2.5: An example how an 'A' could be represent by the NeoCognitron. *Source* [FM82]

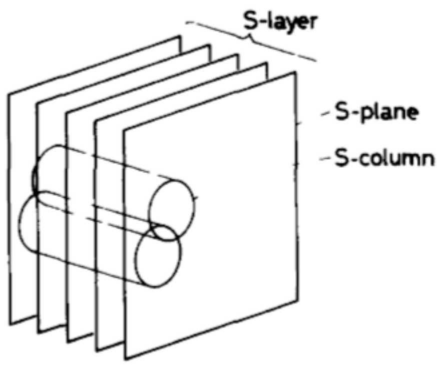


Figure 2.6: Planes from a single layer, with a visualization of how the columns cross through them. *Source* [FM82]

During training, the weights in the NeoCognitron are updated as follows. After the activations for all the cells have been calculated, representatives are chosen in planes that will represent the presence of a certain feature. A representative is selected on the basis of columns and layers as drawn in figure 2.6. Columns are drawn throughout the planes of a layer. For each column the cell with the maximum output is found. After all of these cells have been found, the one with the highest value in each plane will represent the plane and its weights will be updated. The update is done on the basis of the neuron's weight in the activation of the representative.

This selection method, a version of the 'winner takes all' method, should push neurons to become representatives for specific features. As we travel down, the increased receptive field should allow for neurons on the first layers to represent specific features, while neurons should represent more global features on the last layer. This finally leads to an accumulation in a last layer, where neurons that represent a full pattern are present. For more technical details we refer the reader to [FM82].

Chapter 3

Preprocessing

In this chapter we will describe the preprocessing performed on the images. The ultimate goal of this step is to extract the shapes present in the image. Preprocessing an image is unfortunately a costly, but necessary, process. It generally involves running through images pixel by pixel and performing numerous calculations using this pixel and its surrounding pixels. We want to keep this step from becoming the bottle neck of the system. This unfortunately makes it necessary to sometimes trade-off quality for reduction of computational costs. Even after limiting the costs this processing is allowed to take, it will still be by far the most costly process of recognizing an object.

The extraction of shapes from the image is broadly performed in three steps. First we use an edge detection algorithm to find the *edge pixels* in the image. A second step uses the generated *edge pixels* to create the contours of the shapes present in the image. In a third and final step the resulting image of the second step will be used to extract the shapes that have now become visible. In some cases[1] extra preprocessing can help to make the shapes more appropriate for training a neural network. Optional preprocessing that can help prepare the information for the neural network will then follow these three steps. In this chapter these steps will be described in depth, but before continuing we will first devote a small section to the kind of shapes we are about to extract, and the requirements these shapes need to fulfill in order to be suitable for object recognition.

3.1 Shapes

As mentioned before, the creation of the object recognition system has partly been inspired by theories on how humans perceive objects. Taking into account the earlier described approach by Biederman, in extracting features we have tried to focus on obtaining primitive shapes that make up the image.

The main motivation for the recognition by shapes is that in recognition of an object as a whole every discrepancy, even if it is only a small one, can disrupt the recognition process. As such, trying to recognize an object as a whole is possible as long as the object is not partly covered, or seen from a point of view that exposes a different side of the object than

[1]These cases will become clear in the following chapter where we will explain different ways of describing a shape

the system has been trained for. To make sure recognition can also take place in less perfect situations, great benefits can be reaped from distinguishing the elements that make up the object. Often just small parts can indicate the presence of an object. Just the presence of a hand can indicate that there's a person present or a set of wheels can give reason to believe a car might be in the scene. When using shapes for recognition the obstacle of occluded information can much easier be overcome. Furthermore, using loose shapes makes it much easier to perform recognition of multiple objects in a scene, since in that case the scene will no longer form a single input. How we intend to use these shapes in recognition will be the center of discussion later in this document.

Having said this the goal of the preprocessing step has evolved from being the standard approach of extracting edges from a picture, to a more extensive task to extract the shapes that make up the picture. Before starting the discussion on how this is done, we need some ground principles as well as a way to refer to those principles to avoid any confusion. When we are talking about the *edge pixels* we refer to those pixels that lie on any kind of edge detected by the edge detector. We shall make a distinction between two terms, on the one hand we have the *object*, while on the other there are the *shapes*. The *object* refers to the straightforward meaning, i.e. a thing we can see or touch. The object will be described by a set of *shapes*, making shapes the parts that make up an object. It would of course be ideal if we could divide the object in geons such as described earlier in RBC, i.e. volumetric entities that have several clearly distinguishable properties, such as being symmetrical or not, rounded or sharp, thick or thin, and so on. This. however, is difficult to extract from a real color picture, with all kinds of noise, that generally becomes even more prominent during preprocessing. Fortunately, in the end what we need from these shapes is not to be smooth primitives. The main property we require is for them to be consistent, i.e. be the same in all objects of the same category. Which is what we will strive for in this chapter.

3.2 SUSAN

After first having used the well-known Canny edge detection algorithm [Can86] to find the edges, we came across a newer edge detection tool, SUSAN [SB95]. The main advantage of this algorithm is that the edges it produces are more solid and better connected, i.e. there are less missing parts in edges. This makes it easier to extract whole shapes. Furthermore, due to a relatively simple implementation the switch from Canny to SUSAN was easily made. And finally SUSAN is a faster algorithm than Canny which is important in the creation of this most costly step of the system. SUSAN for instance doesn't require smoothing of the image like Canny does.

The difference in approach between Canny and SUSAN is that SUSAN doesn't approximate derivatives in the pixel space of the image, a technique long used in finding edges in an image. Rather, it determines per pixel whether it lies on an edge, by comparing it to the pixels in a surrounding area. The abstract version of the SUSAN algorithm can be seen in figure 3.1.

We will however not use the full SUSAN algorithm, but only the first three steps to obtain the edge pixels, this will be followed by a more specific approach aimed at extracting shapes from the image. We will now describe the workings of the used steps from the algorithm.

```
SUSAN Edge Detection
for each pixel do
    1) Place circular mask around pixel in question
    2) Calculate the USAN, the number of pixels within the
       mask that have a similar brightness value
    3) Subtract the USAN from the threshold to produce the
       edge strength of the pixel
    4) Use moment calculations applied to the USAN to find
       the edge direction
    5) Apply non-maximum suppression, thinning and sub-
       pixel estimation, if required.
```

Figure 3.1: These steps make up the SUSAN edge detection algorithm

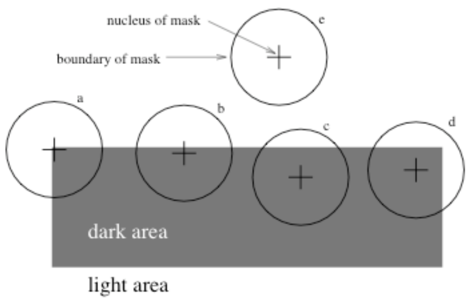


Figure 3.2: A number of circular masks used to mark the area around a pixel. *Source* [SB95]

Since to us however the preprocessing part is not central to this research we will not always fully go into detail and provide a full account for assumptions made. Rather we will explain the techniques on a more intuitive level. We would like to refer the interested reader to [SB95], where a full account can be found for all the assumptions.

3.2.1 Calculating the USAN

USAN stands for ”Univalue Segment Assimilating Nucleus”. It is a partial area within a larger, circular area of pixels that have approximately the same value as its *nuclues*, the center-pixel of the area, does. The more pixels are similar to the nucleus the bigger the USAN area is, reaching a maximum when all pixels in the area have the same value. In the SUSAN algorithm the USAN is calculated for every pixel by making it the nucleus (or center) of a circular mask that passes over the picture. It can easily be seen that as the USAN becomes smaller the chance that it lies on an edge becomes greater. In figure 3.2 examples of circular masks are distributed over a drawing of a rectangle, accompanied by figure 3.3 where the USAN area of these circular masks is shown in white.

To get an even better idea of how the USAN area changes when encountering an edge we

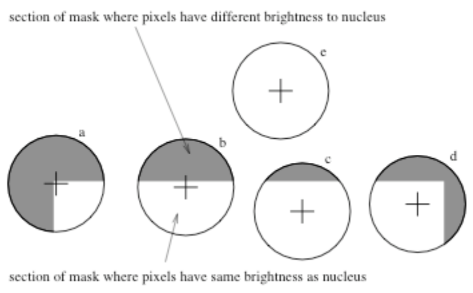

section of mask where pixels have different brightness to nucleus
e
a
b
c
d
section of mask where pixels have same brightness as nucleus

Figure 3.3: The USAN area of the circular masks in figure 3.2 is colored white. *Source* [SB95]

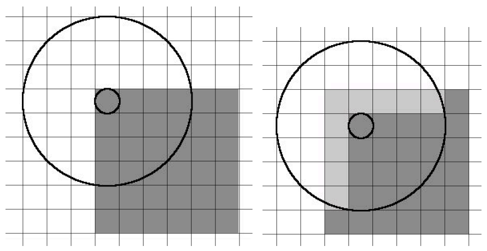


Figure 3.4: The difference between a nucleus on an edge and not on an edge

have included figure 3.4 which clearly shows the difference in the USAN area that is on an edge and the area that is just off the same edge. The property of a circle being the widest in the middle is cleverly exploited here to create this effect. Any shift around the middle, the place of the nucleus, has a much greater effect than on the outer edges of the circle.

The comparison of a nucleus to other pixels can be done using an absolute threshold for the difference allowed. However, it has been shown to perform better using an equation that allows for a smoother comparison.

$$c(r, r_0) = e^{-(\frac{B(r)-B(r_0)}{t_{sim}})^6} \tag{3.1}$$

Here $B(r)$ is the brightness of the pixel, t_{sim} is the similarity threshold and r_0 is the nucleus. Using this comparison measure, the value of the entire USAN can be calculated by running the comparison through the entire area, adding all comparisons:

$$n(r_0) = \sum_r c(r, r_0) \tag{3.2}$$

3.2.2 Finding the edge pixels

After having applied the previous step for each pixel a 3D landscape of USAN values has formed with a large amount of local minima as in figure 3.5 (note the inverted scale). In the next step of the SUSAN algorithm these local minima are filtered by comparing them

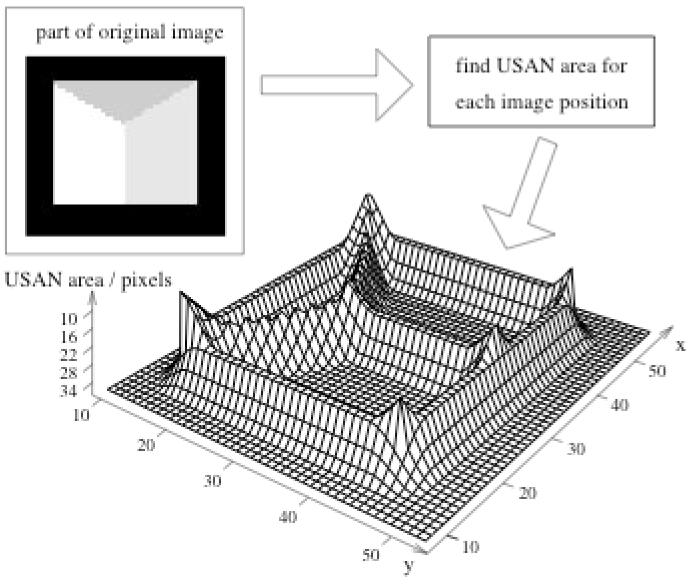


Figure 3.5: A resulting edge map with the corresponding input image. *Source* [SB95]

to a threshold. In general the threshold is not a fixed constant, but is a variable based on the data gathered in the previous step. The threshold is set to 3/4 of the maximum USAN found. This leads to a lower threshold in images that show more variations, in contrast to for instance simple black and white images, where large parts of the image have the same brightness value. The *edge response* is now formally approximated as in equation 3.3

$$R(r_0) = \begin{cases} t_{circ} - n(r_0) & \text{if } n(r_0) < t_{circ}; \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Here t_{circ} is the threshold that decides how great the USAN should be in order to be considered an edge pixel. Since in our approach no distinction is made between edge pixels of different $R(r)$ values we will reduce all pixels to being either an edge pixel (having a $R(r)$ greater than zero) or not (those pixels with a zero value for $R(r)$).

3.2.3 Notes on implementation

Before continuing we would like to make some remarks on issues that arose during the implementation of these steps. As the authors suggest in their article on SUSAN the algorithm can run much more efficiently when, instead of calculating equation 3.1 every time, a lookup table is used that fills out the values initially, based on the domain of the brightness, and then use this table to find the necessary results[2].

[2]The increase in speed of this simple step is certainly worth its implementation. The formula has to be computed 37 times for each pixel, meaning that on a 200 by 200 image, the formula will be calculated close to two million times. With the lookup table the processing time of an image is now decreased to approximately a fourth of the original time.

The SUSAN edge detector depends on the brightness of pixels. In the program the brightness is calculated using the formula recommended by the *world wide web consortium* (`http://www.w3.org/`). This formula uses the red, green and blue components of each pixel, each contributing a different factor to the brightness, for the exact scaling see appendix B that deals with the digital processing issues. Unfortunately this does leave the possibility for different colors to have the same brightness, which could lead to the failure of detecting some edges in the image. One remedy would be to work with a similarity match based on all of the separate components of the rgb-value of a pixel, without combining them to one reduced value. Unfortunately it was difficult to find information on edge detection using these kind of similarity measures. In the future it does seem worth trying a color-based approach as it would increase precision. However, this would lead us outside of the scope of this thesis which focuses on the recognition process.

Furthermore, it should also be noted that a full circle cannot be drawn around the nucleus when a pixel is too close to the edge. This is remedied by making those pixels of the circle not on the image equal to the pixels on the edge of the image they are closest to. This is based on the assumption that the best guess for the values of those pixels is the value of the closest available pixels[3].

3.3 Creating contours, extracting shapes

After generating the edge image, the next thing we want is to extract the shapes in the image. To do this we have used an intuitive method to find all shapes present in the image. This method of extraction is composed of two steps. In the first step the image containing the edge pixels is dilated, making all edges one pixel thicker on each side. In the second step the original edge image is subtracted from the dilated result[4]. The result is an image with whole shapes, i.e. there is a beginning and an end to each of the created contours. These contours will now be regarded to as the shapes that make up an image. These shapes are rather insensitive to an object's rotation, noise etc. They are therefore a good basis for representing the object, hence making them a good basis for our recognition process. An example is given in figure 3.6, where every shape is given a different color.

In a final step we can now, from the newly created image, easily collect all shapes by finding a shape and gathering its pixels in a *depth-first* style to create an ordered chain of pixels, which can be saved in a vector. This process is described in figure 3.7.

3.4 Further preprocessing

There are even more low-level imaging steps that we can use to make shapes more easily identifiable. These steps can especially help improve recognition when using what we will refer to as *direct descriptors*, i.e. those representations that do not attempt to overcome

[3]In one experiment where we were working with simple shapes the background color was known, we substituted this by taking the value of the background color for these pixels

[4]The implementation actually combines these two step to one to keep the procedure from becoming too computationally intensive

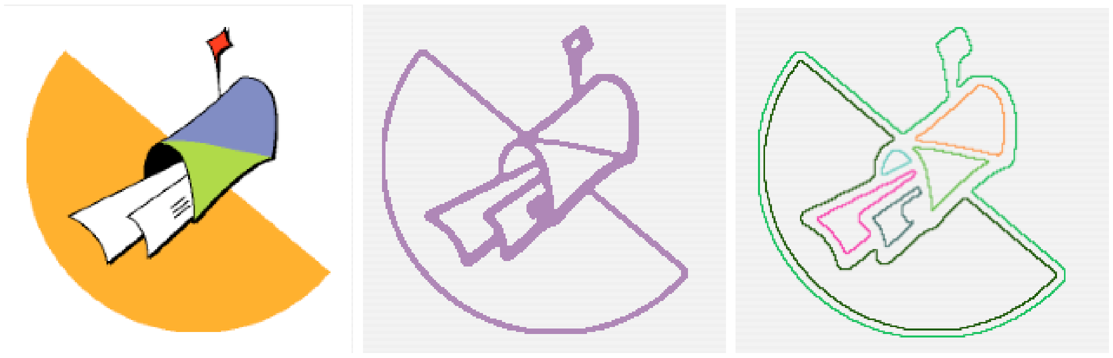


Figure 3.6: A resulting edge map, shown with the corresponding input image

```
Shape finding
object = new Vector();
for each pixel do
    if pixel.isEdgePixel() then stack.add(pixel);
    shape = new Vector();
    while $\neg$ stack.isEmpty() do
        pixel = stack.pop()
        shape.add(pixel)
        stack.add(getNeighbours(pixel))
    object.add(shape)
```

Figure 3.7: The pseudo-code for extracting shapes from the contour image

issues of affine transformations in any way. The best example is the pixel representation of a shape. One rather simple manipulation is the rotation of a shape to a standard position as described below. As on the other hand some representations of shapes themselves deal with perspective issues this part of the preprocessing somewhat overlaps with the creation of those representations, that is done in chapter 4. This is why we will later split the representations in two sets, those that can benefit from extra preprocessing (*direct representations*) and those that do not benefit (*descriptor representations*). This extra preprocessing will only be used for those representations that can benefit from it. In these extra steps we will take into account that it is not possible for a neural network to recognize shapes that are presented in a way that activates it in a completely different manner than seen before. This is, for instance, the case when a shape is flipped upside down, or rotated over large angles. The necessity of this step lies in the requirements of the final representation of the input we will supply the neural network with. The task of trying to present a shape in one single orientation, size etc. is not an easy one, especially as similar shapes can have a number of varying properties. We will not attempt to process every shape to a state where it is in the exact same situation no matter how it deviates from the original. However, if we are just able to reduce the number of possible presentations of a shape this can already bring great benefits to the recognition process using neural networks. Going from a large number of presentations to a subset will greatly enhance performance.

3.4.1 Rotating and centering the shapes

In this step the objective is to rotate shapes to a generalized position so that the number of possible projections of the shape onto the network is reduced to a subset of the possibilities present when a shape can appear in all of the different positions. This does require some kind of heuristic to decide on how to rotate these shapes to this generalized position. Considering we don't have much a priori knowledge on the shape the best option seems to be to rotate the shape over some factor that is generally constant for each shape. Our choice has been to rotate the shape over its longest chord (the longest line one can obtain by connecting two points in the shape). After the orientation of the longest chord is calculated all points are rotated to their new position. During the process of rotation they are also re-centered to the center of the plane so that recognition does not suffer from shapes appearing in different positions. Of course this does not always lead to the same rotated positions. Take for instance a cup with an ear. For some of these cups the longest chord will be found from a point at the top of the cup to a point at the bottom. However, for another cup the line might extend from a point on the ear to a point on the other side of the cup. An example of these two different cases can be seen in figure 3.8. Even though this process does not lead to a deterministic result it can easily reduce the rotation of many shapes to just two or three rotations, which is a good reduction factor considering the total number of possible rotations.

3.4.2 Flipping the shape

As mentioned earlier, just rotating still leaves us with the fact that a shape can be mirrored, making it appear very different from the non-mirrored shape. To further reduce the number

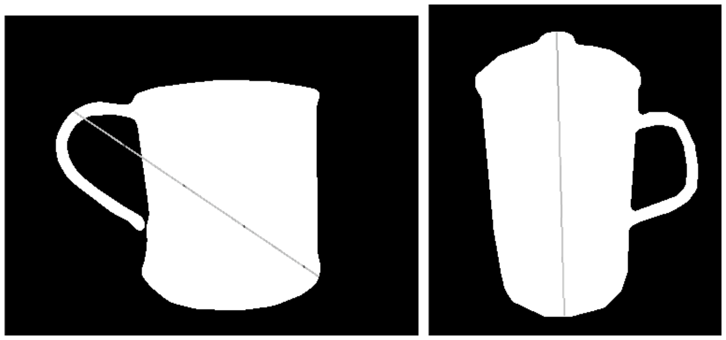


Figure 3.8: Two cups with different orientation of their longest chord

of positions the shape can be in, we can also attempt to remedy the mirroring possibility. The heuristic chosen to flip shapes to the same side is applied after rotating the object to a *generalized* position. We then calculate the area of the shape in the four quadrants of the coordinate system, with the center of the shape as the origin of the system. After determining which quadrant has the largest area we flip the image in such a way that this area is in the left upper quadrant. Again, this is not a method with a deterministic result, but it does lead to a reduction in the number of positions a shape can be presented in. This makes it easier to let a shape be recognized by a neural network that does not generalize well among situations in which a shape would be projected upside down.

3.4.3 Resizing the shapes

To improve the performance of the neural network further, all shapes can be resized to a single size so that the edge pixels will fall in about the same area in most cases. To find the scaling factor we use a somewhat similar heuristic as the longest chord, only now we take the point that is furthest from the center. This point will be scaled to let it have a distance exactly equal to half of the width of the plane on which the shape lies. Dividing the width by this furthest distance gives us the scaling factor, which we then use to re-determine every point's distance from the center. With this scaling factor every point moves either away from or closer to the center. The reason for choosing half of the width of the plane is that in this way when rotating no points can fall outside the image the shape lies on.

Because moving the pixels away from the center when we enlarge a shape can result in gaps between those pixels, which could have a poor effect on recognition, we in a final step draw lines between those points that are not touching anymore after the resizing has taken place. Likewise, points that have come to overlap each other due to downsizing will be removed when a distance of zero is discovered between subsequent points.

3.5 Short summary

As a summary of the preprocessing that will be used in our approach we have inserted diagram 3.5, that shows the entire process. The left side-branch indicate the extra preprocessing that is performed in some cases.

