Richard Lancaster
Churchill College

R.Lancaster@carrotworks.com

Target location in aerial photography

Cambridge University Computer Science Tripos

Part II project dissertation 1999

Richard Lancaster
Churchill College

R.Lancaster@carrotworks.com

Target location in aerial photography

Cambridge University Computer Science Tripos

Part II project dissertation 1999

Word count:	11,700
Project Overseers:	J G Daugman and L C Paulson
Project Supervisor:	Andrew Penrose

Original aims

To design and implement an Automatic Target Recognition (ATR) system capable of locating cars in vertical aerial photographs.

Work completed

The original aims have been met. A functional ATR system capable of locating cars in vertical aerial photographs with reasonable accuracy has been designed and successfully implemented.

Special difficulties

None.

Copyright notices

The aerial photographs reproduced in this document are copyright the University of Cambridge Committee for Aerial Photography. They are reproduced here with permission, but may not be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the copyright holder.

All other parts of this document are copyright Richard Lancaster and may not be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the copyright holder.

Contents

Chapter 1

Introduction

Automatic Target Recognition (ATR) is a branch of the fields of Computer Vision and Pattern Recognition. In general an ATR system is one that reads in real world data from sensors, then locates and identifies any objects of interest contained in that data. Classic ATR applications range from the detection and identification of incoming aircraft, to searching spy satellite imagery for the military hardware of an adversary.

1.1 Specification

The intention of this project was to implement an ATR system that located cars in vertical[1] aerial photography[2]. So for example, if given an aerial photograph of a town, it would return the coordinates of all the cars that are visible in the image.

The aim was to produce a back end library that conceptually had a single function call. The format of this function call was to be along the lines of the specification in Figure 1.1.

```
AnalyseImage(<image>)
returns <array of object locations>

PARAMETERS:
<image> : The image to be processed.

RETURNS:
An array of (x,y) coordinates describing
where all the cars detected are to be
found in the image.
```

Figure 1.1: *Specification of function call to be provided by back end library*

Potential applications of such a system would include aiding in the investigation of traffic congestion, or if suitably adapted, tracking the movements of military vehicles.

1.2 Background

ATR is a field in which there is a large body of past and ongoing research. There therefore exists a wide range of building block techniques and system designs. The problem however is that there are no general solutions. So a design that works well for a particular application will probably require fairly heavy modification, at least of its low level components, to enable it to solve even a quite closely related problem. Hence constructing an ATR system is a matter of selecting and hand tailoring a number of basic building blocks, then adding in some original ideas and techniques to solve the problems specific to the particular application.

1.3 Previous related work

It it highly likely that the military has systems along the lines of this project to track the movement of military vehicles. These are also undoubtedly classified. In the academic domain there is a lot of research available on the building blocks of such a vision system. However while there is a reasonable probability they exist, the author has yet to locate data on any complete systems that perform a function similar to the specified task.

[1]In the aerial photography industry, vertical refers to photographs taken looking down from directly above the object.

[2]Aerial photography is taken here to mean data obtained from satellites, conventional aircraft, airships or balloons.

Chapter 2

Preparation

This chapter details the research, thought processes and design that took place before implementation of the project commenced.

2.1 Requirements analysis

One of the first tasks was to generate a precise set of requirements for the system. These requirements were designed to be general enough, such that the library developed during the project, could be used for any non real-time application that required the location of cars in top down aerial imagery. They were as follows:

- The system must be in the form of a black box library.
- When passed a colour top down aerial image, the system must return a vector of coordinates specifying the locations of cars within the image.
- The system must locate a high percentage of the cars that are visible in the image.
- The number of objects that are not cars, that are marked as being such, must be as low as possible.
- The system does not need to locate partially occluded cars[1].
- The system will be used for offline image analysis.
- The system must be able to process an image at least as fast as the same task could be performed by a human.
- The system must be executable on high end UNIX platforms so that optimum performance can be obtained.

As well as influencing decisions made throughout the design and implementation stages of the project, these requirements immediately led to a number of global decisions:

- As the system had to execute fast on UNIX platforms. It was therefore clear that either C or C++ would need to be used for its implementation, as other languages such a Java simply do not provide the performance required for this type of application or are not available on all of the target platforms. Further it was decided to use C for the implementation as full C++ support on UNIX is patchy at best[2]. The use of C does not however preclude the system being designed on OO principals.
- To execute on high end UNIX platforms the system would therefore have to be POSIX [1] compliant.

2.2 Obtaining a dataset

A dataset of aerial photographs was obtained and used with permission from the University of Cambridge Committee for Aerial Photography. These were high resolution, vertical, colour images of parts of the town of Cambridge, examples of which are included[3] throughout this document.

[1]This is a problem way beyond the scope of this project.

[2]Indeed even mainstream UNIX variants such as Linux only obtained good C++ support 6 months before this project was commenced.

[3]See copyright notice in front matter.

2.3 Research

At the start of the project, the author, while familiar with the language and operating system to be used, had very little knowledge of computer vision systems and techniques. A two month period between October and December 1998 was therefore spent researching techniques such as pattern recognition, segmentation, Fourier domains, neural networks and other classification methods. The results of this research will be referred to throughout the following sections.

2.4 The generic architecture of an ATR system

Early in the research period it became clear that there was a generic architecture common to most ATR systems of the type being developed. This architecture is illustrated in Figure 2.1. The function of each of its modules is as follows.

Preprocessor

This module cleans up the input image and converts it into formats suitable for processing by the downstream modules.

Detector/Segmenter

This module is a course grain first pass filter. Its function is to analyse the image passed in. Then pass out only a small number of segments[4] of the image, each of which it believes might contain an object that is being searched for. The downstream modules can then look at each of the segments in turn and simply say, "Yes, this is an object that I'm looking for" or "No, this is not what I'm looking for".

Figure 2.2 shows an example input image from this particular ATR problem. The white polygons overlayed on it are the boundaries of the segments that might be output by a typical segmenter designed for this application.

The operation of this module is intended to achieve two important goals:

[4]A segment is defined in this context as a contiguous area of an image

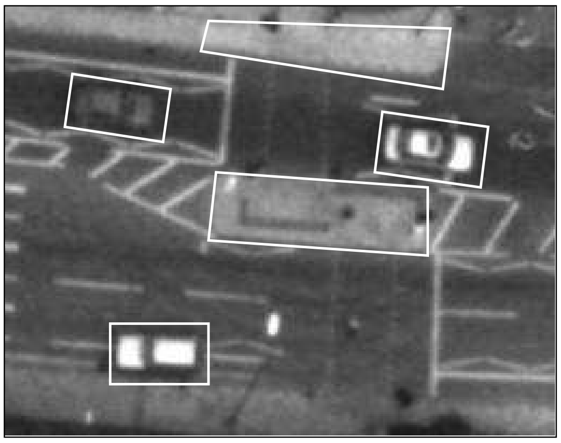


Figure 2.2: *Possible output from an ATR segmenter of a car location application*

- By selecting the segments to be passed downstream using a computationally inexpensive set of heuristics, the module can dramatically reduce the search space that needs considering at a very low cost. Hence the computationally expensive downstream modules have to do as little work as possible. The heuristics used must of course err on the side of passing too much downstream and therefore never discard something that is actually being searched for.
- If the segmenter is sufficiently clever. Then the segments which contain objects of interest can contain only the objects of interest and no background scenery at all. Notice that this is the case in Figure 2.2, where those objects that are cars are exactly wrapped by their segments. This means that the downstream modules don't get misled during classification by background data surrounding objects of interest. Indeed the Fourier invariance transforms discussed later rely on the segment containing only the object of interest and no background information.