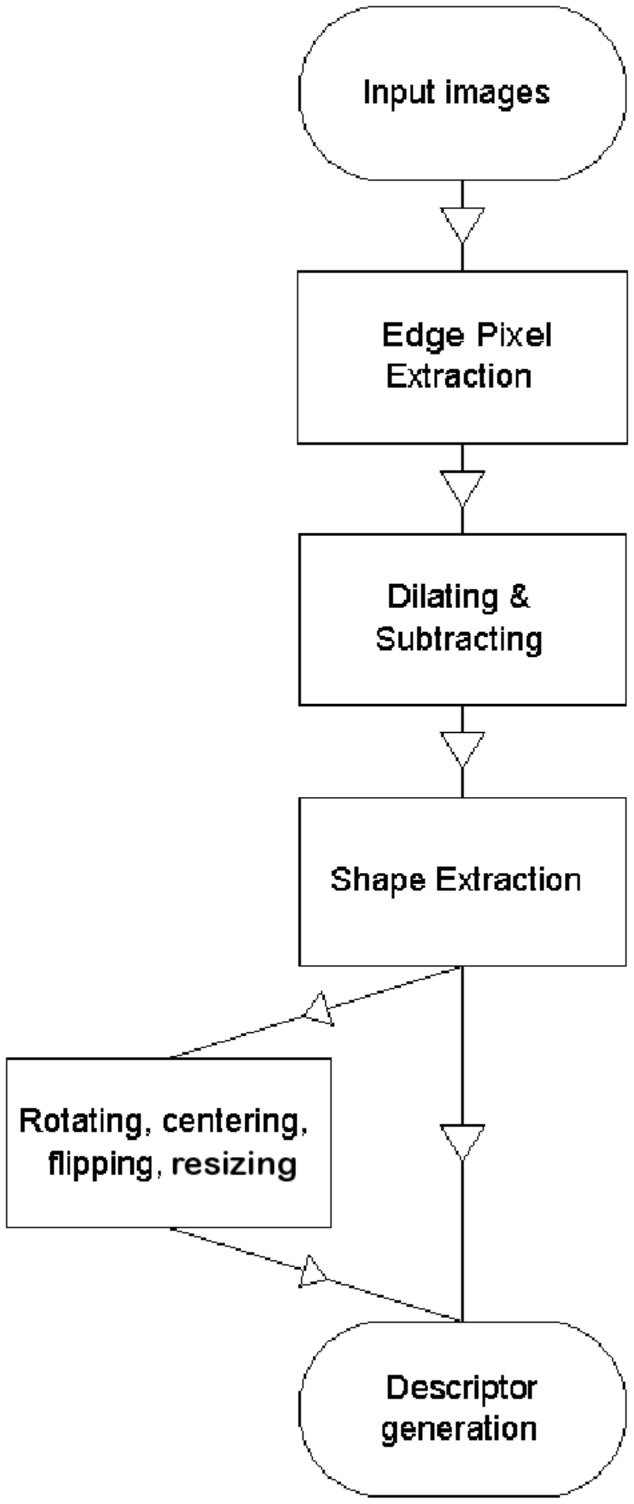


Figure 3.9: The full process from top to bottom

Chapter 4

Representations

In this chapter we will create representations of the shapes extracted in the preprocessing step in order to prepare them for feeding into a neural network. We shall refer to these representations as *shape descriptors* or, when there is no room for confusion, just *descriptors*. A shape descriptor is a set of numeric values that describes the shape in a way that makes it distinguishable from other shapes. For instance, the total area of a shape or the number of pixels on its perimeter are simple ways to describe shapes. In our system we have created and tested several different descriptors to represent shapes. All the descriptors we are about to describe have been tested in our application. Those descriptors that worked best have been used to test our final adaptive system.

The descriptors we will use range from very simple ones, just representing the shape by using its pixels directly, to more complex descriptors that fully deal with the various viewpoints a shape can be viewed from. The importance of the latter kind of descriptors is not only that they greatly reduce the size of the shape's representation. The descriptors are also less sensitive to scaling or rotation of the shapes, so that a shape that is upside down can result in the same representation as a shape in normal position. The quality of these descriptors is of course not just determined by how well they deal with simple rotation or scaling. Also small distortions or affine transformations should not result in great variances of the representation. It is important that shapes we consider to be alike have representations that are alike, while at the same time they should also make it possible to adequately distinguish between different shapes.

We will provide justification for the descriptors used, although we are aware that the creation of a shape descriptor is not an exact science. Determining the entropy of a descriptor on paper is not directly possible. This is why the main way of determining the quality of the various descriptors will be to test them in the recognition processes. For our application we have selected five different descriptors to represent shapes. Before we will introduce these, we will first discuss, in the following section, what we require from these descriptors, and what we consider to be a good descriptor.

4.1 Ideal descriptors

In an effort to lay down some ground rules for the description of shape we will in this section discuss the desirable properties of a good descriptor. Below we have listed six properties that we consider the most important in the design of a descriptor.

- *Capture essential information.* A quite trivial requirement of any descriptor is that it should capture as many as possible of the features that distinguish the specific shape. Such features as, for example, the handle that is present on the cup, should somehow be coded into the descriptor. But also more general features should be enclosed, e.g. if being square or round is an important feature of this shape, it should somehow be translated into the descriptor.

- *Efficient calculation.* As we have already mentioned in the previous chapter, working with data from images can be a costly process. Even though after the extraction of the shapes the data has become easier to work with, it still can lead to tedious processing even to perform simple operations on it. This is why it is important that a shape descriptor should be easy to calculate and create, to make sure that the data manipulations do not become the bottleneck of the process.

- *Invariance to viewpoint.* One of the most important properties of a good descriptor is invariance to rotation, size, translation, etc. The classification process can benefit a lot from a descriptor that deals with these issues. It is not always easy to overcome these problems. Making a shape rotation invariant means that the descriptor of the shape should be the same regardless of the viewpoint. Merely having information in the same order is not enough, the sequence should also have the same begin and end point. To some extent the preprocessing part of the application can also deal with these issues. But the quality of the preprocessing techniques is not perfect, and still takes a lot of extra time. This can be avoided by having a good descriptor that is easy to retrieve.

- *Compact description.* Since the shape data consists of pixels on a plane a neural network having to deal with all this information will be extremely complex and quickly exceed the limits of the computational power of the general computer. As we will see in creating the shape descriptors, we can easily reduce a shape in a field of ten thousand pixels to a vector of a mere hundred values, which reduces the input size by a factor hundred.

- *Robust.* Small changes in a shape in the form of deformations or noise don't generally mean an actual change in what a shape stands for. This means that a good descriptor should be robust when it comes to noise, like minor features in the shape that are not of particular importance.

- *Don't over-assimilate.* Even though a descriptor should be robust, there should of course also be limits, since it should still be sensitive to changes between different shapes. Using, for instance, the area of a shape or the number of points on the perimeter will result in equal values for very different shapes. The descriptor should only generalize over small amounts of noise and unimportant features.

Often the points above have to be traded off against each other since the advancement of one can hurt the fulfillment of the other. Making a descriptor as compact as possible can damage its descriptive quality. Making a descriptor too robust under shape changes can lead to assimilation of various shapes that should be seen as different. Fortunately it is not essential for the shape descriptors to be perfect. As we will see when we introduce our adaptive system in the next chapter, shapes will in turn be combined to make up an object. Having many shapes that make up an object, loss of information on the individual shape level will not necessarily do much damage on a global level.

Aspects of shapes that we do not include in our description are color and texture. We do want to emphasize that this is not because we believe this does not convey any extra information. When distinguishing between an apple and an orange it is without a doubt helpful to have the knowledge that one is orange and the other red, yellow or green. So in an ideal system color and texture information information will definitely have to be used. It would be a loss of information that could increase the certainty of a match in recognizing objects. However, in our view, based on such research as that of Biederman, a robust recognition system should mainly rely on geometrical aspects rather than color and texture. Therefore, in this thesis we will limit ourselves to investigating the power of recognition of shapes and leave other kinds of information present in the image to future research.

4.2 Direct descriptors

First we will describe the descriptors we will refer to as *direct descriptors* in our application. These descriptors, which include both pixel representations and orientation descriptors, have been labeled so because they deal with the shape *as is*, and leave it up to the preprocessing steps in section 3.4 and to the neural network to provide for the recognition under different orientations. Further in this chapter we will also introduce invariant descriptors that deal with issues in the representation of the shape itself. This means that for these descriptors the left branch of the preprocessing process (as can be seen in figure 3.5 in the previous chapter) will be used to prepare the shape for the adaptive system. These descriptors lack most of the properties of an ideal descriptor and are inserted for a different purpose. They, especially the pixel representation, are included primarily as a benchmark for the other descriptors. The orientation descriptor that follows the pixel representation is based on some neurological aspects of recognition. Its main function is to reduce the number of inputs from the pixel representations.

4.2.1 Pixel representation

The easiest way to represent a shape is by its pixels[1]. Creating a neural network with an input layer that has a neuron for each pixel in an image allows us to directly project shapes on the input layer of the neural network. In this way the neurons corresponding to the pixels that are on the shape become activated, while the others stay inactivated. As has already been noted earlier, this does lead to a lot of problems. It, for instance, requires extra preprocessing of the image, as described in the last part of the previous chapter. But

[1]This representation is sometimes also referred to as a grid representation

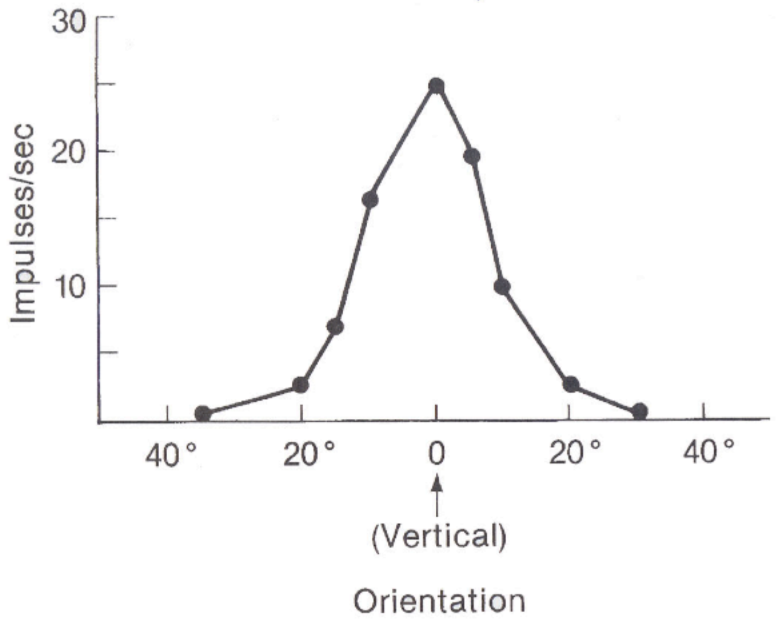


Figure 4.1: A plot of the average spikes set out against the orientation *Source* [HW59]

even with this extra processing there are no guarantees that we are working with something that can be the basis for good recognition, since we are still dependent on a structure in an extremely large space. Of course also the calculation of other descriptors will add extra cost when having to deal with the problems of pixel representations. However, the reduction in the size of the representation that can generally be made when using them as input to a neural network easily compensates for these extra costs. This leads to not only a reduction of the input neurons, but also the neurons in following layers, greatly reducing the number of connections to be calculated and trained.

4.2.2 Orientation descriptor

As our knowledge of the brain continues to grow we learn more and more about its fractionated structure. Most neurons seem to have their own little simple task, the results of which they redistribute to many other areas of the brain. One such task is described in the late fifties in [HW59]. It was found that the so called simple cortical cells in a cat respond more actively to a bar of light at a particular orientation. While measuring the response of a simple cortical cell the orientation of the bar of light was changed, and the cell responded most actively to a single orientation, i.e. for one orientation the number of spikes produced was much higher than when the bar was rotated away from this orientation. In figure 4.1 the spikes per second are set out against the orientation, showing a shape that resembles a Gaussian function. Since then a lot of research has been done, resulting in the discovery of many more cells that perform a simple task on a small receptive field (complex cells, end-stopped cells and many more). This inspired us to also implement the following way of describing a shape.

For this descriptor the plane on which the shape lies is partitioned into small squares of the same size. Each area will serve as the receptive field of two input neurons. One of these neurons becomes activated when there is a horizontal line passing through the square, the other neuron is activated by vertical lines. If no line is present at all the activation of both neurons will be zero.

When the neuron is presented with lines that are neither vertical nor horizontal the activation of the neuron is based on the deviation of the angle that represents its maximum activation (vertical and horizontal respectively). The degree of activation is determined using a Gaussian function inspired by the one that we have seen above.

Of course an important that remains is what the size of the squares should be. Partly this should dependend on the size of the input image. Considering our input images are rather small, usually around 200-by-200 pixels, we have chosen to take 4-by-4 squares as receptive fields of these neurons. By keeping the squares this small we can be sure lines do not start curving too much and will always be relatively straight.

Now of course we are aware that the line crossing through the 4-by-4 square is still not always necessarily a straight one. However, since the square covers only a small discrete space chances of lines deviating from a straight line are rather small. This is why we can safely generalize the original line to a line between the place where it enters the area, and the place where it leaves the area. All the information on angles is now calculated as if these two points were connected by a straight line, even in the few cases where this may not be the case. When the *in* and *out* point are not just a single pixel, the line is drawn between those points that are furthest apart, assuming that that is our best guess for the continuation of the line outside of the area.

Of course this approach is not the same as the physiological principle previously described. It also can not be expected to show huge advantages in shape classification in comparison to the previously described pixel input. However, that was not the reason for introducing it. As long as this descriptor is at least as powerful as the pixel descriptor it has one major advantage: the number of input neurons can be cut back to an eighth of the original input layer, which results in great computational benefits. A simplification of the input layer of the neural network can also decrease the number of hidden neurons necessary, which leads to an even further reduction than by a factor eight (16 pixels per square, 2 neurons per square) when it comes to the costly connections that make up the neural network. And a reduction in the size of the input and hidden layers leads to a quadratic reduction of connections.

4.3 Invariant descriptors

We will continue our discussion of descriptors with the *invariant descriptors*, i.e. those that describe shapes in a way that makes them as invariant as possible to the perspective the shapes are viewed from. This means that these descriptors will attempt to deal with rotation, mirroring, size and translation variances.

4.3.1 Radial representation

A popular way to create descriptors of shapes, as well as those of objects, is by the use of circles. Centered on the geometrical center of a shape they can easily be used to extract rotation invariant features. The main purpose of the descriptor we will describe now is to create a representation of the shape that extracts as much information of the shape as possible while still providing for a robust representation. In this descriptor we shall also use circles to achieve viewpoint invariance, inspired by a technique in [TMRSSG00]. For this approach we calculate the angles between points that lie on (intersect with) a circle of which the center coincides with the center of shape. Since these angles are the same no matter what the rotation of the shape is, they form a good basis for representing a shape invariantly when it comes to rotation. To make sure the size of the shape is not a factor in the descriptor, the radius of circles used is expressed as a proportion of the distance from the center to the point that lies the furthest from the center.

For a single circle this will result in a finite number of angles, which can just as easy be one as it can be five (an example can be seen in figure 4.2 on the left side). Since for a neural network we will need a steady input size, we have decided to include only the biggest and smallest angle of the circle. To avoid totally ignoring the other crossings, we will also include the number of crossings of the shape and circle, leaving us with three values for each circle. To represent the shape we draw n equidistant circles on the shape (see the right side of figure 4.2), where for each circle we include the information just mentioned in the descriptor. Of course the detail of a description is higher when n is higher. But even though we want a good shape descriptor, a bigger n doesn't always mean better, since perfect details can also mean a descriptor becomes too sensitive too small changes and therefore less robust.

To calculate the angles we need to find all points that lie on one of our equidistant circles. When a point is found, it is added to a list that is to contain all points on that circle. To make sure the points are in the correct order they are added using the *insertion sort* algorithm, where they are sorted in a clockwise manner, with a deterministic starting point (it does not matter where the starting point is placed). The resulting list is used to calculate the corresponding radials between the points, as well as to count the number of crossings. There is a possibility that two points that lie next to each other lie on the same circle, because sometimes the lines of a shape can be thicker than one at a point where the circle crosses it. We are, however, not interested in considering this as two crossings and cause the smallest angle to be between these points. This has been solved by simply checking if two points are next to each other, i.e. whether the x or y coordinates are next to each other. In that case the number of crossings is not increased and the angle is not used.

The descriptor of the shape now consists of, for each circle, the largest angle, the smallest angle and the total number of crossings. The input for the network becomes a vector with information for each of the n circles, starting with the smallest circle. More formally it can be defined as follows:

$$D_{radial} = (max_1, min_1, cross_1, ..., max_n, min_n, cross_n) \tag{4.1}$$

Here max_i is the maximum angle for circle i, min_i the minimum angle and $cross_i$ is the number of crossings for circle i. Finally, n is again the number of equidistant circles crossing the shape. To summarize the procedure we have provided the pseudo-code in figure 4.3.

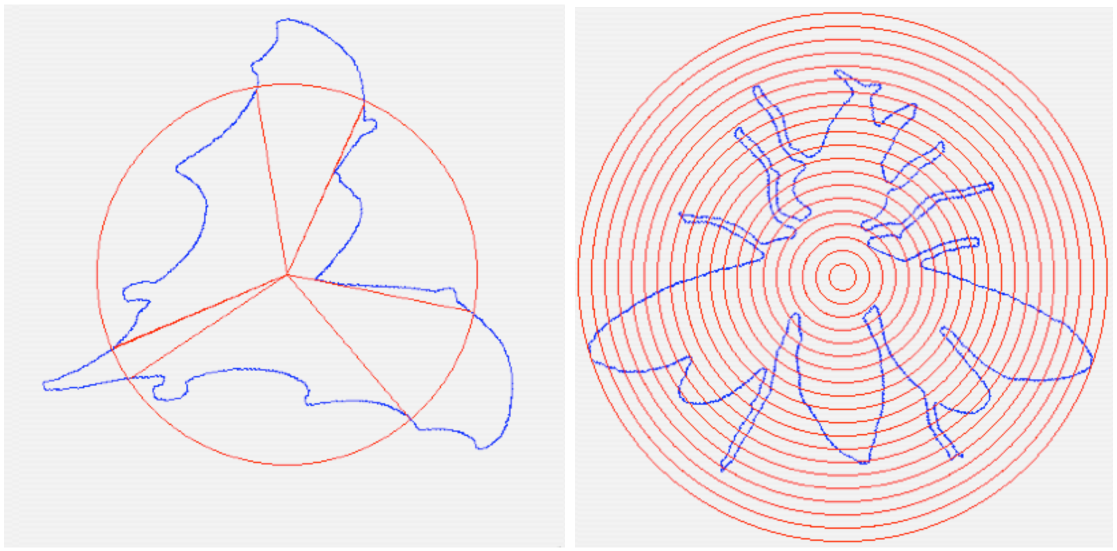


Figure 4.2: On the left a circle drawn with the angles it creates and on the right a shape with twenty equidistant circles

Radial Descriptor

```
circlePoints [] = new Vector[numberOfCircles];
for each pixel do
    if liesOnCricle(pixel) then
        circlePoints[circle(pixel)].insertionAdd(pixel);
for each circlePoints do
    calculate radials between points, count crossings
    take minimum and maximum
    add to vector minimum maximum and crossings
```

Figure 4.3: The pseudo-code for creating the Radial Descriptor

It is not difficult to see that the resulting representation is the same independent of the size, location or orientation of the shape. The more prominent question, though, is whether this kind of representation is powerful enough to distinguish between different shapes, as well as robust enough to cluster shapes together. As it holds a lot of information on the shape in a relatively small vector, it does seem to be one of our strongest ways of representing a shape.

As far as compactness and efficiency go calculating this descriptor is not without cost. It takes a full run of all points to find the point furthest from the center[2]. Another run of the shape to determine which pixels lie on which circle and inserting them with *insertion sort* would be a process that would be qualified as $O(N^2)$, where N is the number of points on the shape. Finally calculating all angles and crossings would lead to another addition of $O(N)$. The result is a complete process with a complexity of $O(N^2)$. In this case the N^2 is not just a worst case, but an extremely unlikely case, only occurring when the shape is a perfect circle. If there are, as is more usual, about five points per circle, the process will perform no worse than a general $O(N)$ complexity descriptor.

4.3.2 Distance histogram

Another, somewhat simpler, option is to look at the Euclidean distance of the points with respect to the center, like thos shown in figure 4.4. We organize the points of the shape by distributing them over an n-point histogram according to their distance from the center. The range of the distances is found by finding the largest and smallest distance from the center in the shape. Over this range n bins on a histogram are created, each bin representing one $n-th$ of this range. Now for each point on the shape the Euclidean distance is calculated and put in the bin of the range its distance falls in. After repeating this process for every point of the shape the number of points per bin determines the frequency within each bin, i.e. how many points fall into this bin in comparison with the total number of points. This is normalized to be a hundred points, in order to avoid the numbers getting too small when we use them as input for a neural network.

We now have as the input for the network a histogram, with from left to right the frequency of the smallest distances to the frequency of the largest distances. The input as a vector is then as follows:

$$D_{distance} = (f_1, f_2, ..., f_n) \tag{4.2}$$

Here f_i is the distance frequency of the $i-th$ bin of the histogram.

The implementation of the distance histogram is rather straight forward. The pseudocode can be found in figure 4.5.

Since the distribution of distances has been based on a fixed total of a hundred points there is no problem here concerning translation or size. Furthermore, rotation of the shape has no impact since the distance of a point from the center is the only factor present. Any other information on location is ignored.

[2]This is something that is needed to describe the shape later in the program as well, so does not really add any cost to the entire application

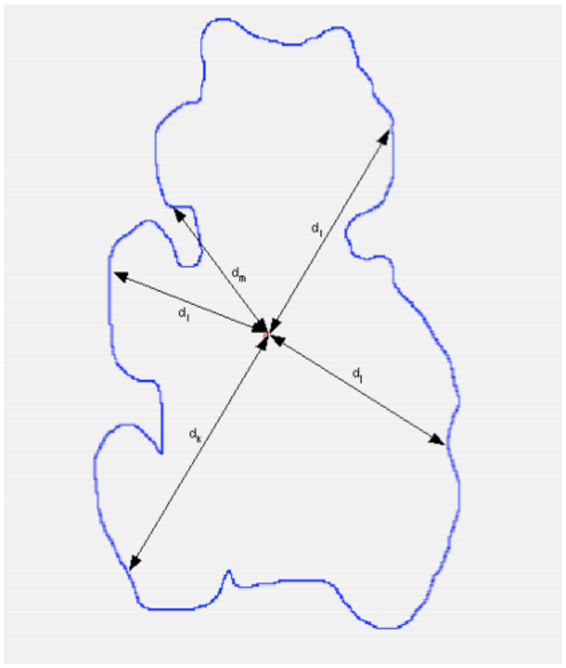


Figure 4.4: Distances

```
**Distance histogram**

centroid = shape.centroid
histo [] = new Vector[numberOfBins]
furthest = shape.furthest(centroid)
closest = shape.closest(centroid)
range = furthest - closest
for each pixel do
        distance = centroid.distance( pixel )
        histo[range/distance]++
    normalize(histo, 100)
```

Figure 4.5: The pseudo-code for creating the distance descriptor

For this descriptor efficiency in calculation and compactness are not an issue. Whether the histogram is made up of twenty or fifty bins (more would spread out the data and easily become too specific), it certainly is a compact descriptor. And with a complexity of $O(N)$ it can be calculated rather quickly. The more pertinent question is whether such a simple generalization of a shape holds enough information. When we are trying to separate cow shapes from horse shapes the answer would be: probably not. Yet, as mentioned earlier, the process of combining many shapes to become an object is based on the usage of very general shapes. Therefore, even with such a low entropy descriptor, it may be possible to represent an entire object robustly using such a simple descriptor.