Invariance transforms

It is often the case that the objects being searched for in the image can occur in any orientation or scale. This causes problems for any program that is trying to decide whether the image segment it is looking at matches the database of examples that

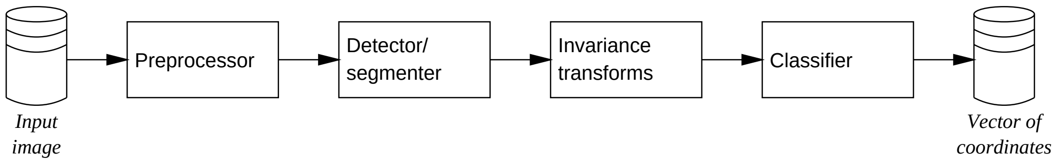


Figure 2.1: *The generic architecture of an ATR system of the type being constructed*

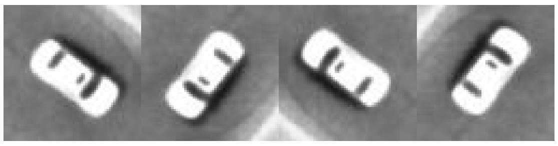


Figure 2.3: *Example of a rotated image*

it is using to make a classification.

For example Figure 2.3 contains images of the same car differing only by rotations of 90, 180 and 270 degrees. It is obvious to the human observer that they are the same car, but when the images are stored as arrays in a computer's memory there will be almost no correlation between the contents of the arrays what so ever.

It is therefore usual to perform a transformation of each image segment to a different physical representation in memory[5], such that in this representation the data stored is invariant under rotations, scalings and translations of the original image segment. Hence the program can compare the data to examples in its database without consideration of such variations in the image held in the segment.

Classifier

This module makes the decision about whether each of the segments it has been passed contains an object that is being searched for or not.

Data representation

It's worth noting that the invariance transforms are not the only place that the physical representation of the data in memory might be transformed. For example the preprocessor might perform an edge detection operation on the data or further down the pipeline a wavelet transform may be performed. The usual reason for such transforms is that the resultant representation provides some form of advantageous characteristics that make the processing being performed simpler. These characteristics will be discussed later.

[5]Examples of such representations are discussed later.

2.5 Instantiating the architecture

It was decided to adopt the architecture detailed in Section 2.4 for the system being designed, as it provided a good framework to work from. The research then turned to searching for possible methods of implementing the modules of this architecture. The options uncovered and the decisions made are now discussed.

2.5.1 Detector/Segmenter

It was clear from the literature on this topic that segmentation algorithms are in general domain specific. In other words a segmentation algorithm that works for one particular application is unlikely to be applicable to another. Therefore in general each application requires the development of its own segmentation algorithm tailored to its needs.

The description of the algorithm developed for this particular application is therefore left until Chapter 3, which deals with the system's implementation. There were however some useful concepts/building blocks discovered during the research and these are detailed here.

Segment growing

This is a common method of generating a simple segment. Its operation is analogous to the flood fill operation of a graphics package. Initially a single pixel (the seed pixel) must be specified in the area in which a segment is to be formed. A segment is then created containing only this pixel. It is then grown outwards as if a flood fill was occurring from the seed pixel until the growing area hits some kind of boundary. What constitutes a boundary is application dependent, however it might for example be that the intensity of the pixels reached are more than a certain number of intensity levels away from the seed pixel.

Segment aggregation

This is a general concept that refers to any technique that analyses a set of segments and merges any that together form a more useful segment than each of the component segments on their own. What kind of segment is deemed to be more useful is of course application dependent.

2.5.2 Invariance transforms

Research [2] suggested that two major methods were commonly applied in ATR applications to make the data representing a segment invariant under rotation, scaling and translation of the segment.

Fourier log polar invariance techniques

This technique, first implemented using purely optical systems [3], relies on the mathematical properties of the Fourier transform to obtain the invariance. Full details of the theory are given in the relevant texts [2, 3, 4, 5]. However the basic result is that an invariant representation of a segment can be obtained by the following process (which is also illustrated in Figure 2.4):

- Perform a 2D Fourier transform on segment's image data.
- Compute the power spectrum of Fourier transform.
- Map the power spectrum of the Fourier transform onto a log polar basis.
- Treating the log polar coordinates as standard cartesian coordinates, perform a further 2D Fourier transform on the data.
- Compute the power spectrum of this Fourier transform.

It must be noted that each segment that this set of transforms is performed on must contain only the object of interest and no background information. This is because background information would propagate through into the invariant representation, hence making this invariant representation vary depending on which background the object of interest is against. This obviously defeats the whole point of having an invarience transform.

High order neural networks (HONN)

This method [6] relies on a neural net being used as the classifier in the system. The basic principle is that extra layers of neurons are added to the input of the net. These layers conceptually allow the net to perform transformations on the data such that it has an invariant representation before it attempts to classify it in the final layers of the net.

This method can also be used in combination with selected stages from the Fourier log polar technique to produce a hybrid system [7].

The decision

Some of the research that has been performed on these systems [6, 7], suggests that using a HONN provides an approximately 10% better classification accuracy than Fourier methods followed by a classical three layer neural net. It is also suggested that they are more resilient to noise in the input image. However it was decided that Fourier invarience techniques would be used to obtain invarience.

This was because HONNs are relatively complex structures which need to be carefully designed and trained or they simply will not solve the given problem. Debugging a none functional net is then problematic due to the difficulties involved in understanding what is going on inside the net. Therefore as the author had no knowledge of neural nets at the start of the project, an engineering decision was made that a HONN was something that could get completely out of hand and not achieve any successful results by the end of the project.

Keeping the invarience transforms separate from the classifier would also aid implementation, in that each of the modules could be tested and validated independently before being connected together. This would not be possible with a HONN which would have to be tested as a whole.

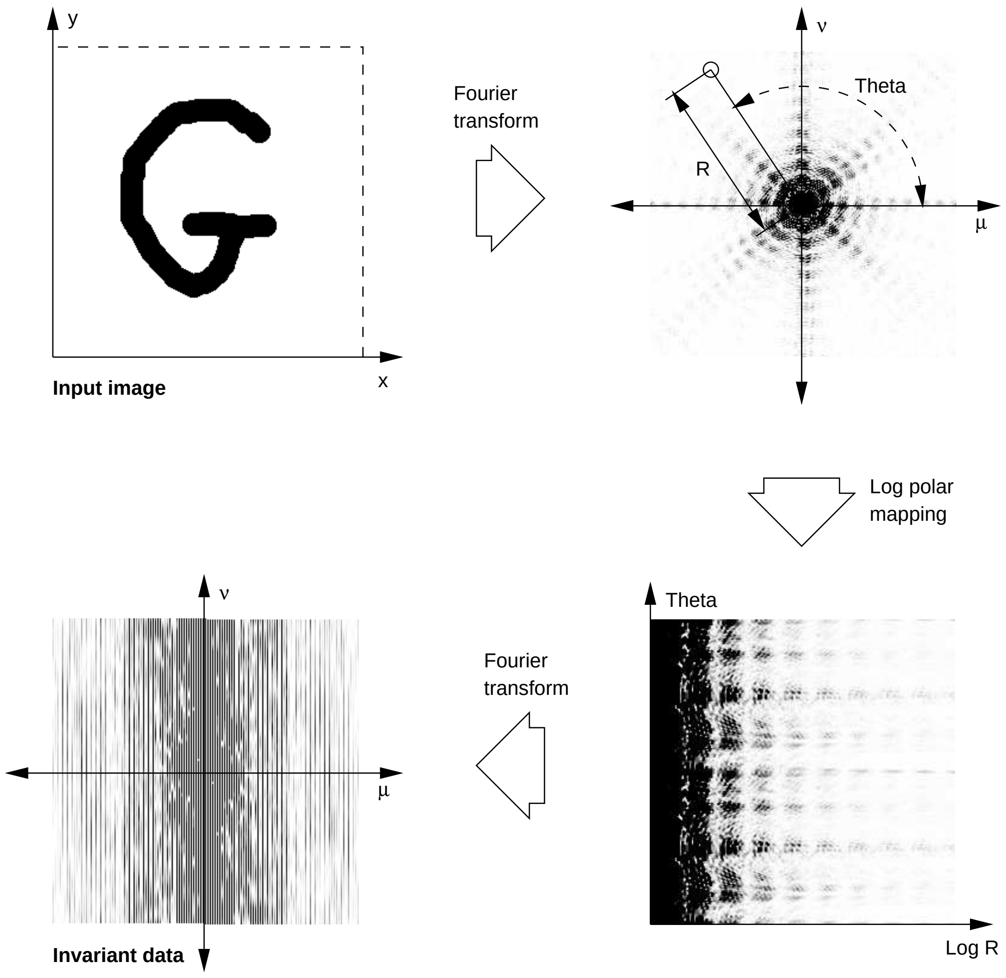


Figure 2.4: *Graphical illustration of the Fourier log polar invarience technique.*

2.5.3 Classification methods

Two major classification methods were deemed to be applicable to this problem. These were nearest neighbour techniques and neural nets used solely as a classifier (As opposed to the HONNs discussed earlier that achieved both invarience and classification). However before discussing these methods a few preliminary concepts need introduction.