4.3.3 Curvature scale space

As a last shape descriptor we have implemented the *curvature scale space* (CSS) descriptor. The curvature scale space [AMK96] is used to create a descriptor on the basis of the dents and bumps in the form of the shape. When traversing the points of a shape the curvature changes continually. At some points a shape will be convex, while continuing along the shape this might change to points that lie in a concave landscape. The CSS descriptor extracts these properties by slowly smoothing the shape and monitoring the change in curvature along the shape whilst performing this smoothing process, finally recording this information for the creation of a compact and invariant representation.

In order to smooth the shape, it is convolved using a Gaussian kernel (see appendix B). By increasing the width of the Gaussian kernel used to convolve the shape, it will become a little smoother every step (see also figure 4.8). Continuing this process will slowly lead to a shape that is completely smooth and no longer has any points where the curvature of the shape is not convex, meaning the curvature will be positive everywhere.

The information that results from this process is monitored as follows; for every widening of the Gaussian kernel all so called *zero crossings* are recorded. A zero crossing is a point on the shape where the curvature changes from being positive to negative, or the other way around. Once the shape has become fully smoothed, i.e. no more zero crossings can be detected, the process is stopped. We now have a number of zero crossings for each different level (width of the Gaussian kernel) of smoothing, corresponding to the points where the curvature changes happened. In figure 4.6 a graph is shown holding this information, making up the curvature scale space, accompanied by its shape.

Calculating the curvature of a point is normally done using the following equation.

$$\kappa = \frac{x'y'' - y'x''}{x'^2 + y'^{2^{2/3}}} \tag{4.3}$$

Here x and y stand for the function of the x and y coordinates respectively. Again the lack of a continuous function for these values causes a calculation problem. How do we find the first and second derivatives of a shape specified by 2D points rather than by a continuous shape? In [AMK96] the problem is handled by using kernels made up of derivatives of the Gaussian function. These are then used to approximate x', y', x'' and y'' in equation 4.3. These formulas have the advantage of incorporating the smoothing process, thus saving us the process of smoothing the shape for every step.

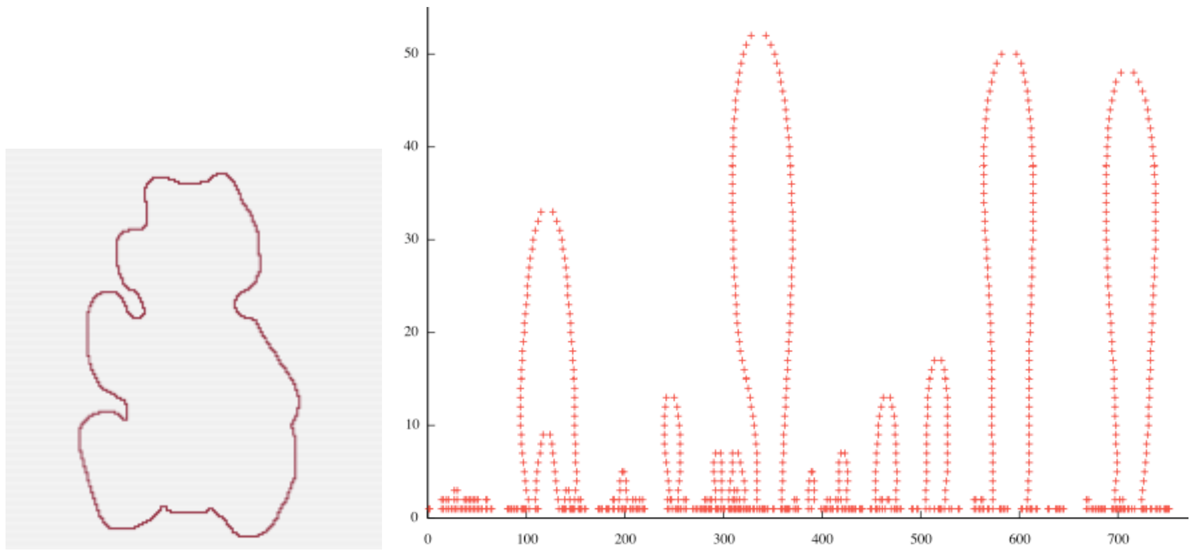


Figure 4.6: A bear shape with the corresponding CSS graph for convolving it. The horizontal axis represents the points (pixels) of the shape, the vertical axis represents the size of the deviation.

$$x' = x(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{4.4}$$

$$x'' = x(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2} \tag{4.5}$$

$$y' = y(t) * \frac{\partial g(t,\sigma)}{\partial t} \tag{4.6}$$

$$y'' = y(t) * \frac{\partial^2 g(t,\sigma)}{\partial t^2} \tag{4.7}$$

Here the '*' indicates convolution and $x(t)$ and $y(t)$ are the x and y coordinates of point t of the shape respectively. Finally $g(t,\sigma)$ stands for the Gaussian function, σ for its width, and t for the point on the shape. Using these formulas we can, given a certain σ, calculate where the zero crossings are. By continuously increasing the σ we eventually create the CSS. However, in a second step a final descriptor has to be created from the CSS keeping into account the following:

- We must find a standard starting point, to make the CSS rotation invariant.
- The CSS is not directly ready to be the input for a neural network, since more than one zero crossing can be found for the same point. We need to process the CSS.
- The CSS size is variable and should be made constant extracting the most valuable information of the CSS

```
CSS Descriptor

zerocrossings = true
while( zerocrossings ) do
    convolveShape(shape, width)
    pixel = pixels(0)
    curvaturePrevious = curvature(pixel, shape)
    for each pixel from pixels/(0) do
        point = shape.elementAt(n)
        curvatureNext = curvature(pixel, shape)
        if ( sign(curvPrevious) != sign(curvNext) ) do
            zerocrossings = false
    width.increase()
```

Figure 4.7: The pseudo-code for creating the CSS descriptor

In order to overcome these obstacles our final CSS descriptor will be created using a selection of maxima of the resulting graph as shown earlier. For a set number of ranges we will find the maximum and use the largest maximum of these to order them, by putting this one in front.

It is also not entirely clear what the level of entropy of the CSS is in comparison to, for instance, an array holding all Euclidean distances from the center on the shape. The pattern resulting from such a method seems at first sight not much less informative than the CSS descriptor, since in both the distance of a point from the center in comparison to its neighbors is at the basis of the representations. However, it must be said that the CSS descriptor has in practice proven to work well for the comparison of shapes for CBIR. But the main question here of course is how much of the power of the CSS can be translated into input for the neural network, since the process of standardization also means a loss of information.

The final CSS descriptor is as follows:

$$D_{CSS} = (max_1, max_2, ..., max_n) \tag{4.8}$$

Here max_i is the maximum within the range $[(T/n \cdot (i-1)) : (T/n \cdot i)]$ of points, with T standing for the total number of points on the shape. A summary of the general principle behind the CSS can be found in figure 4.7.

Unfortunately the CSS descriptor has a rather steep price when it comes to computational cost. As multiple convolutions should be performed, until no more zero-crossings are present, the process can easily become costly. And since each of the convolutions means applying two complex kernels for every point on the shape the process easily slows down, especially since it has to be done for every new σ.

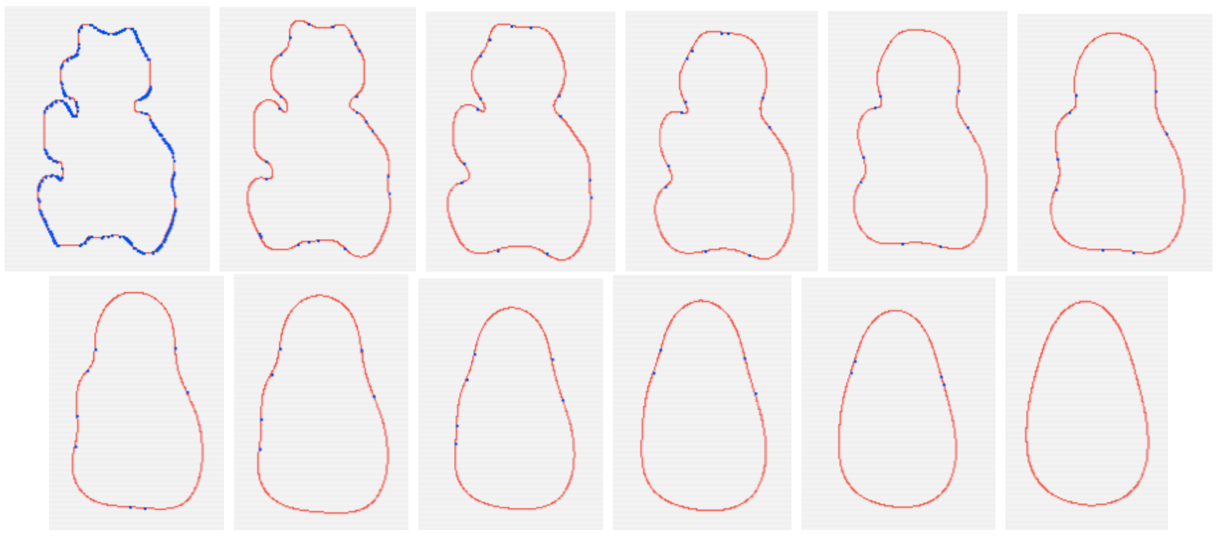


Figure 4.8: A step by step display of the shape convolving and the corresponding zero-crossings

4.3.4 Summary

To summarize this chapter we have inserted a diagram (see figure 4.9) that illustrates the process of descriptor creation. In the final implementation a choice has to be made what information is to be used from the several descriptors. Our program allows to use all the descriptors together, as well as just one in order to recognize an object.

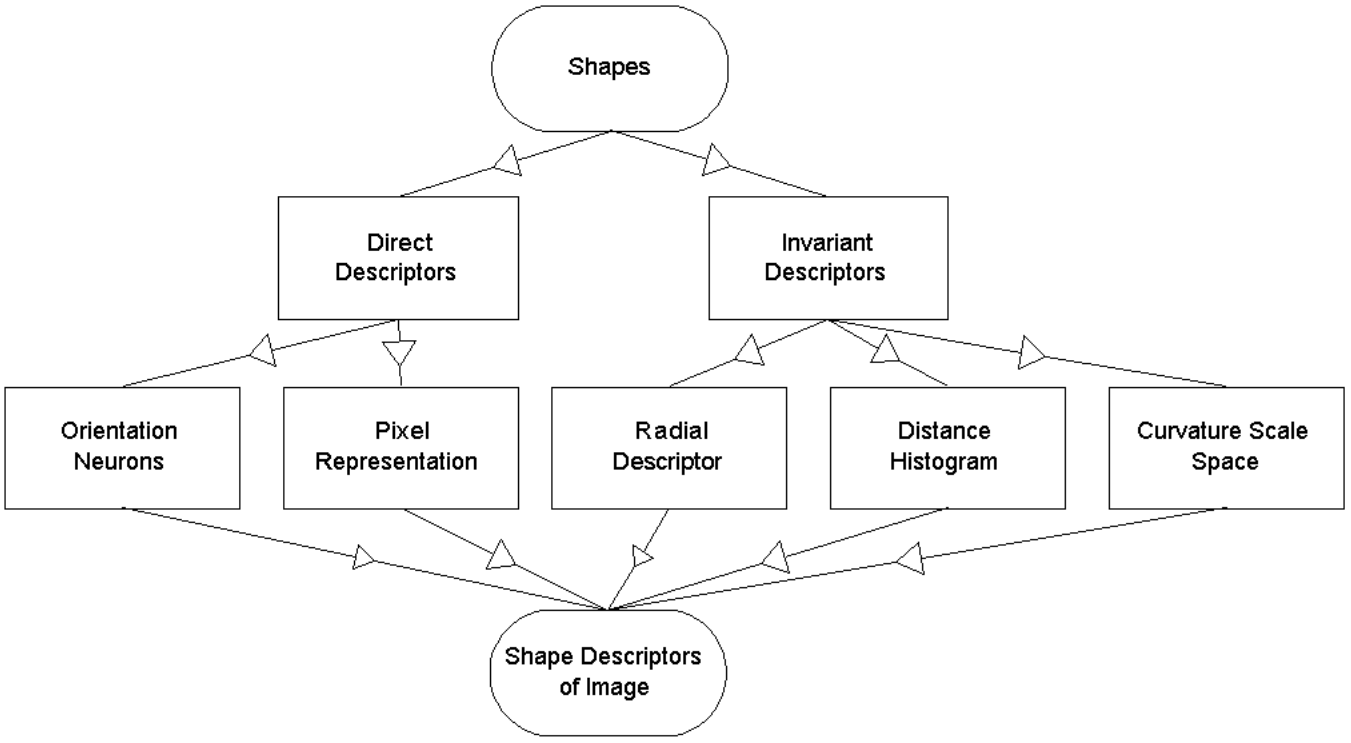


Figure 4.9: The creation of the shape descriptors

Chapter 5

Adaptive System

This chapter will describe the learning component of the application. We will give an overview of the techniques used to create the adaptive system that will recognize objects, using the previously described descriptors of the shapes in an image. Before we describe the architecture we first discuss the techniques that are used to make up the system.

5.1 Prerequisites

In our system we will use both feed-forward neural networks and a self-organizing map to achieve object recognition. We will first introduce and discuss these techniques in a more general way before we start describing the full workings of the system. The reason for this split up is two-fold. First, readers that are already more or less familiar with these techniques can skip or quickly skim over the first part. Second, because introducing these techniques on the fly can clutter the discussion of our final system.

5.1.1 Simple feed-forward or biological networks

A neural network can nowadays be anything from a simple feed-forward connectionist model to a collection of neurons that have been made to simulate, as closely as possible, an actual biological neuron, with bursting spikes, complex spike trains, etc. Inspired by new discoveries of the functioning of the brain, mainly in the cortex, there has recently been an increase in work done on these biologically inspired networks. In [Izh04] a full account can be found of the desirable properties for a neuron in such a network. The systems, however, are extremely complex to model, due to the various temporal aspects of the network (spike trains, timings, etc.), as well as the large number of properties a single neuron can hold.

Often the reason for using these biologically inspired networks is simply the fact that they are biological. The question is how beneficiary it is to model in such detail the core biological concepts, while in fact we are working with very different hardware. Many of the functions such as spike trains code in a way so that it can be interpreted by other neurons, which we believe is more likely to be a result of the way the basic elements of the brain work, rather than what gives the brain an edge over the computer. The core of the computer is, however, very different from that of the brain, and it seems illogical to implement techniques that are made to deal with a different structure, just because they are biological. This is especially true when we consider the high computational cost this requires. We should take from the brain what benefits the cause, but should not try to copy it completely.

Of course to some extent we do believe the temporal aspect of such a network can help to model some more complex concepts, since it adds another dimension to the network. For instance, the occurrence of two features at the same time can be represented by having two neurons fire simultaneously. But in our case modeling such structures would be very complex, and the large number of neurons needed takes too much computational power. For now it seems more sensible to work on the basis of computer hardware and not to make a biological structure an objective in itself. This is why we will use a standard feed-forward network, rather than a more biologically inspired system.

5.1.2 Multi-layer feed-forward network

The standard feed-forward model was first introduced in [MP43]. The model is made up of several layers, each layer consisting of a number of neurons. Each neuron has its own activation function and incoming connections from the previous layer. Each of the connections has a weight, and this weight is used to scale up or down the activation over the connection towards the neuron in the next layer. Finally an offset, or bias, is added to each neuron to give it its full representational power. Summing the incoming activations of a neuron over its connections from the previous layer and the bias give us the input of a neuron, which is then used by the activation function of the neuron to calculate its activity level. This activity is propagated forward in the same manner to the neurons in the next layer to determine the activation of the neurons in that layer. When the final layer has been reached the resulting outputs of the system are the activations of the final layer (a standard three layer version of a feed-forward network can be seen in figure 5.1). The system became popular later when the *error back propagation* (EBP) method was introduced by [RHW86] to *learn* the weights of the network, which gives the system an edge over many other learning models. In EBP forward propagation of inputs is followed by a backward propagation of the error of the outputs back through the network, pushing weights along the way slightly in the direction of giving the correct output.

Feed-forward step

As said, the input to a neuron in the feed-forward network is simply an addition of all outputs of neurons in the previous layer scaled by the weights of the connections plus the bias that corresponds to the neuron. More formally this input s given a pattern p, such as a shape descriptor, is as follows:

$$s_j^p = (\sum_i y_i w_{ij}) + \theta_j \tag{5.1}$$

Here j is the index of the neuron, w_{ij} is the weight of the incoming connection from neuron i in the previous layer, y_i is the output (activation) of neuron i and θ_j is the bias of neuron j. By performing this step for all neurons of all layers, we can calculate the output activations corresponding to the given inputs.

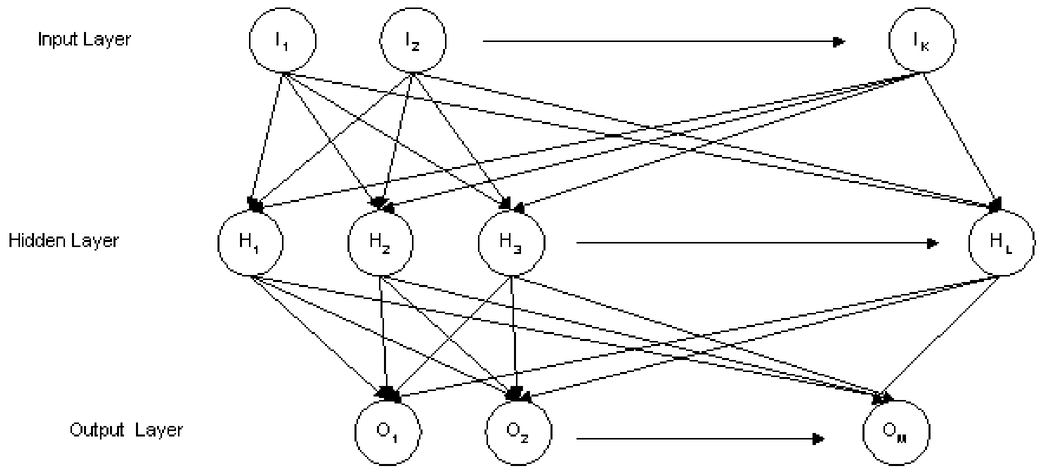


Figure 5.1: A schematic feed-forward neural network

Using an activation function the output activation of the neuron is calculated using the input in equation 5.1.

$$f(s_j^p) = \frac{1}{1 + e^{-s_j^p}} = y_j^p \tag{5.2}$$

We have used the sigmoid activation function here. This is probably the most popular activation function used in neural networks. Its popularity mainly stems from it good generalization property as well as the possibility to express the derivative in terms of the activation, which will be used later.

Error Backpropagation

Error backpropagation is a gradient descent method where the weights are updated by adding $\Delta_p w_{ij}$ at each training step for all connections. This is a function of the gradient of the error based on a small change in the weight, i.e. $\frac{\partial E_p}{\partial w_{ij}}$. The entire equation is as follows:

$$\Delta_p w_{ij} = -\lambda \frac{\partial E_p}{\partial w_{ij}} \tag{5.3}$$

Here λ is the learning-rate and E_p is the error for pattern p. The error of the network can be approximated in many different ways. We use the following function which takes the square of every difference between the desired output and the actual output for a neuron.

$$E_p = 1/2 \sum_{o=1}^{N_o} (d_o^p - y_o^p)^2 \tag{5.4}$$

Here d_o^p is the desired output for output neuron o given the input of pattern p, and y_o^p the actual output for the output neuron.

Using the chain-rule we can rewrite the last part of equation 5.3 as follows

$$\frac{\partial E^p}{\partial w_{ij}} = \frac{\partial E^p}{\partial y_j^p} \frac{\partial y_j^p}{\partial s_j^p} \frac{\partial s_j^p}{\partial w_{ij}} \tag{5.5}$$

We now split up equation 5.5 and handle each partial derivative separately. We will start with the last two terms, as they are independent of the location of the neuron. Using equation 5.1 we obtain:

$$\frac{\partial s_j^p}{\partial w_{ij}} = y_i \tag{5.6}$$

The next term is the derivative of the sigmoid function:

$$\frac{\partial y_o^p}{\partial s_o^p} = \frac{\partial}{\partial s_o^p} \frac{1}{1+e^{-s_o^p}} \tag{5.7}$$

From equation 5.2 we know that

$$f(s_j^p) = (1+e^{-s_j^p})^{-1} \tag{5.8}$$

The derivative can now be calculated as follows:

$$\begin{aligned} \frac{\partial y_j^p}{\partial s_j^p} &= -(1+e^{-s_j^p})^{-2}(-e^{-s_j^p}) \\ &= \frac{1}{1+e^{-s_j^p}} \frac{e^{-s_j^p}}{1+e^{-s_j^p}} \\ &= y_j^p(1-y_j^p) \end{aligned} \tag{5.9}$$

The possibility to express the derivative of the sigmoid as a function of its activation is one of the properties that makes the sigmoid activated neuron so popular.

The last partial derivative, $\frac{\partial E^P}{\partial y_j^p}$, that accounts for the error does depend on the location of the neuron. Here two cases have to be considered, one where j represents an output neuron, and one where j represents a hidden neuron.