Feature vectors

A feature vector is a list of parameters, representing an object, that is fed into the classifier for classification. So for example if the problem being tackled was the classification of rectangles into those that are squares and those that are simply rectangles, then the obvious feature vector to feed into the classifier would be one containing the width and height of the rectangle to be classified.

The process of converting the data held about the object to be classified into a feature vector is termed as feature extraction. So if the rectangles in the example above were input as images, then feature extraction would involve the measurement of the width and height of the rectangles.

However in real world problems, such as the one being tackled by this project, it is often the case that all the data that we have about the object is simply placed into the feature vector. This is because with complex real world data it is often difficult to pass the classifier all the data it needs to perform classification without giving it all of the data. Hence due to the shear complexity of the data to be output by the Fourier invarience transforms[6], it was deemed necessary to simply take this output as the input feature vector.

Training sets

The classifier in the rectangles example simply needed the knowledge that a square has a width that equals its height to make a classification. However in a real world problem it is common for a classifier to be fed a large set of examples of the things it is trying to classify, each tagged with their respective class[7]. It can then use these in some way to make classifications of unseen objects it gets presented with. This data set is known as the training set.

[6]See the example in Figure 2.4.

[7]A class is simply the group of things to which an object belongs. So in the rectangles examples there would be two classes - rectangles and squares.

Nearest neighbour techniques

In this style of classifier, a feature vector is taken to directly represent a location in n dimensional space, where n is the number of parameters in each feature vector. The idea is that if all of the feature vectors from the training set are plotted into the space. Then it becomes possible to classify a new object by plotting it into the space and working out which of the training set points it is closest to in terms of Euclidean distance. The class of the new object is then taken to be the same as the class of this nearest training set point. A variation of this is to take a majority vote on the class using the classes of the k nearest training set points. Where k is an arbitrary number.

Neural networks

This style of classifier attempts to simulate the pattern recognition abilities of the brain by using artificial representations of biological neurons. A comprehensive coverage of this topic is impossible in the space available here, so I refer the reader to any good book [8, 9] on the topic for the full details.

However in brief, a network of artificial neurons is constructed. These are then trained by repeatedly presenting them with all of the feature vectors from the training set. The error that they make in classifying each feature vector is used to alter weighting functions on the synapses[8] between each of the neurons and the activation threshold of the neurons themselves using an algorithm such a back propagation [8, chapters 4, 5 and 6].

Unfortunately if the net is trained for too long then it learns every noise perturbation in the training set. This noise generally makes the objects in the training set slightly a-typical of the objects to be classified by the net. This means that knowing about all these noise perturbations makes the net worse at classifying unseen objects than a net that hasn't been trained for so long and hence has a slightly more general knowledge of the training set.

[8]A synapse is a connection between two neurons.

To prevent this over generalisation a second, unseen, annotated training set known as the validation set is employed. This is not used to train the net. Instead at the end of each training pass through the training set (called an epoch) the total error in classifying the validation set is computed. If this falls after every epoch then the net is obviously getting better at classifying unseen data. However if after a period of decline the error starts to rise, then it is likely that the net has started to over generalise and hence is becoming worse at classifying unseen data. At this point training is stopped as the net has reaching its maximum performance.

A trained net can then be presented with previously unseen objects, which providing the net has actually managed to get a feel for the underlying problem[9], it should be able to correctly classify.

The decision

It was decided that a neural net would be used to perform classification. This was because neural nets intrinsically exhibit two properties necessary for this application that are more difficult to obtain with nearest neighbour techniques.

The first is that during training, a neural net generates a weight for each of the parameters that make up a feature vector. Hence some parameters have more bearing on the classification result than others. Nearest neighbour techniques are not able to do this by default, as all of the parameters of a feature vector have a direct and equally important bearing on the Euclidean distance between the points plotted in feature space. It is possible to perform an analysis of the training set to determine which of the parameters is of more importance and then generate weights for each of them. However neural nets give you this by default.

This weighting ability is essential if classification is to be perform directly on the raw data rather than on carefully extracted features of the data[10]. To understand why, consider the fact that the feature vectors contain the power spectrums of Fourier transforms. It is highly likely that a particular parameter of these feature vectors bears no relation to whether a feature vector is a car or not. Therefore giving this parameter the same importance as the other parameters will simply confuse the classification.

Secondly, during training a neural net classifier is able to bind together several input parameters in its first layer of neurons and set up a simple relation function on them, the results of which it can then use to make the classification decision in its final layer of neurons. This is important because a particular feature that is important for classification may move around between several different feature vector parameters for different objects of the same class. This is rather more difficult to obtain using nearest neighbour techniques, and hence is the main reason for selecting a neural net.

Which training set

Thought was then given to what exactly should be in the training set used to train the net, and how this training set was to be obtained. As the job of the net was to be to classify the output from the segmenter, it hence seemed logical that the training set should be the segments output from the segmenter, when run on example images. Each of the segments would then be tagged by a human user with its class, to generate an "annotated" training set. The hope was that the net would then be able to learn the difference between a car and the other objects that the segmenter output[11].

This therefore generated the requirement that it must be possible to execute the pipeline in two modes other than the linear mode that has been discussed so far. Firstly it must be possible to cut the pipeline just before the classifier and write out all of the segments generated to disc as a training set. It must then be possible for the user to annotate the segments with their classes and use this annotated training set to train the neural net. The data flow diagrams illustrating this are included in Figure 2.5.

2.5.4 Discontinuities equal information

One of the points that came out of the literature [5], was that classification should not be performed by looking at the pixel intensities of the segment to

[9]This is termed as having generalised.

[10]Such as the width and height in the rectangles example.

[11]Of course if the other objects were not sufficiently different from the cars then the classifier would never be able to tell them apart.