For the output neuron the equation is rather simple.

$$\frac{\partial E^p}{\partial y_o^p} = 2 \cdot 1/2(d_o^p - y_o^p) \cdot -1 = (y_o^p - d_o^p) \tag{5.10}$$

Using equations 5.6, 5.9 and 5.10 we can now rewrite 5.5 to calculate the weight update for the connections between the hidden and the output nodes as follows:

$$\Delta_p w_{ho} = -\lambda(y_o^p - d_o^p)y_o^p(1-y_o^p)y_h \tag{5.11}$$

For the connection to the hidden units the error does not directly affect the neuron, but is contributed by each output neuron it connects to. Using the chain rule we can write out this gradient as follows:

$$\frac{\partial E^p}{\partial y_h^p} = \sum_{o=1}^{N^o} \frac{\partial E^p}{\partial s_o^p} \frac{\partial s_o^p}{\partial y_h^p} = \sum_{o=1}^{N^o} \frac{\partial E^p}{\partial s_o^p} \frac{\partial}{\partial y_h^p} \sum_{h=1}^{N_h} w_{ho} y_h^p = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial s_o^p} w_{ho} \tag{5.12}$$

Using equation 5.9 and 5.10 this can be rewritten as:

$$\frac{\partial E^p}{\partial y_h^p} = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial y_o^p} \frac{\partial y_o^p}{\partial s_o^p} w_{ho} = \sum_{o=1}^{N_o} (y_o^p - d_o^p) y_o^p (1 - y_o^p) w_{ho} \tag{5.13}$$

The update of weights for the hidden neuron now is as follows:

$$\Delta_p w_{ih} = -\lambda \left(\sum_{o=1}^{N_o} (y_o^p - d_o^p) y_o^p (1 - y_o^p) w_{ho}\right) y_h^p (1 - y_h^p) y_i^p \tag{5.14}$$

The bias can be found in a similar fashion, by treating it as an incoming connection just as in the other cases. Here the bias gets an activation value and the value of the bias is treated as the connection.

Number of hidden units

The number of hidden units is always a problem with neural networks. Finding good material on how many hidden units to use is very difficult, since most of the research is based on experience and testing, rather than on logical or mathematical reasoning. Many articles describe an artificial task and then approximate the number of hidden neurons by testing iteratively and changing the number of neurons in each step. No good method, not even a good heuristic is presently available to estimate the number of hidden neurons needed in advance. Therefore, in this document we will also rely on testing to find the best number of neurons.

Neuron activation

We use the sigmoid activation (shown in figure 5.2) for the neurons because it has some very favorable properties. Looking at equation 5.9 we can see that its derivative can be expressed in terms of its input, which allows for fast calculation. Furthermore it has the nice property of quickly generalizing its inputs to either zero or one. The generalization also has a drawback, however: if the neuron is stuck in one of its extremes, on the basis of the inputs it is hard to *unlearn* this generalization. Because the slope of the sigmoid is continuously decreasing towards the extremes at either side, the derivative at these ends is small as well. Since the derivative is a factor in the weight update, being stuck in one of the extremes will mean weight updates are small, making it hard to escape these extremes.

Weights

Weights are initiated at random in a range of $-x$ to x. Even though the average over many incoming weights should be zero, especially with not too many neurons this is not necessarily the case. This is why it is important not to initiate the weights too high, since this will easily put the network in a position where one or more neurons will be at one of the extremes. And as explained in the previous paragraph this can lead to problems that can severely impair the learning process. Generally we will therefore initialize the weights between -0.5 and 0.5.

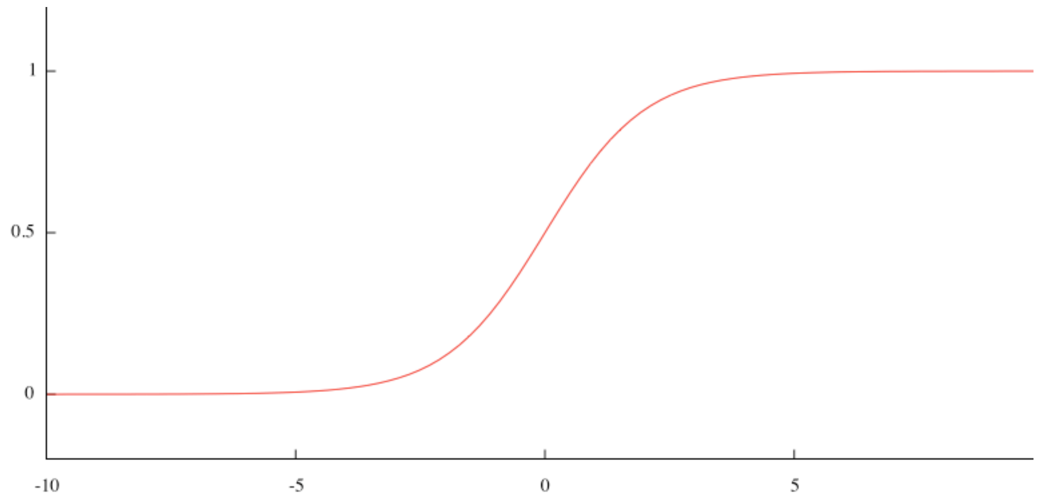


Figure 5.2: The sigmoid function

Bias

Finally each neuron has an incoming bias. This offset is necessary to give the neural network its full range of representational power. The importance of the bias can be made clear with a simple example. Imagine the need for an output of any non-zero number and an input activation of zero. Without a bias just multiplying these inputs and propagating them forward will never lead to the desired output.

5.1.3 Self-organizing map

The self-organizing map (SOM) [Koh01], also known as Kohonen map after the person who first introduced it, is a field of neurons that allows us to map vectors of high dimensionality onto a field of lower dimension, most often to a one-dimensional or two-dimensional field (we will use a 2D version). After having been trained, the map is organized in such a way that values that lie close together are more similar than those farther apart. In other words, similar vectors group together. The SOM is trained using unsupervised learning, on the basis of selecting desired outputs by proximity to the input, rather than supervised learning like in the feed-forward network. The process is similar to the *winner takes all* method.

Training the SOM

The basic process is to first initialize the 2D map with a vector for each point on that map that holds a number of randomly initialized values. The number of values should be equal to the number of values in each training sample and should be initialized in the same range. This is then followed by iteratively taking a random sample of the desired training set, finding the *best matching unit* (BMU) on the map, and pulling this point, as well as to a certain degree its neighbors, closer to the input vector.

The BMU is found by simply calculating the Euclidean distance between the weights of the vectors on the points of the map and the input vector. The point, m_{BMU}, on the map with the smallest Euclidean distance from the training pattern, x_p, is then considered the BMU. Or more formally:

$$\forall_{i \in N}(\|x_p - m_{BMU}\| \leq \|x_p - m_i\|) \tag{5.15}$$

Here m_i is the i-th point on the map and N is the total number of map points. After the BMU has been located the BMU and its neighbors have the weights, the values in the vector, pulled towards the values of the input vector. This change in weight is calculated as follows:

$$m_i(t+1) = m_i(t) + \alpha(t)\aleph(t,i)[x(t) - m_i(t)] \tag{5.16}$$

Here m_i is any node found on the map, $\alpha(t)$ is a decreasing function for the learning-rate, and the $\aleph(t)$ is known as the neighborhood function. We will now take a closer look at these two functions.

The neighborhood function determines to what extent points on the map are in the neighborhood of the BMU and thus what their share in the update should be. As the training continues we want to make the selection of neighborhood nodes stricter, to give the SOM a chance to evolve towards more specificity. This is why the neighborhood function takes the time-stamp, t, as a parameter.

The neighborhood function is now shaped as follows:

$$\aleph(t,i) = e^{\frac{-||r_i - r_{BMU}||^2}{\sigma(t)^2}} \tag{5.17}$$

Here r_i and r_{BMU} are the coordinates for the BMU node and node i respectively. This function, that satisfies the properties of a Gaussian function, is ideal as a neighborhood function. It takes nodes close to the BMU with a large factor, but slopes quite steeply downwards as we get closer to the deviation value σ and converges towards zero. As we decrease the value of the σ the slope becomes steeper making the neighborhood smaller. In this way the desired effect for nodes becoming more specifically tuned is achieved.

Several solutions have been proposed for both the σ and the α decay functions. We use the functions in [RMS91]. These functions are concave and slowly decreasing in slope. We prefer them over a linear decay function since this would result in a much more convex learning-rate function over time, due to the usage of a Gaussian function to determine neighborhood values.

The σ, the deviation of this Gaussian function, is a time decreasing function, that slowly narrows the width of the kernel, and so allows the map to specify better. We have chosen the following function for the σ:

$$\sigma(t) = \sigma_i(\sigma_f/\sigma_i)^{t/t_{max}} \tag{5.18}$$

Here σ_i is the initial value of the σ and σ_f is its final value. Furthermore t_{max} is the value of the last iteration. Since the neighborhood of a node is a relative concept we made σ_i dependent on the size of the SOM, setting it equal to the width of the map. The final value, σ_f, is chosen to be 2/3 finishing the process with a narrow Gaussian function that causes the weight update of the direct neighbors to be no higher than a third of that of the

BMU weight update. As an extra factor to enhance the process as it updates the map, the learning-rate itself is also a time decaying function, and also takes time as a parameter. We want to slowly decrease the update values and keep the algorithm from rigorously pulling the weights, in order to allow them to settle at certain values. For this purpose the following function has been chosen:

$$\alpha(t) = \alpha_i(\alpha_f/\alpha_i)^{t/t_{max}} \tag{5.19}$$

Here t is as always the time-stamp, and the α values have the same meaning as the earlier described σ values. The initial value is set to 0.25 and the final value to 0.05.

This process is repeated for a fixed number of iterations. Since this method is an unsupervised technique it is hard to decide on a good stopping criterion, and it is quite difficult making up criteria without knowing the complexity of the data. When the map is fully trained we can use it as a classifier by finding the BMU for inputs, and considering the input as being represented by the point of the BMU on the map.

The complexity of finding matching nodes on the SOM is $O(N)$. This is relatively steep, since for a SOM to have a large range of expressional power we want it to be rather large. Yet for the present study we have accepted the limitations to the size of the SOM. To keep the number of calculations for the training of the weights to a minimum we have decided not to recalculate all values for every single iteration. In a series of thousands of iterations these values do not change much from one iteration to another. Therefore, to save computational power, we only recalculate the learning-rate and neighborhood values every twenty iterations.

5.2 System architecture

Now that we have introduced the neural techniques and devised the part of the program that will process input images all the way to separate sets of shape descriptors, we are ready to introduce the learning component of the program. Basically our adaptive system will consist of three components. The first component will interpret shapes, even further reducing the dimensionality of each shape of the object. The second component recombines these shapes by mapping combinations of different shapes to a field. And the last component is a network that interprets this field and will classify the shape. The entire program is shown in figure 1.2. Below we will discuss each of these components in turn.

5.2.1 Shape interpretation network

The first component, which we will refer to as the *shape interpretation network*, is a basic three-layer neural network as described above in section 5.1.2. Its main purpose is to recognize and classify shapes in such a way that they can be used to form the basis for object recognition. Having a set of descriptor vectors of which each is about a hundred values long is a difficult basis for object recognition. This is why the interpretation network should facilitate a reduction in this dimension, by somehow lowering the number of values to a short indication or description of the shape. The network will therefore take the descriptor of a shape as an input and output the classification of the shape.

For training this network it would be a possibility to indicate in every object what local shape is equal to a shape from another identical object. This would mean including a process that is not just tedious, but also involves a large amount of manual labour making the process error prone. Furthermore, it would lead to a great number of possible shapes, while we would like to reduce this number as far as possible. So we needed to come up with a way of training that does not require knowledge about the shapes we are dealing with. Direct training is therefore unfortunately not possible.

For this indirect training we have considered several methods, from using randomly initialized networks, to unsupervised learning, to supervised learning. A first option is to use evolutionary techniques to find the best network. This option would be slightly preferred over random initialization, because it would work with several random networks and will enable us to retrieve the best of all their properties. Evolutionary updating would mean having a large population of adaptive systems (n) and training all of them for each generation. Since the process would take several iterations (i) this would quickly lead to training a large number of systems ($n + n \cdot i \cdot p$, where p is the offspring percentage) and because training one is already a costly process, this technique might give us good results, but it is not practical.[1]

A second option is to set the weights of the network randomly and leave them fixed. Although this process can lead to very different results, as soon as a network performs well it can be stored and used in all future cases. This approach might sound somewhat arbitrary, but is certainly not without potential. It has been shown to work in other techniques like for instance the *echo state network* as described in [Jae01], a recurrent neural network that works on a random basis. Also in [SH96] the power of random initialization has been shown, by comparing iterative random initialization to general learning methods.

Even though a random network could provide a good result, it should be possible to find a solution actually specified for the problem that performs even better. To do this we came up with an approach similar to the one in [VV99], where a CBIR application is created on the basis of indexing shapes by comparing them to a set of *vantage shapes*. This provides a basis for retrieving similar shapes, indexed close together. The dictionary describes the word *vantage* as *the quality of having a superior or more favorable position*. Our problem now is to find such vantage shapes that can help us to classify our shapes.

Two approaches for selecting vantage shapes were considered. One approach was to select shapes that are well distinguishable and clearly satisfy the idea of being different, not just on the basis of human perception, but also with respect to the descriptors used. Another option was to come up with a set of primitive properties that are inherent to all shapes, classify a large set of many different shapes to these properties, and train the network to learn to indicate the presence or absence of these properties.

At first sight the second approach seems the most attractive. However, how do we come up with these primitive properties, and the shapes to match them? Using the MPEG-7 test database [LLE00] to attempt to come up with properties and indexing shapes, left us with sets of images that satisfied questionable properties and with an unequal amount of shapes representing the different properties. This is why we opted for the first approach, which

[1]The option has been included in the program, but has not been tested due to its extremely high computational cost

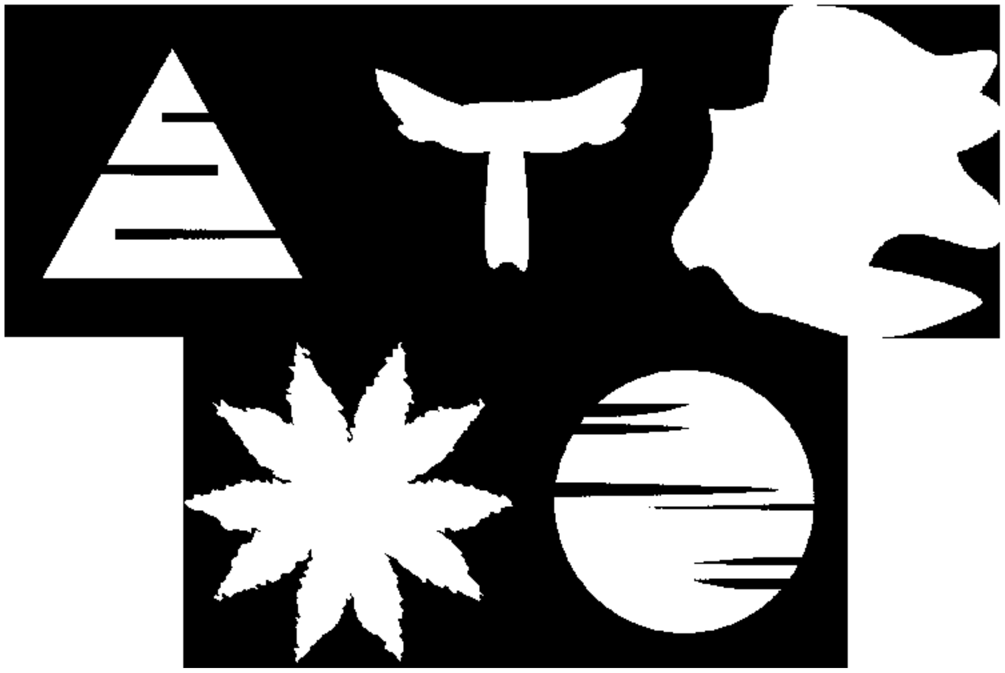


Figure 5.3: Examples of the five vantage shapes used in training the shape interpretation network

was much easier to do. We selected five different shapes from the MPEG-7 test database, which we used for training and testing this component. The shapes, each represented by twenty different versions, were selected as they are clearly distinguishable. An example of each shape is given in figure 5.3. After having trained the network with one output neuron corresponding to each shape, it can be used to classify the shapes, generalizing them to a vector of five values between zero and one.

The result of the interpretation network is now an output vector that holds an, admittedly crude, approximation of each vantage shape, indicating to what extent the input shape is similar to each of the five different vantage shapes. Of course this is not an official distance measure. The idea is that the activations in the hidden layer should represent the *features* of the vantage shapes and that the absence of features will lead to a decrease in similarity. By using the neurons with a sigmoid activation function that are stretched the outputs indicating the similarity should decrease and increase more gradually than when using the original sigmoid.

The sigmoid neuron stretched

As mentioned the sigmoid activated neuron is very good for generalization. This also means that in a case where we want less generalization we can consider to adapt the activation function. Since we do not want our shape interpretation network to generalize too much, but rather to give us a good approximation of what the shape is like, the activation function for this network's neurons is stretched. The idea behind stretching these neurons is that the sigmoid neuron derives its ability to generalize from its steep slope in the middle and its convergence to zero and one in the limits (negative infinity and positive infinity resp.). To undo some of this generalization we have simply stretched the function by diminishing the input by a factor of four.

The change in the earlier shown equations is minimal as we can see in the new activation function:

$$f(s^p) = \frac{1}{1 + e^{-(s^p/4)}} \tag{5.20}$$

This can be rewritten in the same way as equation 5.2, to again express the derivative of the activation in terms of the activation:

$$\begin{aligned}\frac{\partial y^p}{\partial s^p} &= -(1 + e^{-s^p/4})^{-2}(-e^{-s^p/4})(\frac{1}{4}) \\ &= (\frac{1}{4})\frac{1}{1 + e^{-s^p/4}}\frac{e^{-s^p/4}}{1 + e^{-s^p/4}} \\ &= (\frac{1}{4})y^p(1 - y^p) \end{aligned} \tag{5.21}$$

Component quality

Even though for objects we can test the created interpretation network only indirectly, for shapes we can test it directly as well. And since this is at the basis of our recognition system we feel it is important to test this thoroughly. In the next chapter, where we will test our system, we will start with a test of the ability of such a neural network to classify shapes. This will give us a better idea of the potential of a neural network as a shape identifier.

5.2.2 Shape combination map

Even though we have now further reduced the dimension of the shapes descriptor, the descriptors still need to be combined to form the object's representation. Of course one possibility, again, is to supervise this process and to create this knowledge in advance. The objective of the present system, though, is to create a system that is as autonomous and adaptable as possible. This excludes the actual labeling and classification of shapes that make up an object in advance. Furthermore, the adaptability of the system would be compromised by this process, since learning new shapes would also require these extra steps of categorizing shapes. From this point of view it is necessary to create a system that is able to organize the shapes by itself. This is a rather difficult task, since interpretations of shapes, that are in no way limited to discrete values, can take an infinite amount of gradations. For a neural network to interpret such a cluster of shapes, the cluster needs to be presented in such a way that all objects out of one class are similar.

One way to do this is to present the network with a selection of these shapes, ordered according to a specified comparison measure. This, however, will give us no certainty that the shapes will be presented to the network in the same manner every time. Shapes can be missing or, since there is no perfect comparison measure, can be ordered differently. Furthermore, one object can be represented by two shapes, while another can be represented

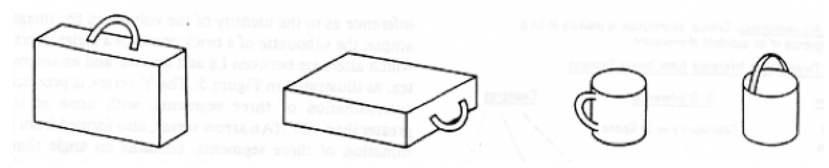


Figure 5.4: Two examples of how how the same elements combined differently can represent different objects. *Source* [Bie87]

by many, which is also hard to model. Therefore we have inserted a structure that is stable in mapping shapes, no matter what their order of appearance is, whether there are missing shapes etc. For this we have included a SOM that will represent the objects.

We started out by mapping these shapes, using the five-valued vectors, directly to the SOM. This meant loosing relational information we have on the shapes, namely size differences, distances between shapes and orientation differences. Since this information can be quite important (an example is given in 5.4) to determine which object is presented we have explored options to include this information. To do this we decided to create a vector containing the combination of the information of two shapes rather than just the information of a single shape. We created this vector by concatenating the output for both shapes, adding additional relational information at the end of the vector. This information is three-fold. First there is the difference in orientation between the two shapes. Second there is the size ratio (the size of the first shape divided by the size of the second shape). And finally there is the distance between the two shapes, which we will express as a fraction of the largest distance between any two shapes in the object.

The final vector is given in the following equation:

$$combi_{ij} = (m_1^i, m_2^i, m_3^i, m_4^i, m_5^i, m_1^j, m_2^j, m_3^j, m_4^j, m_5^j, o_{ij}, r_{ij}, d_{ij}) \tag{5.22}$$

Here i and j are the first and second shape respectively and m_x^s is the output (match) for the node trained on vantage shape x given shape s as input. Further, o_{ij} is the orientation difference in radials, r_{ij} is the length of shape i divided by shape j, and d_{ij} is the distance of the two shape centroids divided by the distance between the two shape centroids lying apart the farthest. To avoid having two different combinations for every shape pair, the shapes are ordered so that the smallest one (in terms of length) is the first one, and the biggest one the second shape.

The vector above is calculated for every possible shape combination in each object. After the SOM has been trained using shape combinations from the training data of several objects, the SOM can then be used to map the shape information of an object.

Mapping a thirteen-dimensional vector to a two-dimensional field seems like a rather steep reduction in dimension. How well can this be done? This question can be looked at both from a general and from a more specific point of view. The more general version of the question is one of the possibility of mapping thirteen dimensions to two. As a general reduction of thirteen dimensions to two will lead to a loss of information if all the dimensions are used to their full ranges, a lengthy discussion can be held on the ways to salvage some of the lost information. The specific version of the question is whether the 2D field is sufficient to hold all of the necessary shape combinations. Can the available information be mapped to a 2D field?

In an earlier discussion of Biederman's work, namely the recognition by components theory, it has already been mentioned that the number of combinations of geons is extremely large. When we use real values and relational information for our shape the number of possibilities gets even bigger. This rules out having a point for each possible combination. Fortunately, we do not really need a point for each combination. Only those combinations that are present in the objects have to be represented. This still leaves us with a large amount of combinations. Taking into account that the SOM is a computationally expensive mechanism we probably can not even afford to map all those combinations. However, varying the dimension of the SOM should lead us eventually to a map that is large enough to hold most combinations, while generalizing those that can be considered similar. The size of the SOM should lead to a balance of representing different combinations and generalizing over some to create a landscape that is sufficient to distinguish between the objects.

The points on the SOM will not always give a perfect representation. Therefore as an input to the last network we do not only take the points corresponding to the shape combinations. Rather, we extend the activation of the winning units to its direct neighbors (the similar units) on the map. We achieve this by giving a large activation to the BMU, while also activating its neighbors slightly. Since there is a possibility of having several inputs for the same point we allow the activation of an input neuron to increase by adding to the activation of the input neuron, rather then activating it once. In this way a three-dimensional, rather than two-dimensional, landscape appears that represents the shape combinations present in the object.

5.2.3 The object classification network

Now that we have the possibility to map these shape combinations to a field, we still have to interpret these combinations and classify them as belonging to a particular object class. In this section we will discuss the final addition, completing our system.

The last step in finding the object is to use the 3D landscape on the SOM to classify the object. We will again use a neural network that will learn the objects corresponding to the field. In order to be able to distinguish it from the earlier described shape interpretation network, we will refer to this network as the *object classification network*. The network is initialized with an input neuron for each node on the SOM. In the feed-forward step of the object network every neuron on the input layer now has an activity level that is equal to the build up on the corresponding node of the SOM. The pattern of the SOM is thus translated to the input layer of the object network and with a feed-forward movement the outputs of the network are calculated. We then use EBP once more to train the connections of the network. For this neural network we will return to the original sigmoid function, as we want to return to the high level of generalization.

If the SOM resulted in a perfect representation a full neural network would not be necessary to classify the object. More solid probabilistic measures could be used. As the noise on the SOM will be rather large and the patterns vary quite extensively we use another neural network to perform the final readout.

Output classification

The last layer should represent the classification of the object, meaning that all output neurons somehow need to code different objects. An efficient way to do this would be to do a binary coding, which is well possible since all neurons generalize to zero or one. This would mean that with just seven output neurons we could classify over a hundred shapes or objects. However since we do not have any a priori knowledge on the inputs we would like the distances between all of the different desired outputs to be equal. This is why we have chosen to make all of the desired outputs orthogonal by letting each output neuron represent its own shape or object. Desired outputs will now be a vector of zeros, apart from the one for the place that corresponds to the input pattern, which will be one.

A problem of trying to train a network having a considerable number of output neurons is that almost always the desired output for a neuron will be zero. Since we perform online learning, this can easily lead to the convergence of the system to giving a zero output for every neuron on every input, since it gets pushed in that direction most of the time. Especially with many objects and a randomized selection of the next input during learning it can take quite a while for a specific object to be selected. This would lead to a continuous reinforcement towards a zero output. Also in a sigmoid activated system it can be hard to escape from these extremes, especially since the slope in these extremes is very small, resulting in very small weight updates. It is something that is hard to be unlearned. To remedy this shortcoming the weight update concerned will be given an extra scaling factor, in an attempt to correct for this effect. This factor is varied along with the number of output units. The error of the output neuron that should be the winner is doubled from the normal standard in contrast to the other neurons where the error is downscaled considerably by multiplying the error also by two but then divided by the total number of outputs.

5.2.4 Complexity analysis

Computational cost are a central issue when working with images. The huge amount of pixels a general image is composed of often forms a barrier when working with these images. And in trying to make sense of an image it is very tempting to perform several operations that each run over the entire image every time. When using pixel descriptors this complexity is propagated through to the neural network, making the entire system extremely slow, especially during the learning phase. The learning phase only has to be performed once and does not have to be extremely quick. But with recognition, the learning phase for a pixel based approach can easily take so much time that it is no longer practical.

With the growth of the system also the complexity has grown. The mapping of a neuron to the SOM already takes $O(n)$, with n being the number of nodes on the SOM, in case it has to be done for every shape combination $\binom{m}{2}$, where m is the number of shapes, this becomes a complexity of $O(n\binom{m}{2})$ just for mapping the shapes to the SOM. Adding to this the time taken to process information through both the shape network and the object network, means that the complexity of the system easily exceeds acceptable limits. The general feed-forward step of a neural network comes at the expense of the total number of connections in the network. For a fully connected network this means $O(ih + ho)$, where i stands for the number of input neurons, h for the number of hidden neurons, and o for the number of output neurons. In the first network this means the i will be the size of the descriptor and the o will have a value of five. For the last network the i will be the size of the SOM and the o will be the number of objects to be learned.

In the end, however, in the adaptive system the complexity of the mapping to the SOM is by far the largest. The object interpretation network (which takes the field of SOM as its input) and the shape interpretation network are relatively quick.

Still, the processes just described does not constitute the most costly part of the whole system. The preprocessing of the input image as well as the feature extraction and descriptor creation form the actual bottleneck of the program. They are also the part that is the hardest to speed up. In future implementations neural techniques can perhaps be enhanced. An attempt could be made, for instance, to use the earlier mentioned TS-SOM that would save time.

5.3 Learning and Classifying

Now that we have introduced the system, we will finish by explaining the training and recognition procedure.

Training

First the shape interpretation network is trained using the five vantage shapes that are shown in figure 5.3. The network is trained using the twenty different samples per shape. Iterations are stopped once eighty percent of the outputs are predicted correctly, i.e. the neuron with the highest output corresponds to the input shape. In this way the network is moderately trained, in order not to over-fit, or become too generalizing.

In a second step all the shapes of the objects in the training set are used to make pairs as described earlier. With these combinations the SOM is trained[2]. To take shape this will be run for a number of iterations, taking out random combinations, with a slowly decaying learning rate.

Once these components have been prepared we can train the system to recognize the objects. For this purpose, random samples have their shapes extracted and interpreted by the interpretation network. Next, these interpretations are combined and give shape to the SOM. Finally the last network is used to calculate the final outputs. To train the last network, after every step the EBP is used to update the object classification network. After the network converges the system has been trained and can now be used to recognize new images containing learned objects.

[2]To speed up the process, the outputs of the entire data set are computed directly after the SOM has been trained. These outputs are used to determine the BMUs on the SOM. The BMUs are then stored with the object data, so that they can be retrieved when training and testing the system. This approach saves large amounts of time. Determining the shapes and mapping their combinations to the SOM is a costly process and this process is now only performed once for each shape. This means the process is equivalent to one epoch. In contrast to being calculated for each epoch the system is trained over.

Recognition

The recognition process is now a simple process. First all shape descriptors are interpreted by the first network. After having collected all shape interpretations, combinations are created according to equation 5.22. These vectors are then mapped onto the SOM, increasing the activations of the neurons that make up the SOM. And finally the object is classified using the object recognition network.

5.4 Summary

In conclusion our final system is depicted in figure 5.5, where the various components of the system are bundled together.

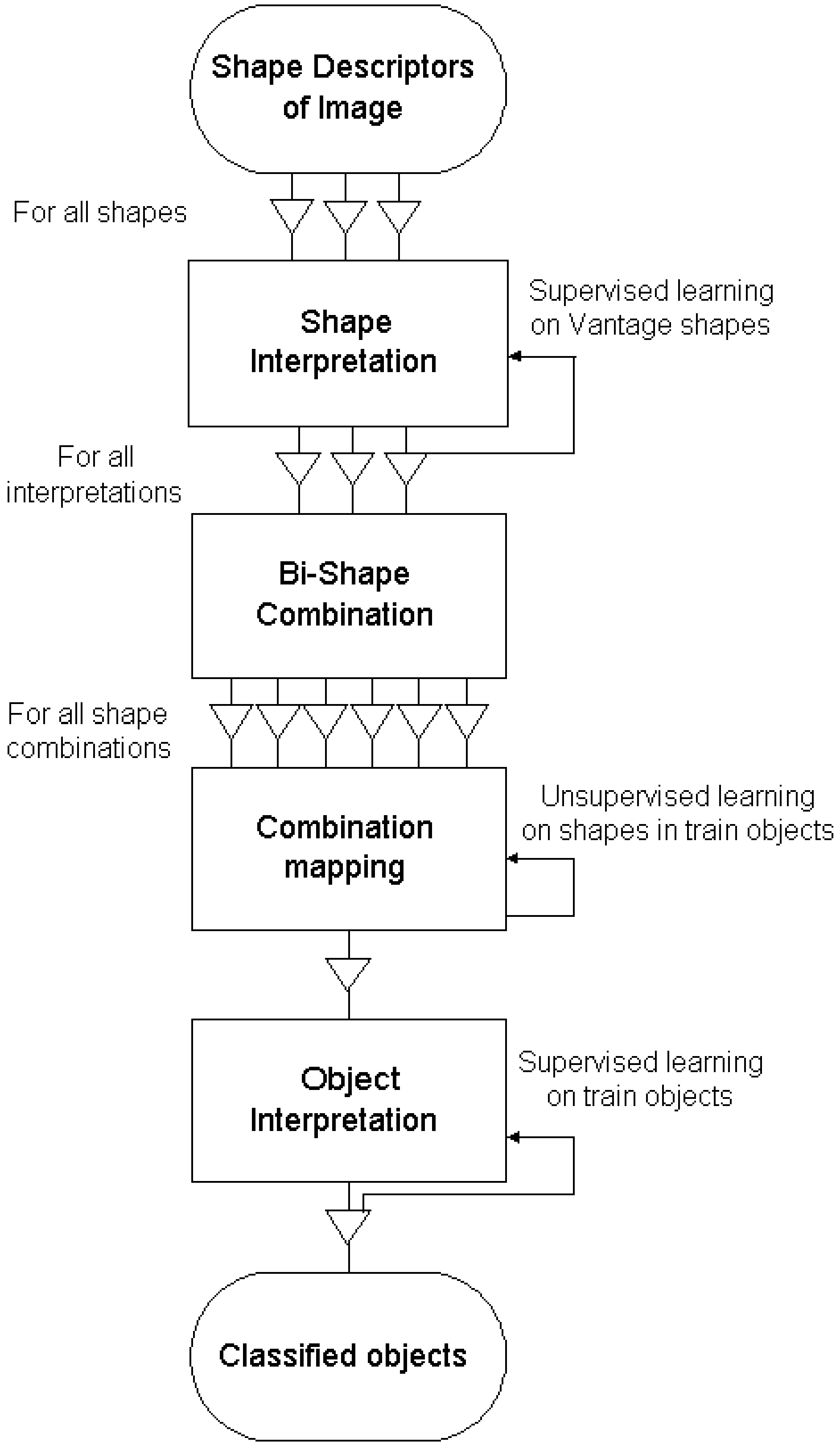

Shape Descriptors of Image
For all shapes
Shape Interpretation
Supervised learning on Vantage shapes
For all interpretations
Bi-Shape Combination
For all shape combinations
Combination mapping
Unsupervised learning on shapes in train objects
Object Interpretation
Supervised learning on train objects
Classified objects

Figure 5.5: The final adaptive system

Chapter 6

Testing

We have performed many experiments to determine the quality of the system as well as of its individual components. In this chapter we will summarize them in three parts, corresponding to the three test sets we used. The first test will look at the performance of the shape interpretation network in combination with the shape descriptors. In order to obtain the best possible evaluation of the quality of shape descriptors and the interpretation network, this test focuses on simple shapes instead of objects.

The following two tests evaluate the ability of our system to recognize real objects. The first of these is a test using 2D objects. We will perform this test to evaluate the quality of the system when it comes to the interpretation and recombination of shapes. The last test will show us whether our system can cope with 3D objects that can be viewed from any angle. This test is included to see whether our system can generalize over a broader set of different inputs and can work with real 3D objects. The description of the tests will be preceeded by a brief description of the methods used to test our programs.

6.1 Test methods

To get a good approximation of the quality of the adaptive system without having to perform too many tests, we have used N-fold cross validation. Tests are performed by partitioning the data set into equally large subsets that are mutually exclusive. During training and testing each subset will be used once to test the system. This approach ensures that every sample in a data set will function as the subject of a test at least once. The final result is an average of the results over the different test sets. This will mean that if we decide to use twenty percent of the data for testing (as we will do in all of our tests), we have to perform the same test five times, each time using a different test set (and the complement of the data set for training).

In testing we would of course like to vary parameters as much as possible. This, however, is constrained by the explosive character of the costs of trying all kinds of different combinations. Also, it will easily lead to an information overflow of numbers, tables and graphs in this section. Therefore, we have selected the variation of parameters on the basis of our experience of random tests run while creating the system. We will focus on the tests that give significant results.

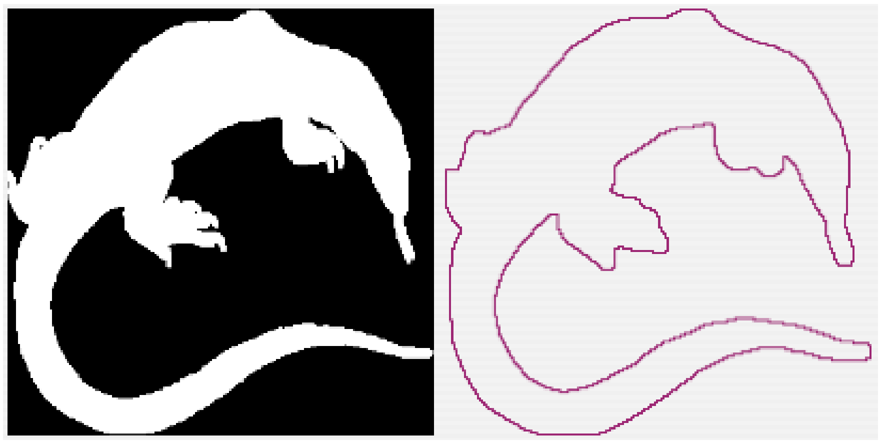


Figure 6.1: Only the outer contour of the shape is used for recognition

In order to distribute the samples of the data set over the different test sets, a data block for each test set is created. Loading the data of a particular shape or object, samples are divided over these blocks by adding every newly loaded sample to a different block, rather than adding the first batch to the first block, the second batch to the second block, and so on. In this way data will more likely be uniformly distributed.

6.2 Shape recognition

We will start by testing our shape interpretation network. This gives us both a way to test whether the neural network can recognize shapes, and a way to test the quality of our shape descriptors.

We use the MPEG-7 test database [LLE00] which contains 20 GIF images of 70 different shapes each. The images are drawings in simple black and white, where the white is the shape and the black the background. Using the earlier described preprocessing we extract the shape from the image. For this test we only use the outer contour of the drawing, instead of all shapes that can be retrieved from the image (see figure 6.1[1]). Unlike in the full system we have, in this network, used normal sigmoid activated neurons instead of stretched out sigmoids, since we want definite results instead of approximations.

In order to test our system we have split the data set in a train and a test set. The train set contains eighty percent of the images (or 16 images of each shape) and the test contains the other twenty percent of the images (4 per shape). Using the N-Fold testing principle explained above we average our results over five different test, using a different test set each time.

[1]This is obtained by choosing the biggest shape from the extracted shapes. Because some of the shapes were not solid white, manual whitening of these shapes was performed to guarantee the process of finding the outer contour never fails

6.2.1 Results

The results for the shape descriptors using the shape interpretation network are given in Table 6.1 below. No stopping criterion was used, as results did not show any signs of over-fitting. This is why all the tests have been run over the same number of epochs, in this case 2000. Most tests converged quickly and reached their goal within a thousand epochs. As the percentage of correct classifications is an average we have also included the deviation over the different test sets. The maximum score obtained on any of the different test sets is also included in the table.

# Descriptor type	Learning rate	Hidden Neurons	Max	Percent-Correct	Deviation
Radial	0.03	70	65.7	62.5	2.1
Radial	0.03	90	63.9	61.9	1.7
Radial	0.06	70	65.6	62.9	1.6
Radial	0.06	90	65.7	63.2	1.8
Distance	0.03	20	33.9	31.2	2.5
Distance	0.03	30	41.1	35.4	3.1
Distance	0.03	40	43.2	39.5	3.1
CSS	0.03	30	33.2	30.4	2.3
CSS	0.03	40	38.6	35.8	1.6
CSS	0.06	30	32.9	30.1	2.0
CSS	0.06	40	33.2	31.5	1.8
Orientation	0.03	70	69.4	65.5	3.0
Orientation	0.06	70	68.6	65.5	3.6
Orientation	0.03	95	70.0	65.1	3.2
Orientation	0.06	95	70.0	65.4	3.6
Pixel	0.03	70	63.6	59.5	3.5
Pixel	0.06	70	65.7	61.9	2.4
Pixel	0.03	95	65.4	62.1	2.3
Pixel	0.06	95	67.8	63.6	3.0

Table 6.1: *Results for learning a shape network on the 70 shapes of the [LLE00] database, for 2000 epochs.*

Before discussing the results we will give the specific parameters that applied to the descriptors during testing.

1. *Radial descriptor.* The radial descriptor is applied using 30 circles on images scaled to 200 by 200 pixels. The resulting vector, holding 90 values (three for each circle), is directly used as input for the network.

2. *Distance descriptor.* For the distance descriptor 25 bins are used on the histogram. The images are again scaled to 200 by 200 pixels.

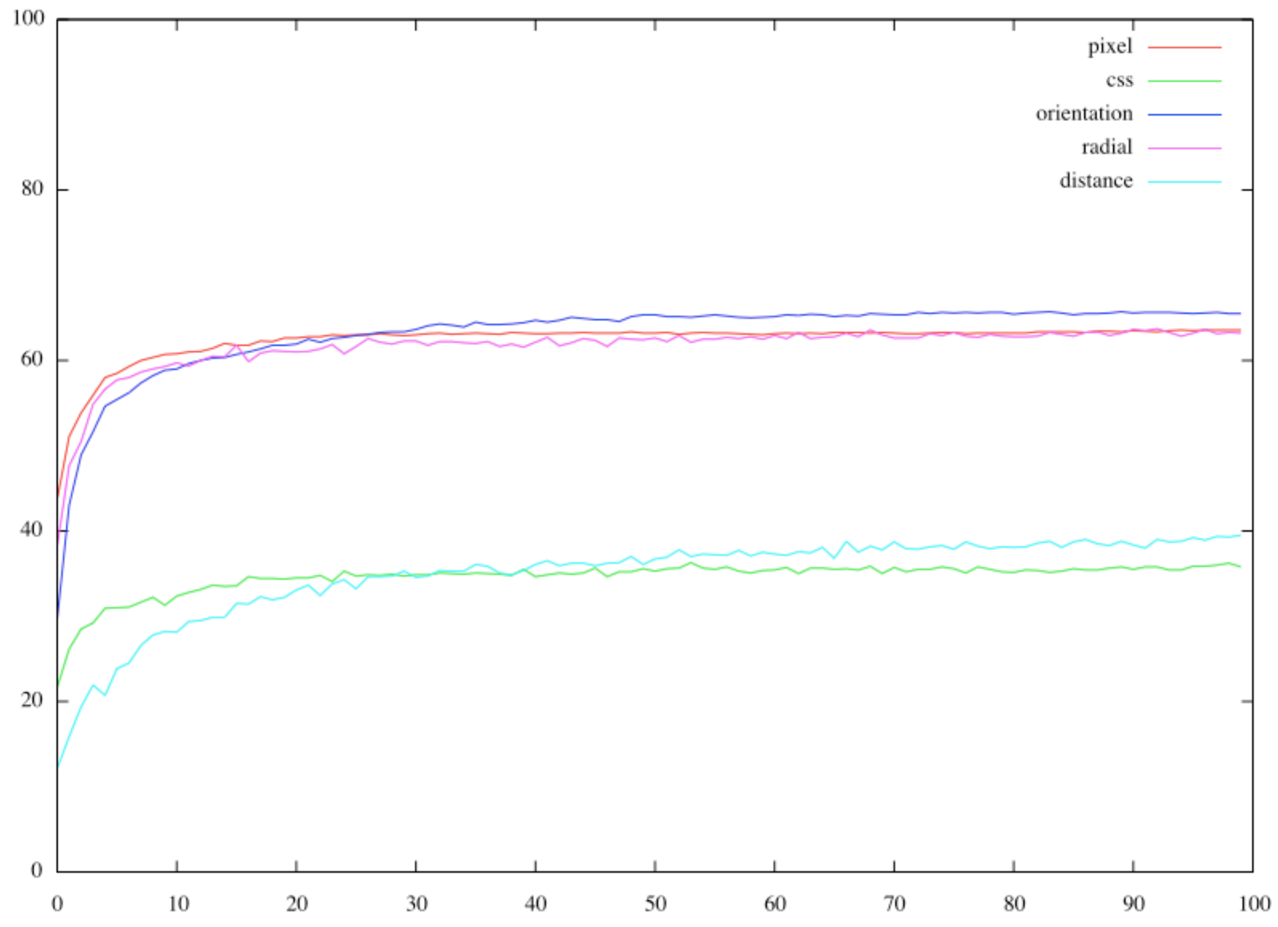


Figure 6.2: The convergence of the different descriptors in one graph. The horizontal axis shows the percentage of the 2000 epochs

3. *Curvature.* For the curvature descriptor 30 maxima are used to create the vector. As the curvature descriptor has relatively high computational costs the images are scaled to 100 by 100 pixels.

4. *Orientation descriptor.* The orientation descriptor is used by applying 4 by 4 orientation squares on images scaled to 100 by 100 pixels. Each square has two neurons representing it, one vertically and one horizontally activated neuron.

5. *Pixel descriptor.* The images for the pixel descriptor are scaled to 50 by 50. Any higher resolution caused the training to become extremely slow. Neurons corresponding to points on the shape get an activation of one, neurons not corresponding to a point on the shape get an activation of zero. Furthermore, to give the pixel descriptor a little more thickness around its contours, neurons adjacent to activated neurons are also activated by an input of 0.5.