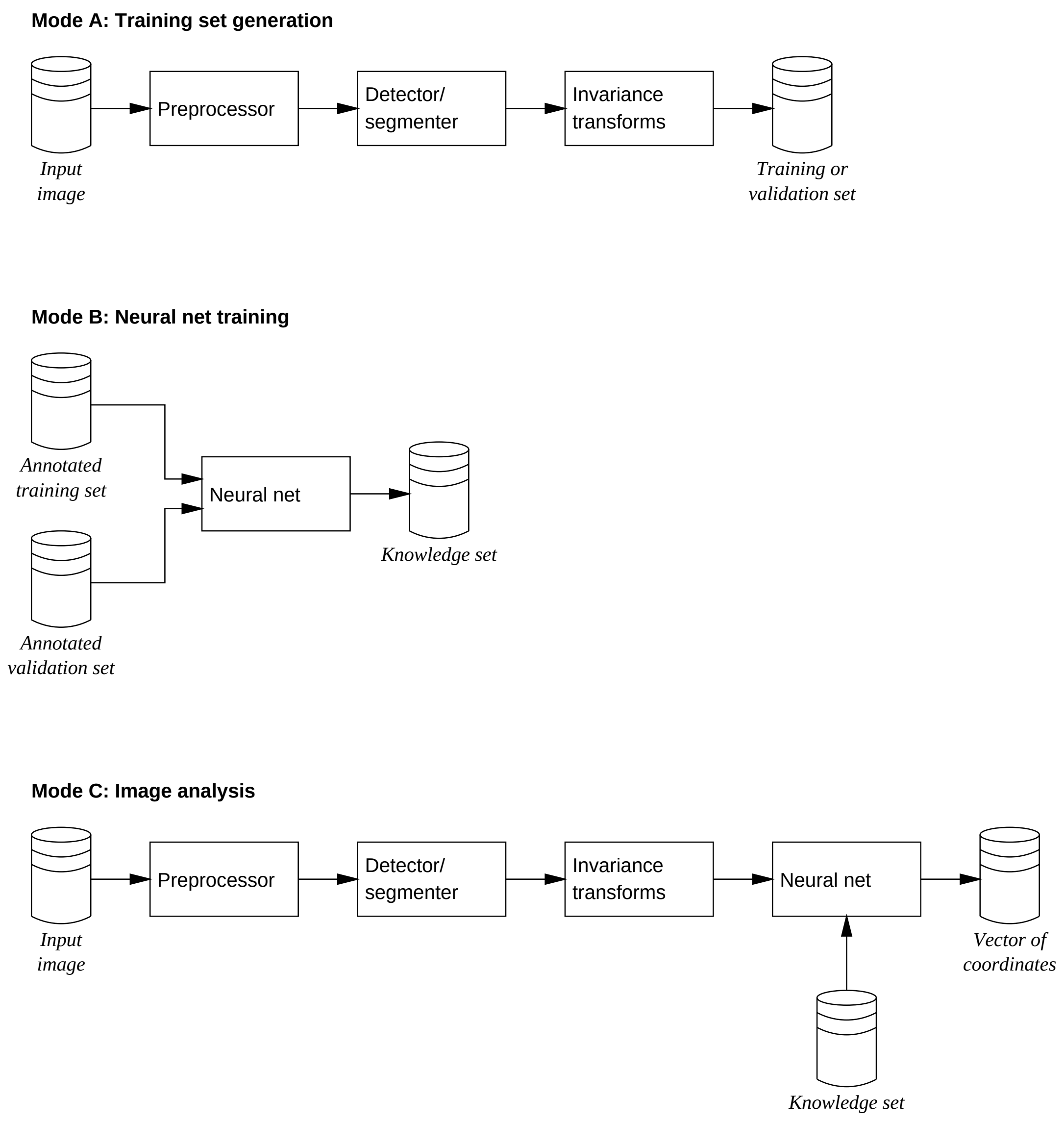


Figure 2.5: *Execution configurations of main pipeline*

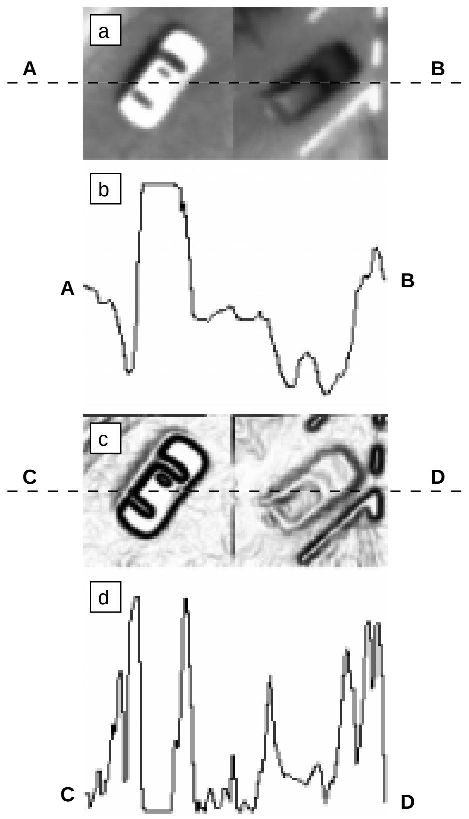


Figure 2.6: *Illustration of edge detection*

be classified. Instead it should be performed on the intensity differences between neighbouring pixels. These differences are more commonly known in computer science as the edge data or mathematically as the two dimensional first derivative of the image.

This is because "Discontinuities equal Information" [5]. What this means is that it's not really the intensity of each pixel in an image that allows you to identify a car as a car, but the differences in intensity between neighbouring pixels. E.g. it is the edges that are important.

To demonstrate this point take a look at Figure 2.6.a, this contains an image of a white and a black car on a grey background. Figure 2.6.b shows the intensity of the pixels along the scanline AB. It can be seen that the white car produces a large peak in the intensity and the black car a large depression. So while they are both cars they have completely different signatures in the intensity domain.