In the graph in figure 6.2 we have summarized the results by plotting the test averages for each descriptor. It is promising to see that the orientation descriptor outperforms the pixel

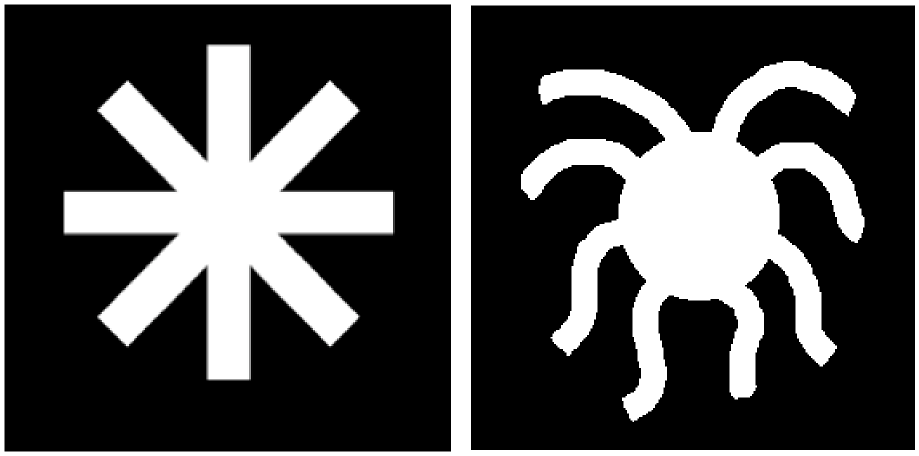


Figure 6.3: These two shapes have a very similar radial descriptor

descriptor, since it allows for much faster training than the pixel descriptor, which can take hours for a single test. The recognition rates are rather high considering the high number of different shapes, and especially considering the fact that so many shapes in the set show clear similarities. In the mistakes made by the system we can see the effect of these similarities. In one test using the radial descriptor, for instance, the octopus shape was classified as and eight-pointed shape 2 out of 4 times (see figure 6.3), although admittedly not all mistakes were that easily explicable.

As the deviations show, the results vary somewhat over the different test sets. Partly, this is due to the fact that the technique we are using is subject to a random initialization, as well as to the existence of different local maxima in the error surface. However, in some cases it was clear that the performance on some test sets were deterministically higher than on others. Apparently, some test sets were just more difficult than other test sets, probably because they contained shapes that differed rather substantially from those in the other sets.

6.3 Object recognition

To test our system on real objects we have created two different tests both consisting of pictures of objects on a solid, single colored background. The first set consists of 2D objects, in the form of playing cards. This set is used to test how well the system performs on a task that it should minimally be able to do if it wants to have any chance of performing well in a real environment. The second test is performed on a set of pictures of random 3D objects found around the house and shot from different angles. This is of course the hardest task by far and will indicate the actual ability of our system and its possible future potential.

In the previous section we saw that the radial descriptor was amongst those that had the highest performance. Also it is the least costly one in terms of computational capacity required. Both tests are therefore performed using the radial descriptor.

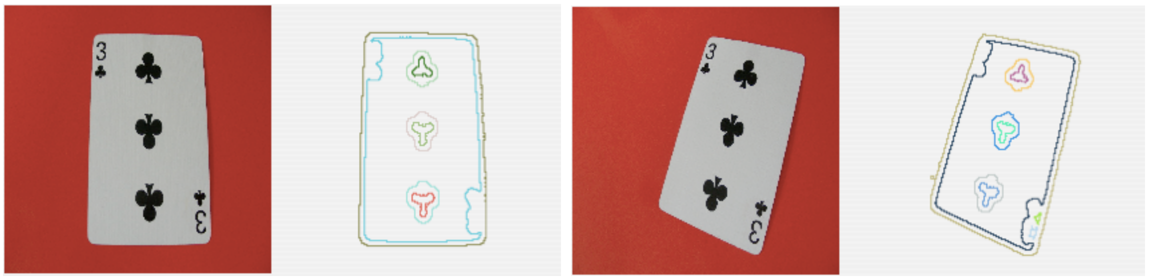


Figure 6.4: The results of shape extraction on cards visualized by indicating the extracted shape by different colors

6.3.1 Results for Playing Cards

We have now tested our system to see whether it can distinguish between 10 different playing cards. For this purpose we have used a set of pictures of playing cards shot by a simple digital camera. The cards selected come from each of the four different suits and include numeral cards as well as face cards.[2] Every card is represented by 20 pictures taken under different angles.

The preprocessing and shape extraction in these pictures is not an issue. The resulting shapes are clear, and will be consistent over the different pictures, as can be seen in figure 6.4. Therefore, this is the first test of our complete adaptive system, since it will tell us something about the adaptive system rather than about the components that fall outside of this system. We have already seen that the descriptor works with a single neural network. What we want to know now is how the combination of our different elements work, and whether the system is able to interpret the shapes in a satisfactory manner, and to recombine them in order to recognize the object.

The different settings used in this experiment are listed below:

- To keep the preprocessing as stable as possible we have shot the pictures of the objects under relatively constant lighting conditions. Angles of shooting the pictures have been varied and we have tried to simulate many of the affine transformations, as described in the first chapter.

- The shape interpretation network has been rebuilt for every single test. Each time its weights are randomly initialized and the network is trained until it manages to classify 20 out of 25 randomly selected shapes correctly. Normally, after creating one network that performs well, you want to save and use this one rather than create a new one every time. However, since we want to demonstrate the stability of the system we have decided to reinitialize the network for every test.

- In order to keep neurons in the shape interpretation network from generalizing too much, neurons with a stretched sigmoid activation function rather than the regular

[2]The complete selection includes the following cards: The ace of harts, the three of clubs, the five of diamonds, the seven of spades, the eight of spades, the ten of diamonds, the jack of spades, the queen of clubs, the king diamonds and a joker.

Shape Neurons	Object Neurons	Som Size	Max	Percent Score	Deviation
20	20	10	80.0	67.5	8.1
20	30	10	77.5	68.5	5.6
20	40	10	75.0	69.0	5.6
30	20	10	75.0	64.5	5.6
30	30	10	82.5	73.0	5.1
30	40	10	77.5	68.0	6.8
40	20	10	72.5	63.0	5.3
40	30	10	75.0	71.0	3.4
40	40	10	82.5	72.0	9.4
20	20	15	80.0	72.5	6.1
20	30	15	75.0	69.5	5.6
20	40	15	77.5	70.5	4.0
30	20	15	87.5	75.0	7.1
30	30	15	77.5	74.5	2.4
30	40	15	72.5	67.5	4.2
40	20	15	82.5	75.0	5.7
40	30	15	80.0	72.5	6.1
40	40	15	75.0	69.0	5.6

Table 6.2: *Results for the full object recognition system on pictures of playing cards, shot using a regular camera*

sigmoid activation function are used. This is done because we want the network to give us an approximation of similarity rather than just zeros and ones.

- The SOM is trained for 15,000 iterations. With some 50 shape combinations per object we wanted to make sure the SOM is shaped using all samples, while at the same time keeping the computational costs under control, since a number of tests had to be performed.
- The object identification network is trained over 500 epochs. This number is sufficient for the small set of objects we want to train the system on, since the network converges quite rapidly.
- The results are given in table 6.2, that is set up in the same style as table 6.1.

The results show us that the system is clearly capable of learning to recognize different objects. However, recognition rates with a maximum of eighty percent and averages around seventy percent are somewhat lower than we aimed for, especially since this is a rather fundamental task. We will come back to this issue in the last chapter.

6.3.2 Results for 3D objects

For our final tests we have taken pictures of objects found around the house. The set includes anything from a jar of peanut-butter to a box of toast. In appendix C samples can be found of the ten objects used to perform this test. This test is obviously the most difficult one and will show us whether the system has any potential when it comes to recognizing objects in a real environment.

Shape Neurons	Object Neurons	Som Size	Max	Percent Score	Deviation
20	20	10	67.5	58.0	13.3
20	30	10	67.5	57.5	5.2
20	40	10	65.0	58.5	6.1
30	20	10	67.5	59.5	6.4
30	30	10	67.5	58.0	10.7
30	40	10	65.0	60.5	3.3
40	20	10	62.5	54.0	8.2
40	30	10	65.0	59.5	5.1
40	40	10	55.0	51.0	5.8
20	20	15	72.5	56.5	9.4
20	30	15	72.5	59.5	9.3
20	40	15	67.5	59.5	5.8
30	20	15	67.5	57.5	7.2
30	30	15	70.0	58.0	6.2
30	40	15	72.5	57.5	7.9
40	20	15	75.0	63.5	8.6
40	30	15	67.5	59.0	5.6
40	40	15	80.0	61.0	12.1

Table 6.3: *Results for the full object recognition system on the Aloi database*

Most of the settings described in the previous section still hold for this task, as it is of the same nature. The only differences are

- In total there are twenty shots of each object, taken under a wide variety of angles. No standard has been chosen for the angles the pictures are shot under, as we are interested in seeing how the system works under general viewings rather than standardized views.
- Lighting conditions were not varied purposely, but with the usage of natural light they vary a little from picture to picture. In this way it is possible to see the results of the system under more realistic conditions.

As expected recognition rates in this tests are lower than in the previous one. The difference is, however, quite limited considering the increase in difficulty of the task. What can clearly be seen, though, is that maximum recognition rates and averages are further apart and the deviations are higher than in the previous test. This is most likely due to

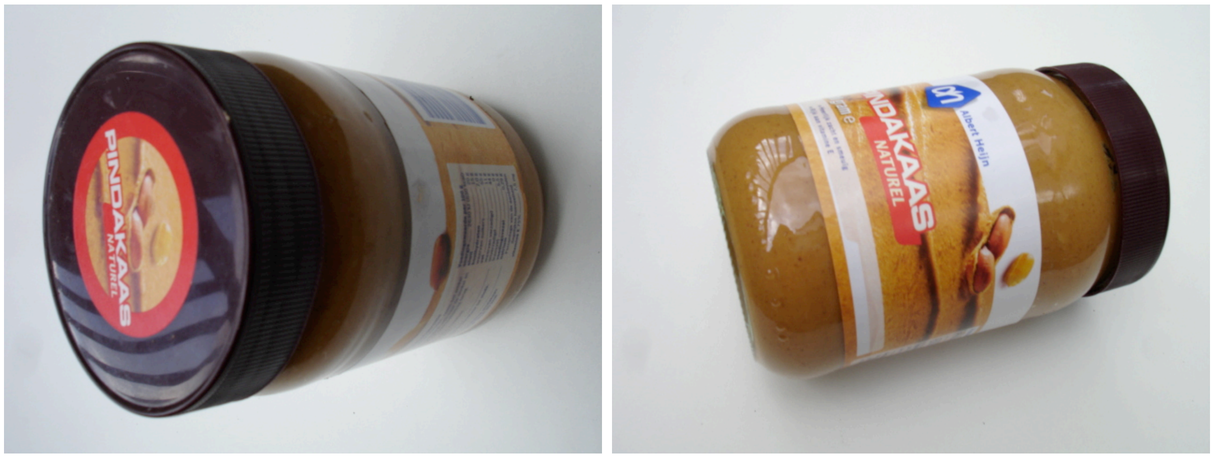


Figure 6.5: Samples of the peanut-butter jar seen from the top and side

the fact that the number of samples is limited, since the same number of pictures are used, even though we have moved from 2D objects to 3D. Especially in cases where objects are photographed from the top and from the sides (an example can be seen in figure 6.5). For tasks like these 20 pictures might be too low, and can easily yield high deviations over the different test sets as some will contain pictures taken from the top and others will not.

Now that we have obtained the results of our system over a broad spectrum of circumstances we are ready to move to the next chapter for a more in-depth discussion of the results.

Chapter 7

Discussion

In our view there are good reasons to believe that methods of artificial intelligence can provide good solutions for a problem as complex as object recognition. Obviously, the present application is not yet sufficiently developed to be implemented in important real-life applications. However, looking at the results in general, we believe that the current performance provides an opening for the future, which satisfies our main objective. To conclude this document we will review the performance of the system and discuss its future potential.

7.1 System

As our application covered many different aspects it is always difficult to attribute the level of its performance to a specific component. In our conclusion we will attempt to evaluate each component individually and discuss its performance, since it is of interest to assess which components perform adequately and which components do not. We will perform this evaluation by analyzing the results of the various tests and look at what the differences among these results can tell us.

7.1.1 Preprocessing

On the black and white images preprocessing performed well, as expected. For real color images performance is of course less exact, especially in cases of noisy or complex images. But, as mentioned earlier, there is no actual need for edge extraction to give nice smooth shapes. Rather, the results should be constant over similar objects. And if we take a look at the shapes that were abstracted from the different images of the same object we can say the preprocessing component performed rather well in terms of consistency.

It has to be mentioned that the extraction of the shapes, on top of the edge detection, does add to the already high costs of the preprocessing process. However, the resulting shapes can be used to create descriptors quickly and compactly, which has many benefits. Our desire to represent the object in the form of shapes required some way of splitting up the object into different entities anyhow. And any such process would have been costly, since it generally takes at least a full evaluation of every pixel in the image.

As for enhancements of this process it has to be said that the dilation followed by the subtraction in general performs well. The only problem with this method is that in some cases, when the edges are dilated, shapes become connected at points where the edge pixels were not, and should not, be connected. This is especially the case for low resolution images where the number of pixels between shapes decreases.

7.1.2 Descriptors and the Shape Network

The results for the various descriptors are not direct measures of their quality, since in the end a lot also depends on the performance of the adaptive system as a whole. Yet, as we have kept the structure of the system constant throughout the testing of the descriptors, we can at least take a look at which one performs best in our environment.

The results of the pixel representation, our benchmark, were with about 60 percent recognition good. The computational cost was, as expected, extremely high. In order to test this descriptor several computers were running for several days each. This in stark contrast to the other descriptors that did not need much longer than a few hours to be trained for several variations in their parameters on all the different test sets.

The orientation descriptor proved to be a good alternative to the pixel descriptor. The results obtained were better, taking only a fraction of the time to train. It also showed us, as many times before, that what we learn about processes in human cognition generally forms a good basis to solve problems on the computer.

The radial descriptor did not perform quite as well as the orientation descriptor, but was close. And its compactness and efficient calculation probably makes it the most fruitful descriptor we have used. It was the only descriptor out of the best performing three that was efficient enough to be used in our complete object recognition system.

The CSS descriptor and the distance histogram both did not perform extremely well. However, the distance histogram is simple, as its calculation is performed extremely fast and the resulting vector is so small that it can be used without any problems by a neural network. Therefore, the distance histogram should certainly not be completely disregarded. The CSS descriptor does perhaps not fully show the strength of the CSS to describe a shape. Limited computational power, combined with the difficulty to create a descriptor for a neural network, made it hard to get the fullest out of the CSS. But even with more computational power it is doubtful whether the performance of this descriptor will easily catch up with, for instance the radial descriptor.

To conclude our overview of the general performance of the shape descriptors in combination with the shape interpretation network we can be satisfied with the results, especially as the database used did not always have the easiest shapes to classify. Looking at figure 6.3 in the previous chapter probably illustrates this best. The fact that the interpretation network achieved such high recognition rates emphasizes what we already knew about neural networks, which is that they have powerful generalizing properties. So we believe with recognition rates of over 60 percent the result of our shape descriptors in combination with a neural network is sufficiently promising for the task they are supposed to perform.

7.1.3 Object recognition

The two tests for object recognition showed us results that can be interpreted in many ways. The results on recognition of playing cards showed that our preprocessing techniques are without doubt able to process and extract shapes from real pictures and that the adaptive system can generalize among these. However, the recognition rates are a little lower than we hoped for since the task at hand was a very basic one and we expected recognition rates to be closer to ninety percent. This shows us that even though our system has the capability to recognize, there is still clear room for improvement. Most likely the problem lies in one of the first two components of the adaptive system, the interpretation network and the recombination on the SOM. It is difficult to speculate which part to blame. Both components cut down the dimension of input information, so both can cause a loss of information when they are not performing their task properly. Since the results, although not as high as expected, surely indicate a reasonable capability of the system, this means that improving either of these techniques in the future can certainly bring rewards.

Similar conclusions apply to the final and most difficult test where objects were viewed from all possible angles. The results could have been a little better: even though our expectations were not as high as for the cards test, we still had hoped the recognition rate to be clearly above eighty percent. What is promising to see here, though, is that the results do not deviate that far from the results in the cards test. As this test moved from 2D inputs to 3D inputs, without an increase in learning samples, we were pleased to see that the percentage rates did not show a large decrease in performance. Since the difficult step from recognizing 2D objects to recognizing 3D objects was taken with little loss of recognition ability, this suggests that improvements to the easier 2D recognition process might also yield significant rewards in the 3D area.

Moreover, if we for a moment forget the high expectations that are built-in into every researcher, the results certainly show significant rates of recognition, indicating that the system is clearly capable of learning to recognize objects on its own. Given that there is a viable basis for recognition, further work could definitely prove to be both useful and rewarding.

7.2 Future/Further work

With our application we have shown some possibilities of using artificial techniques to perform object recognition in the real world. The program is designed in such a way that it can easily be expanded by either improving existing elements and techniques, or by incorporating new ones. New descriptors can be included, preprocessing methods can be replaced and neural systems can be adapted or extended.

To conclude this document we have included a couple of possibilities that come to mind when considering the expansion of the program.

- *Tuning or replacing components*

  The test results show that not all components work as well as they should. This means some of the components should be fine-tuned even further, or replaced by more suitable versions. For fine-tuning more tests can be run changing parameters to further optimize performance or use different vantage shapes to enhance shape interpretation. As the program is implemented in an object orientated style, components can easily be taken out and replaced. Also new adaptive systems can be created that utilize components differently.

- *Speeding up the neural system*

  The TS-SOM as seen in PicSOM could make adaptive systems more practical since saving time on the search methods of the SOM, could allow for the usage of a more detailed SOM. This is certainly something to be considered once the program works well. Of course this is mainly a question of time. Our primary concern in developing the adaptive system was to get it to recognize as well as possible. We did not yet reach the point where we could consider trading of computational power against faster solutions that might in some way compromise results.

- *Dissecting multiple objects*

  The program can be extended to work with multiple objects. Due to its shape-based approach the program can easily change the way it looks at a picture. Since in the current situation all shape combinations are mapped to the SOM, we can also decide to take a subsection of these. With this we could explore a picture in further depth rather than assuming there is just one object in it. Techniques could, for instance, try combinations of shapes that have close proximity to each other. Since a single step through the object network is not time consuming a lot of variations in the map of the shape-combination can be tried. And even to enhance the recognition of a single object, there could be an attempt made to leave certain shapes out of the main set. If the omission of shapes can lead to clearer outputs by the last layer of neurons this could indicate a more reliable recognition.

- *Color and Texture*

  A serious option to consider in the future is to use color or texture, which has not been handled in this thesis. It is of course not without value towards the process of recognition. Blue oranges do not exist, nor is there a large possibility of finding an orange pear. Colors are not all that difficult to include in the object recognition process. We could supplement the shape interpretation or descriptor with color information, only slightly increasing its complexity. Interpreting texture is more difficult, since we cannot as easily classify textures as we can colors. But there are some techniques for working with textures. Gabor filters as described in [FS89], for instance, have been used successfully in representing texture.

- *Neural Preprocessing*

  Another option for preprocessing an image is using a neural network. Not much work has been done in this area, yet some applications can be found. In [dRRDvV99], for instance, the application of kuwahara filtering to an image is learned by a neural network. The neural network gets a block of pixels as its input, on which it outputs the corresponding values of the original filter. The results are rather good and even though these techniques cannot be directly applied to our program, they show the possibility for using neural networks as a preprocessing tool. Yet a large reduction in computational costs, which is one of the main reasons for trying to find new ways of general preprocessing, is still hard to accomplish by neural networks.

Working with images for now is still a costly process. And it will take some time before we can actually extract information from pictures without making any reservations. Progress in this field will depend not only on further research, but also on further progress in the processing and storage capacity of computers. Given the interest in image processing that comes with the growing importance of security, as well as the interest in other automated solutions in our society, rapid progress in both areas is to be expected. This makes this field of research a very promising one for further work.

Bibliography

[AHS85] D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

[AMK96] S. Abbasi, F. Mokhtarian, and J.V. Kittler. Robust and efficient shape indexing through curvature scale space. In *BMVC96*, page Shape, 1996.

[Bie87] I. Biederman. Recognition-by-components: A theory of human image understanding. *Psychological Review*, 94(2):115–147, 1987.

[Can86] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[dRRDvV99] D. de Ridder, P.W. Verbeek R.P.W. Duin, and L.J. van Vliet. The applicability of neural networks to non-linear image processing. *Pattern Analysis and Applications*, 2:111–128, 1999.

[FM82] K. Fukushima and S. Miyake. Neocognitron: a new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15(6):455–469, 1982.

[FS89] I. Fogel and D. Sagi. Gabor filters as texture discriminator. *Biological Cybernetics*, 61:103–113, 1989.

[Fuk75] K. Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20:121–136, 1975.

[Hop82] J. Hopfield. *Neural networks and physical systems with emergent collective properties*, volume 79. 1982.

[HW59] D.H. Hubel and T.N. Wiesel. Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 155:385–398, 1959.

[Izh04] E.M. Izhikevich. Which model to use for cortical spiking neurons. *IEEE Transactions on Neural Networks*, 15:1063–1070, 2004.

[Jae01] H. Jaeger. The echo state approach to analyzing and training recurrent neural networks. *GMD Report 148*, 2001.

[Koh01] T. Kohonen. *Self-Organizing Maps.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[LKO02] J. Laaksomen, M. Koskela, and E. Oja. Picsom–self-organizing image retrieval with mpeg-7 content descriptors. *IEEE Transactions on Neural Networks*, 13:841–835, July 2002.

[LLE00] L.J. Latecki, R. Lakamper, and U. Eckhardt. Shape descriptors for non-rigid shapes with a single closed contour. *IEEE Conf. on Computer Vision and Pattern Recognition*, 5:424 429, 2000.

[MP43] W.S. McCulloch and W. Pitts. A logical calculus of ideas immanent in neural activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[MPE01] *Overview of the MPEG-7 Standard*, 2001.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. *Learning internal representations by error propagation.* MIT Press, Cambridge, MA, 1986.

[RMS91] H.J. Ritter, T.M. Martinetz, and K.J. Schulten. *Neuronale Netze.* Addison-Wesley, 1991.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386408, 1958.

[SB95] S.M. Smith and J.M. Brady. SUSAN – A new approach to low level image processing. Technical report, 1995.

[SH96] J. Schmidhuber and S. Hochreiter. Guessing can outperform many long time lag algorithms. Technical Report IDSIA-19-96, 6, 1996.

[TMRSSG00] L.A. Torres-Mendez, J.C. Ruiz-Suarez, L.E. Sucar, and G. Gomez. Translation, rotation, and scale-invariant object recognition. *IEEE Transactions on Systems, Man and Cybernetics*, 30:125–130, February 2000.

[VV99] J. Vleugels and R.C. Veltkamp. Efficient image retrieval through vantage objects. In *Visual Information and Information Systems*, pages 575–584, 1999.

[WH60] B. Widrow and M.E. Hoff. Adaptive switching circuits. *1960 WESCON Convention Record, Part IV*, pages 96–104, 1960.

Appendix A

Application

The program has been written completely in Java 1.5 and all of the components have been implemented personally. The objective of this appendix is not so much to give a full explanation of the source code of the program, but to give a general idea of the structure of the program and the way it can be operated. We will start by going over the main structure of the program briefly and then describe its individual components. More detailed information can be found in the generated Javadoc of the program. In the second part of this appendix we will describe how to operate the program and show some screenshots.

A.1 Structure

The program is divided into several packages. The main packages that make up the program are the *neuralnetwork* package, the *data* package and the *gui* package. They are all part of the *objectrecognition* package that is the main package: it holds the Controller class that is the control center of the program, and provides the communication between all packages.

A.1.1 Neural network package

This package holds everything necessary for the adaptive system. There is an abstract class describing the adaptive system and several implementations of this class are present. Furthermore, the package includes a subpackage *network* holding all the network components, from the networks themselves to the neurons they are made up of. The three most important components are listed below:

- *AdaptiveSystem*

  This is the abstract class that defines the general structure of any recognition system. It regulates testing and to a certain extent training, even though independent adaptive systems can have their own specific way of training. Furthermore there are some abstract methods any subclass has to implement. The three most important implementations available of the AdaptiveSystem class are the *SingleShapeSystem*, the *RandomSystem* and the *StandardShapeSystem*. The *SingeShapeSystem* is used for the testing of the shape interpretation network only. The *RandomSystem* is the full object recognition system with a randomly initialized shape network. And the *Standard-ShapeSystem* extends the latter by introducing the use of vantage shapes to train the shape network.

- *NeuralNetwork*

  The neural network class holds the collection of layers and the weights that connect the layers. Furthermore there are also the methods for propagating activation through the network and applying backpropagation.

- *SOM*

  The SOM class holds the implementation of the SOM and its general methods, such as finding the BMU or updating the weights.

A.1.2 Data package

The *data* package deals with the processing of all image data. Anything such as reading of files, segmenting images or the creation of descriptors is handled by this package. The three most important parts of this package are:

1. *DataControl*

   The DataControl is the class responsible for managing the training and test data. It contains methods to read the data from images, and divides the data over different sets so that they can easily be used for N-fold testing. Finally the DataReader provides a wide array of methods for retrieving samples, randomly or indexed, from the train set or the test set.

   Data is divided in several data-blocks as described in section 6.1. To make sure all test sets are the same size, the last datablock is used as the standard, since this one will have the least amount of samples in it as it is always the last one to receive its sample. This means that if data is not equally divided over the different blocks, the block with the lowest number of samples sets the standard and data beyond this point is discarded.

2. *Descriptors*

   An important subpackage is the *descriptor* package. This package contains all classes for creating descriptors as well as one abstract class, the *Descriptor* class, that holds the mandatory methods necessary for implementing a descriptor.

   Descriptor classes are required to also implement a *visualizeDescriptor* method, that creates an image that shows what the descriptor does. This makes it possible to check whether the implementation of the descriptor results in the desired effect.

3. *Tools*

   The *tools* package of the *data* package holds all the tools necessary to dissect an image apart from the descriptors. Here the *EdgeDetector* class can be found that holds the SUSAN edge detection algorithm. But also the *ImageTools* class that can perform simple operations on images, necessary for, *inter alia*, the creation of descriptors.

A.1.3 GUI package

The *gui* package contains all the panels for the different tabs of the program, one class for each panel. Furthermore there is an *ImageCreator* class which creates images that can be drawn to the screen. And there is a *ResultPlot* class, extending the canvas, that draws a nice, resizable graph of the networks results.

A.2 How to operate the program

We will now give a short explanation of how to operate the program. Since the program was created for personal usage it does not always provide warnings or guidance like one could expect from a program made for public use. However, as long as the necessary steps are performed in the correct order the program works properly.

The main class is in a package, so it has to be started from the top of this package, i.e. from the directory that holds the *objectrecognition* directory. From here the program can be started typing:

```
java objectrecognition.Recognition
```

In case the plan is to load large images it is wise to increase the memory heap by filling out a certain number of megabytes. Increasing it to 512mb would work as follows:

```
java -Xmx512m objectrecognition.Recognition
```

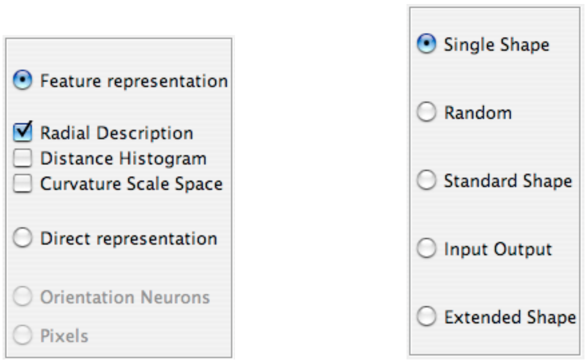
When working with the program it is advisable to:

- always load data before using any of the methods of the adaptive system. Wait till all of the data has been loaded, before creating a new adaptive system.
- make sure parameters are set correctly before starting any process. Pay attention to such things as selecting the right type of preprocessing technique and filling out the correct amount on neurons considering the size of the chosen descriptor.

A.2.1 Setting neural parameters

On the *Neural* tab all the things needed for creating and training the adaptive system are present.

- On the left side in figure A.1 a panel can be found that can be used to select the descriptor type(s). Using the radiobuttons and checkboxes the desired descriptors can be selected to train the network on. The program also allows for a combination of indirect features to be used. It is important to note however that the types to be used should be loaded into the memory of the program, as described in the explanation of the next tab.



[✓ Feature representation]

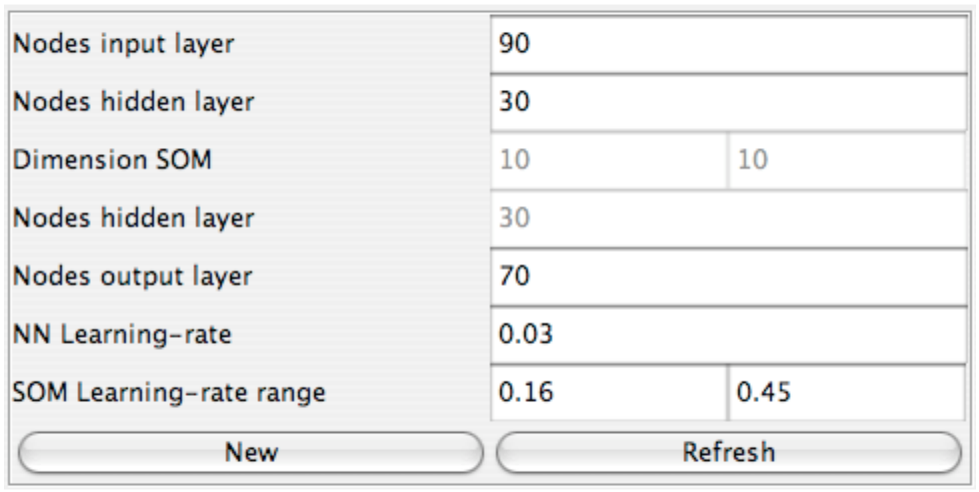
[✓ Radial Description]
Distance Histogram
Curvature Scale Space

Direct representation

Orientation Neurons
Pixels

[✓ Single Shape]
Random
Standard Shape
Input Output
Extended Shape

Figure A.1: A selection of the descriptors that the system has to be trained on



Nodes input layer	90	
Nodes hidden layer	30	
Dimension SOM	10	10
Nodes hidden layer	30	
Nodes output layer	70	
NN Learning-rate	0.03	
SOM Learning-rate range	0.16	0.45

Figure A.2: Parameters for the adaptive system

- On the right side in figure A.1 a panel can be found that allows the user to select the type of system to be used. This selection should be made before a new system is created using the middle panel.

- Using the middle panel as pictured in figure A.2, the new adaptive system can be created. Using the text-fields the sizes of the dimensions of the layers can be set as well as the desired learning-rates (rate of adaption to the error). The number of parameters in the *output* field will automatically adapt to the loaded data, which saves the user time by not having to find out how many directories he has selected. Based on the selected system on the right of the screen some fields may be disabled when they are not necessary for the creation of such a network. After all parameters have been set the *New* button can be used to create the new adaptive system. The *Refresh* button, next to the *New* button can be used to reinitialize part of the adaptive system, as it is not always desirable and often tedious to create an entire new system. Using this button the object classification network of the extended system will be reinitialized.

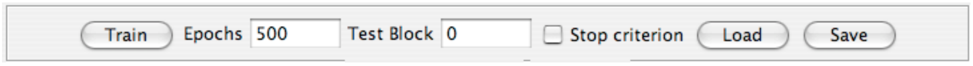


Figure A.3: Training options for the loaded system

Pressing the train button on the lower panel in figure A.3 will initiate the training process. A window with a graph will open, displaying the percentage of correct recognitions in the total test set. The percentage will be calculated after every percent of the training process that is completed. With the text-field for the number of epochs set to the standard 500 the percentage will be calculated every five epochs. Of course the test data will be kept separate from the train data and no adaptations will be made on the basis of the test results.

A.2.2 Data loading

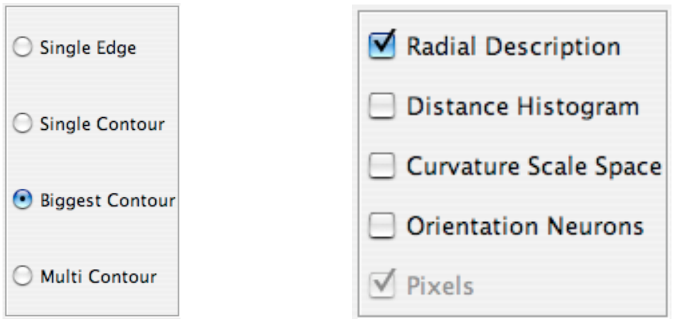
Before any network is created the necessary data should be loaded. This is done utilizing the second tab, the *data* tab, of the program. This tab consists of four main parts.

- Using the left panel in figure A.4 a selection of the type of preprocessing can be made. The selection should be based on the kind of experiment that is desired. In order to work with real objects the *Multi Contour* option should be selected. The image will then be segmented into separate shapes based on the procedure in section 3.3. For a simple shape recognition experiment one of the other options should be selected. For most of these the *Biggest Contour* option is the most appropriate, as this finds the largest contour, normally the outer contour. Sometimes it can be desired to skip the procedure in section 3.3 and work with raw edge data as provided after applying the SUSAN algorithm without any further preprocessing. For the pixel approach this method could, for instance, be used when the lines of the edge will be thicker, which might suit the constraints of the pixel based recognition approach better. *Single Contour* finds the same image as *Multi Contour* does, but in this case the multiple shapes are not found separately but are all treated as one. This method can be used to perform object recognition in combination with the single shape system.

- Using the panel on the right in figure A.4 the selection of the descriptor types that will be loaded can be made[1]. This should be done with care, since any option that is not selected in this panel before loading the data will make it impossible to use this type of descriptor in the experiments of the system. At this stage of the development of the program it is not yet possible to add them later, and if the user nevertheless wants to add them, the entire data loading process has to be repeated.

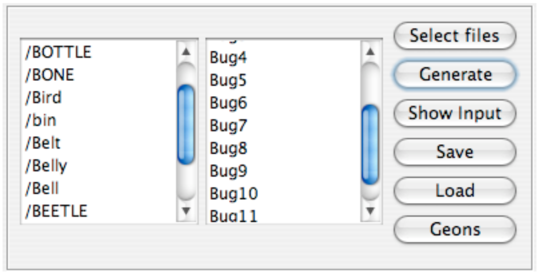
- Using the middle panel (figure A.5) the data can be loaded and the results can be viewed. Using the top button on the right a selection of directories to be loaded can be

[1]Even though Pixels is listed this feature will always be loaded (as it is necessary to create any of the other descriptors).


Single Edge
Single Contour
[✓ Biggest Contour]
Multi Contour

[✓ Radial Description]
Distance Histogram
Curvature Scale Space
Orientation Neurons
[✓ Pixels]

Figure A.4: Preprocessing selection panel


/BOTTLE
/BONE
/Bird
/bin
/Belt
/Belly
/Bell
/BEETLE

Bug4
Bug5
Bug6
Bug7
Bug8
Bug9
Bug10
Bug11

Figure A.5: Panel that allows for selecting, generating and visualizing data

made. This is an important step for which the correct procedure needs to be followed in order to have the system analyze the data correctly. The process works as follows: every directory loaded is treated as holding one type of shape or object only. This means that data to be trained on should be divided into directories with only one type of shape or object per directory. Putting different shapes or objects in one directory will keep the program from making the proper distinction between objects or shapes of different types. The name of the directory will serve as the type name of the respective object or shape. For the program to work properly care should be taken that all shapes or objects should be equally represented since the program does not take into account the number of items per directory. If one directory contains more samples than the other it will be seen more frequently by the system possibly compromising the results of the training. After the data has been loaded a list of the data will appear in the column on the right. The mouse (in combination with the keyboard) can be used to select samples out of the list. Using the *Show Input* button the selected samples can be viewed, to verify the quality of the preprocessing steps. When the *Multi Contour* option is selected the different shapes will be colored randomly to show the different shapes. The *Save* and *Load* buttons can be used to save and to load previously created representations. However as the information is currently encoded in an elaborate XML the data files easily become extremely large, quickly exceeding the desired boundaries.

- The bottom panel shown in figure A.6 allows the dimensions of the data to be set. As a standard option every image file is resized to 200 by 200. However, for an experiment

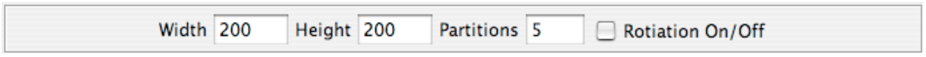
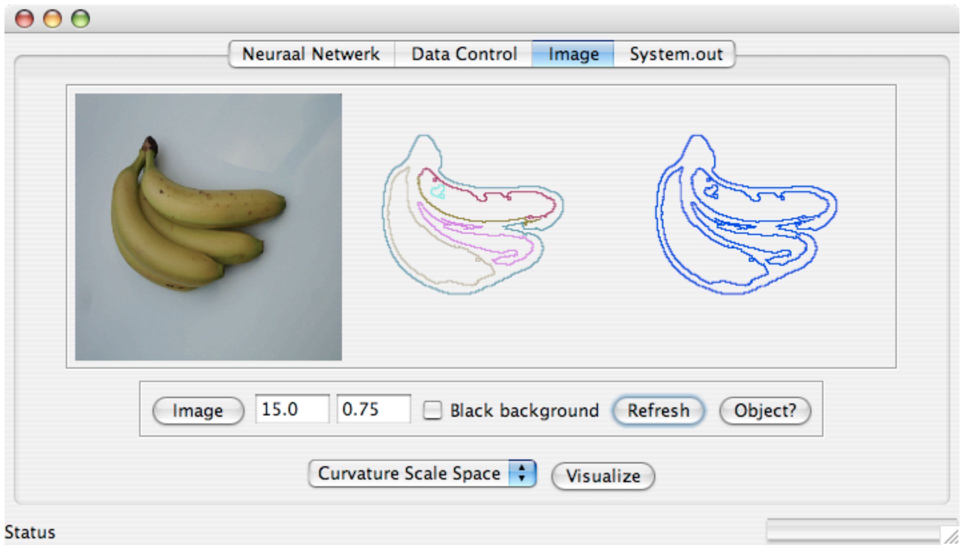

Width 200 Height 200 Partitions 5 ☐ Rotiation On/Off

Figure A.6: Width, height and other options



15.0 0.75 ☐ Black background

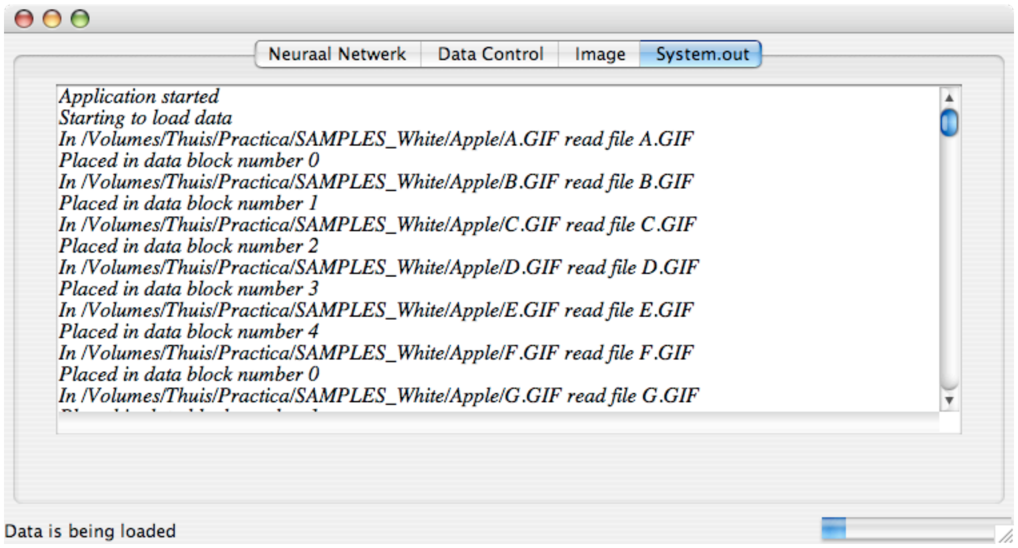
Curvature Scale Space

Figure A.7: Tab where image techniques can be viewed and parameters can be adjusted

using the *pixel* representation this leads to extremely large representations. In these cases further downsizing is advised. A text-field is available to select the number of blocks the data should be divided in. As has been explained in section 6.1 of the main document, these blocks form the basis for testing the system, so it is important to realize that the number of blocks determines the size of the test set. The standard here is five blocks, making the test size a fifth, hence 20 percent, of the data set. Furthermore a check box rotation (on/off) is present. Enabling this option will include the preprocessing step described in section 3.4, thus introducing a heuristic for rotating the object or shape to a standard orientation.

A.2.3 Image tab

Using the image tab in figure A.7 a better perspective can be obtained on the preprocessing steps. An image can be loaded and both the final result (all the way till the last preprocessing step) and an extra visualization are displayed. The extra visualization serves to verify the quality of specific parts of the preprocessing steps. As a standard an image is displayed connecting all the points of the shapes by lines, so that it can be verified whether they are ordered correctly. Using the *Visualize* button and the selection box also the results of the CSS process and the finding of circle points can be checked. Finally a button *Object* is included that utilizes the currently loaded system to classify the displayed image.



```
Application started
Starting to load data
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/A.GIF read file A.GIF
Placed in data block number 0
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/B.GIF read file B.GIF
Placed in data block number 1
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/C.GIF read file C.GIF
Placed in data block number 2
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/D.GIF read file D.GIF
Placed in data block number 3
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/E.GIF read file E.GIF
Placed in data block number 4
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/F.GIF read file F.GIF
Placed in data block number 0
In /Volumes/Thuis/Practica/SAMPLES_White/Apple/G.GIF read file G.GIF
```

Data is being loaded

Figure A.8: Tab that displays feedback on the running of the program

A.2.4 Output tab

In the tab in figure A.8 feedback is given on the steps the program undertakes; its primary use is to facilitate the tracing of errors in the program.

A.2.5 Progress-Bar

Another status indicator is the progress-bar. This element, that is constantly visible in the bottom of the screen indicates the progress made during lengthy processes. While loading data or training the adaptive system such a bar indicates the progress of the action.

A.3 Output

The program offers two options for viewing results. One is the earlier mentioned *ResultPlot* window, producing a real-time graph in a pop-up dialog that shows the performance of the network on the test set and is updated every time another percent of the training has been completed. The results are displayed by a blue line that indicates the percentage classified correctly of the current test set.

Another option is to plot data in gnuplot. Some of the classes contain methods for writing data to files that can be read by gnuplot. In the source code the usage of these methods will be commented, but they can easily be reinserted to write data to a file when the process is performed in the program.

A.4 Code and documentation

The source code as well as the samples used are available at:
`http://www.students.cs.uu.nl/~jpvries/thesis`
together with a full documentation created using javadoc.

Appendix B

Digital Image Processing

Digital image processing in general brings with it a number of difficulties. One of the main problems is that all the points lie on a discrete 2D grid. Problems often arise when trying to compute data normally based on continuous principles (derivatives, Gaussian functions, etc.).

We will not discuss topics in general, but we will take a look at the problems we had to overcome to work with images and shapes in a computer environment.

B.1 Centroid

The geometrical center of a shape, its centroid, is calculated for both the x-coordinates and the y-coordinates separately. Its coordinates are the average values of those coordinates, i.e.

$$x_{center} = (\sum_{i=0}^{N} x_i)/N, //and//y_{center} = (\sum_{i=0}^{N} y_i)/N \tag{B.1}$$

Since each point has to be evaluated once to calculate the resulting coordinates, the complexity is $O(N)$, with N being the number of points on the shape,.

B.2 Longest chord

The longest chord of a shape, is the line between those two points in the shape that lie apart the furthest. This chord can easily be found in two steps. First the point furthest from the centroid is found, and in a second step the point furthest from this point is located. In the end this leads to two runs through all the points of the shape, which makes the complexity of the procedure $O(N)$ where N is, again, the number of points on the shape.

B.3 The brightness formula

The ways the brightness of a pixel can be calculated using the red, green and blue component are numerous. The simplest way is just to average the three components. We have, however, opted for a different way of scaling the three elements of the pixel. The scaling factors used,

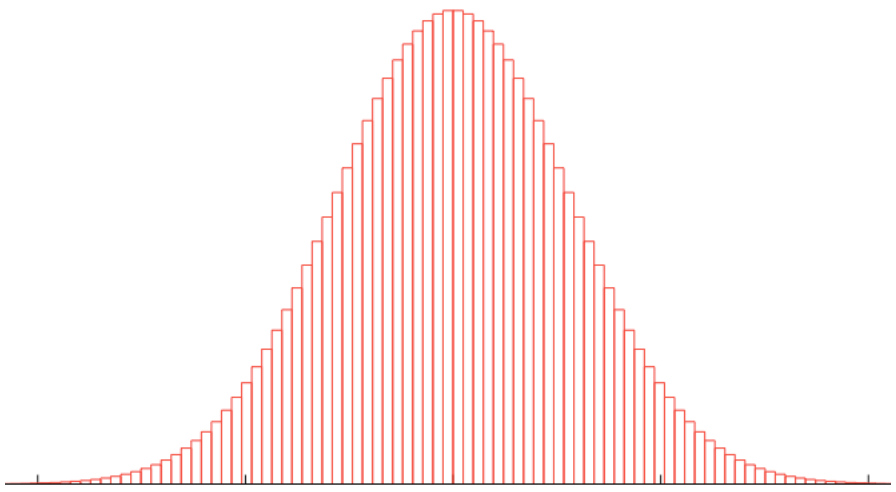


Figure B.1: An example of a Gaussian convolution kernel

which have been recommended by the *world wide web consortium* (`http://www.w3.org/`) are given in the equation below.

$$b(p) = 0.299 \cdot red(p) + 0.587 \cdot green(p) + 0.114 \cdot blue(p) \tag{B.2}$$

Although, interestingly, the scaling factors for the individual components do seem to correlate with the frequency of red, green and blue cones present on a human retina, we have not yet been able to find the reasoning behind this formula, and therefore cannot comment upon its justification. However, the formula did result in better edge detection results than other options.

B.4 Convolution

Convolution is the process of determining a new value for a point by taking into account not just the current value of the point, but also its surrounding area. The way convolution takes into account this area is by the usage of a so-called *convolution kernel*. In figure B.1 we have an example of a 2D convolution kernel. Placing the kernel, with its center on the point in question, allows us to approximate the new value of the point. This is done by taking the sum of the multiplications of the kernel's height by the value of the underlying points. The total sum is generally normalized to make the area of the kernel equal to one so that the new value is in the same proportion as in the previous situation. It is also possible to perform 3D convolution, which works in a similar way, but now with a 3D convolution kernel.

Since the values of images are not continuous but discrete a convolution kernel should also be a discrete function. For a Gaussian kernel, as used for the CSS descriptor, this requires the construction of a discrete Gaussian function, on the basis of its continuous curve. The simplest way is to calculate the values of the function that correspond to the discrete points and use these. This is done for each value. Since the Gaussian function goes on forever we end the process once a certain threshold value has been reached.

Appendix C

Test Sets

This last appendix shows a selection of the test sets used to train and test our system. The images have been reproduced here to give the reader an idea of the kind of data the tests have been performed with.

In the figures C.1 through C.4 a sample of each of the 70 different shapes from the MPEG-7 database [LLE00] is shown. Figure C.5 shows a sample of each of the 10 playing cards. Finally, in figure C.6 the reader will find a sample of each of the 10 3D objects used.

C.1 MPEG-7

C.2 Playing Cards

C.3 3D Objects

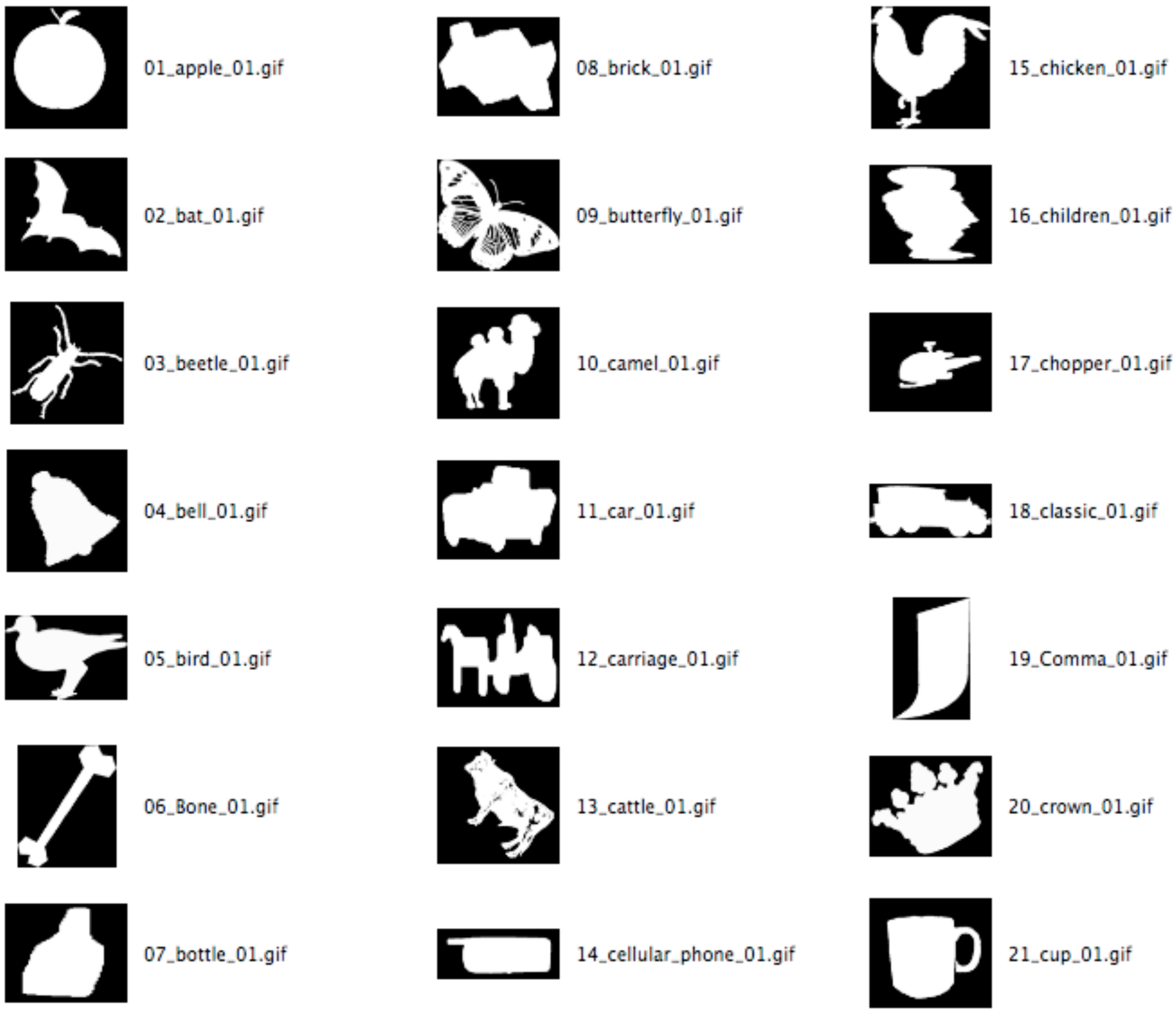


Figure C.1: Examples of the shapes in the MPEG-7 test database.

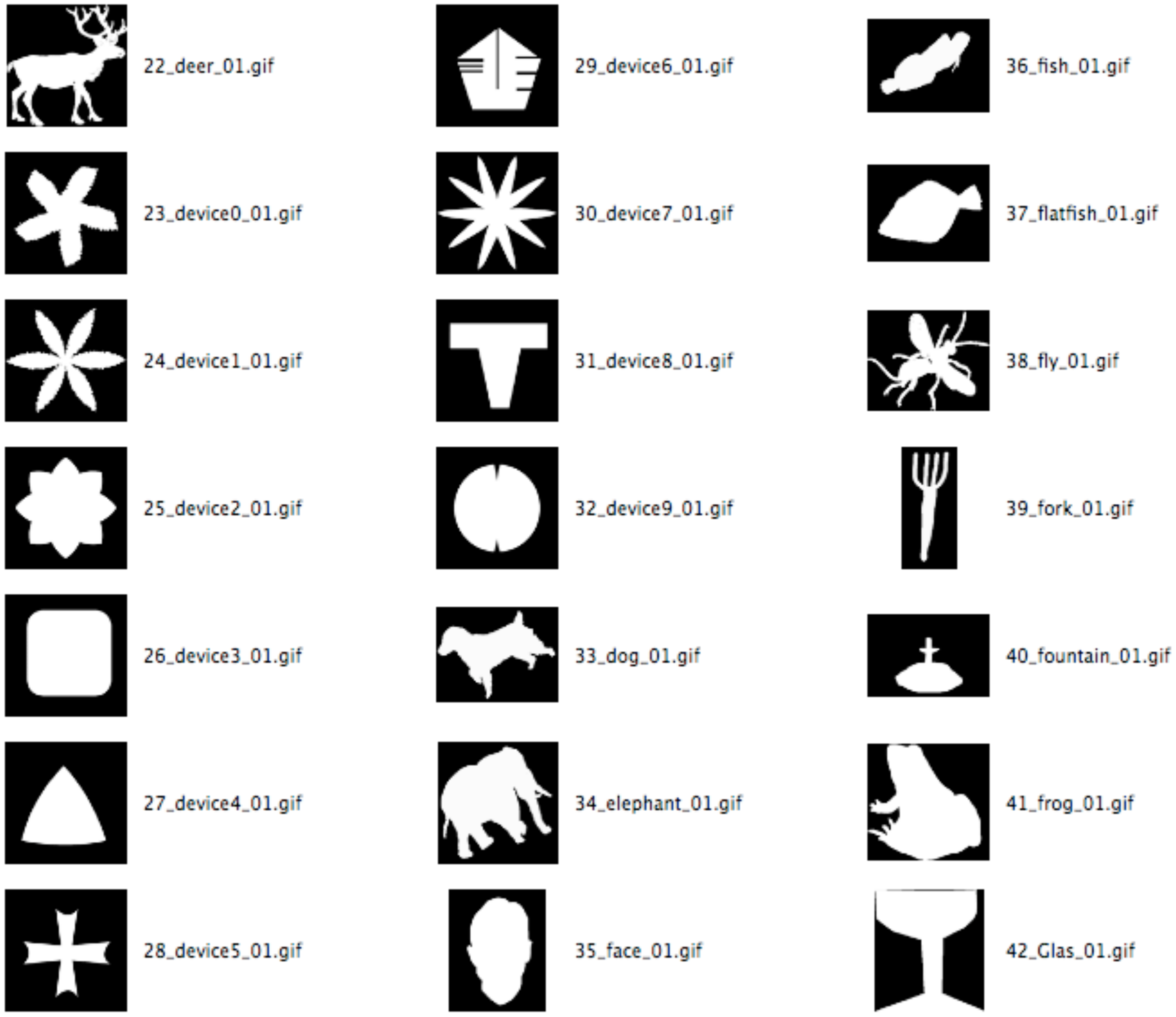


Figure C.2: Examples of the shapes in the MPEG-7 test database.

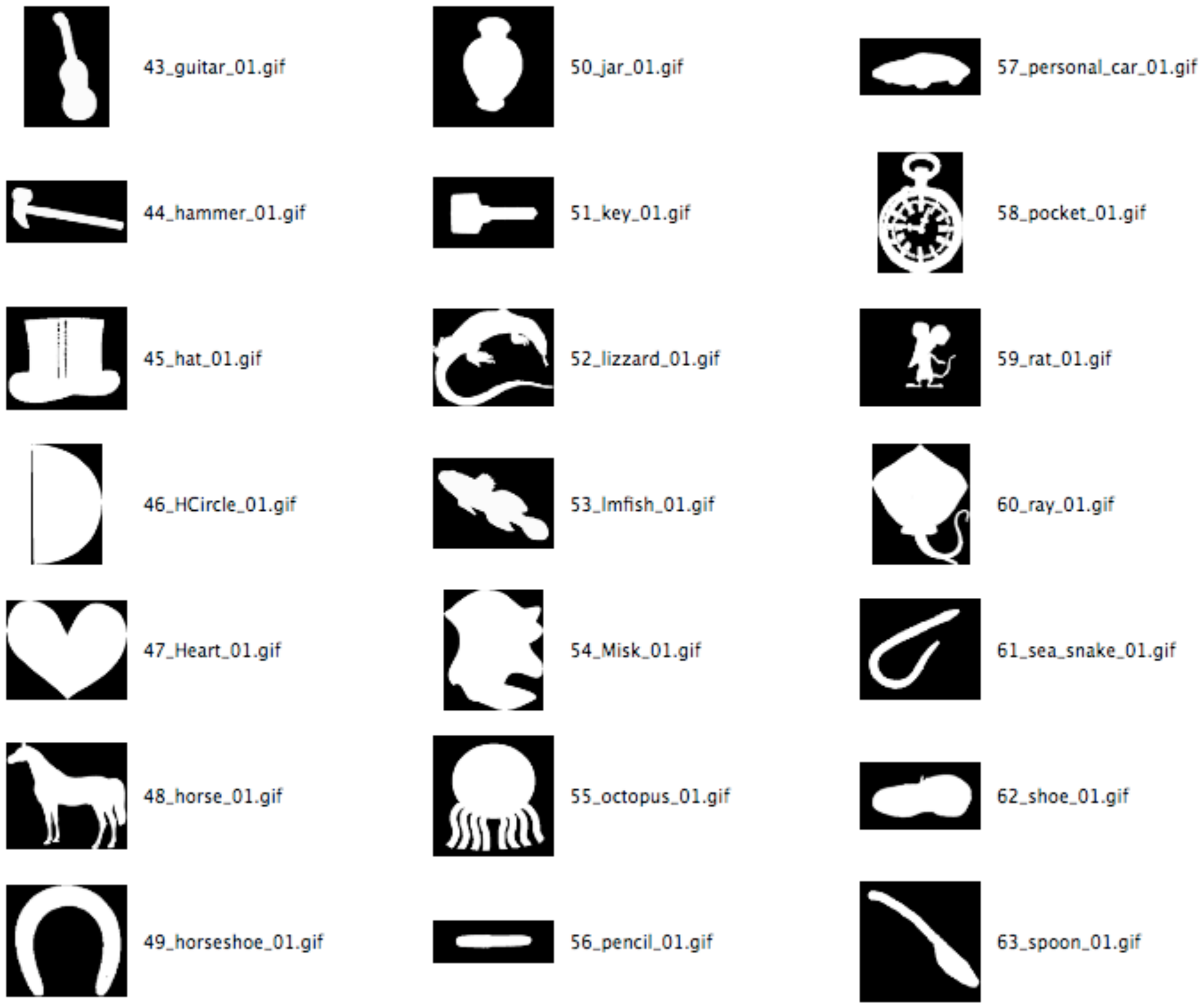


Figure C.3: Examples of the shapes in the MPEG-7 test database.

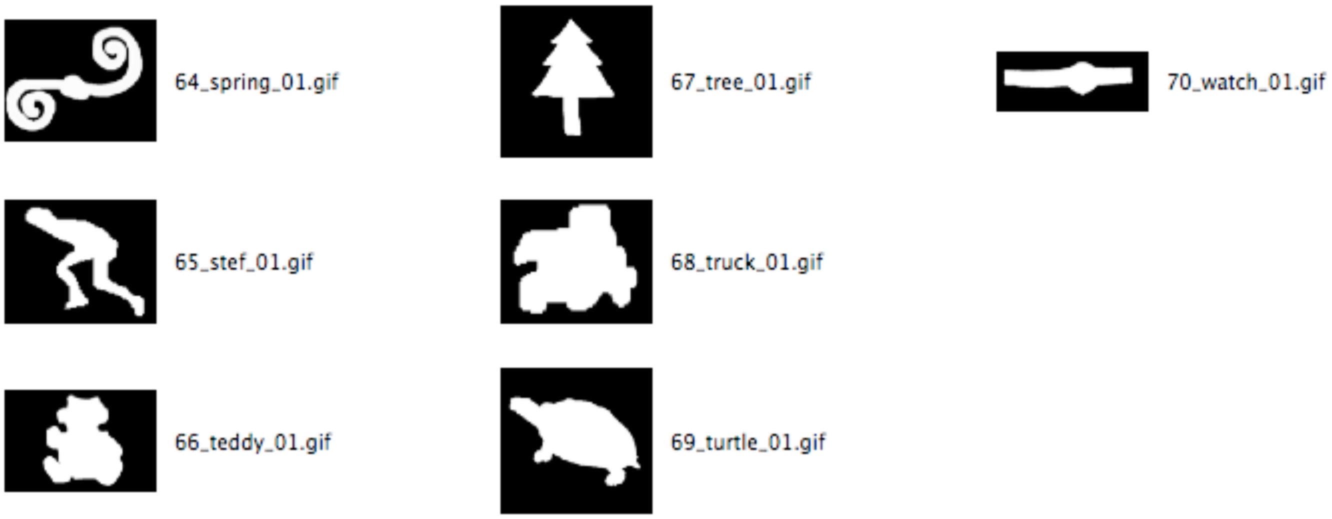


Figure C.4: Examples of the shapes in the MPEG-7 test database.

DSCN0111.JPG

DSCN0144.JPG

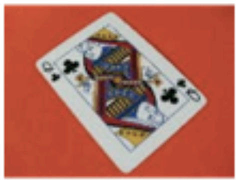
DSCN0183.JPG

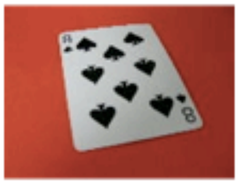
DSCN0222.JPG

DSCN0250.JPG

DSCN0276.JPG

DSCN0309.JPG


DSCN0338.JPG


DSCN0366.JPG

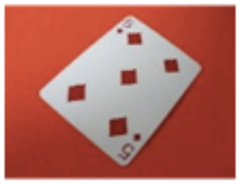

DSCN0404.JPG

Figure C.5: Examples of the pictures of playing cards, one of each.

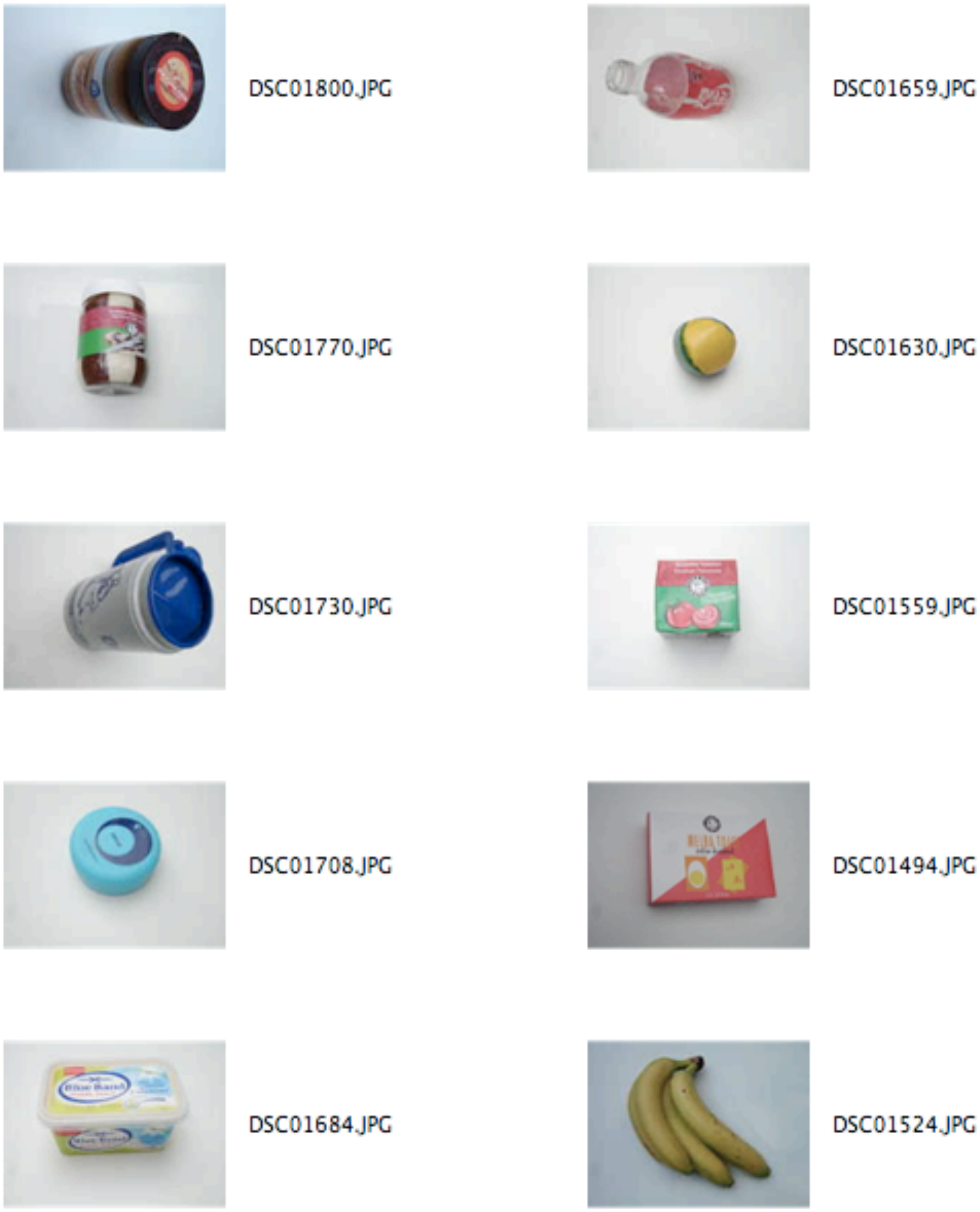


Figure C.6: Examples of the pictures of the 3D objects, one of each.