Figure 2.6.c is the magnitude of the edges in Figure 2.6.a. Figure 2.6.d is a plot of the same scanline, but this time taken across the edge data. Now accepting the fact that the data is noisy because this is a real image, it can be seen that both of the cars are represented by a pair of spikes. So by moving into the edge domain the signatures of cars with different intensity signatures have been given the same form.

To obtain this edge data it was therefore logical to place an edge detection module into the preprocessor. Hence segments output by the segmenter could contain edge information rather than pixel intensity information.

2.5.5 Noise filtering

The final major point that came out of the research was that the preprocessor should contain a filter to remove the noise that is present in the majority of images. Failure to do this could result in the segmenter or classifier being overwhelmed or confused by a spike on a single pixel.

2.5.6 Putting it all together

Taking all these design decisions and putting them together resulted in the core pipeline structure shown in Figure 2.7.

2.6 Testing

One advantageous side effect of the linear pipeline design is that it makes testing very easy. This is a result of it consisting of a number of totally independent modules, each for which can be independently excersised with test data and checked to make sure they are producing the correct results.

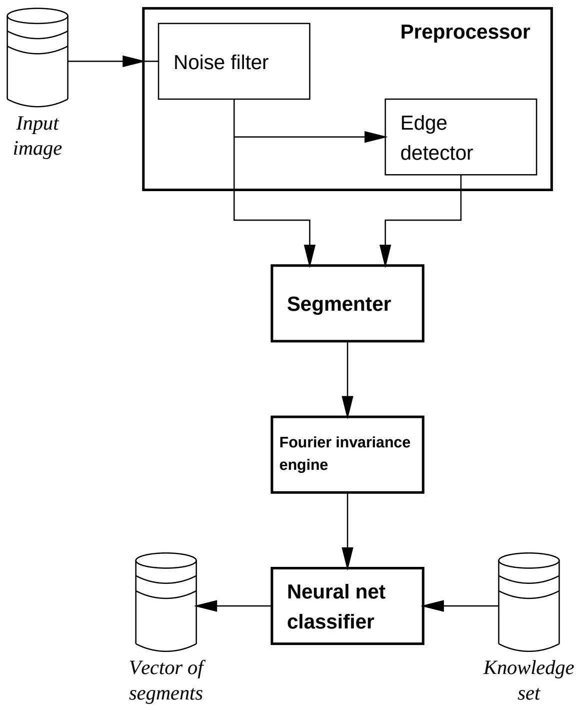


Figure 2.7: *The core structure of the system*

Chapter 3

Implementation

This chapter details the system that was constructed to implement the top level design detailed in Chapter 2. It contains both a detailed description of the system and some of the reasons it was constructed in this way.

3.1 Low level objects

Figure 3.1 contains a diagram of the low level utility modules and data structure objects that are part of the system implemented. Each of them is annotated with the methods it supports. The following sections contain a discussion of the more interesting ones.

InplaceVector & IndirectVector

One of the major data structures used in the system is that of the simple direct access list, or array. These are used to hold lists of things like pixels, segments and coordinates. An array is easy to create in C by making a simple call to `malloc`. The problem is that if you run out of space in the array then you start having to call `realloc` and handle error conditions. This code is then going to be duplicated everywhere that the list is written to, resulting in an error prone mess.

`InplaceVector` and `IndirectVector` hence provide encapsulated vector structures that simple resize themselves when necessary[1]. On construction they are passed a `default_size`[2] and an `inc_step`. Then when elements are added or removed using their `get`, `set`, `append` or `cut` methods, they ensure that their size is always the `default_size` plus the minimum number of `inc_step`s required to contain the elements they are holding. Note that this means that the vectors also contract, saving memory, as elements are removed. The reason for using size increments of `inc_step` is that calling `realloc` every time you need to add a single element would serverly damage performance and fragment memory.

The difference between the two vectors is that the `InplaceVector` stores elements one after another in a contiguous block of memory, whereas the `IndirectVector` only stores pointers to the elements that are placed into it. This difference is illustrated in Figure 3.2. Both of the vectors also have some interesting extra features. For example the `InplaceVector` has `push` and `pop` methods that allow it to be used as a stack. While the `IndirectVector` is selectively able to call the destructors of the objects it has pointers to when it its self is destroyed. This greatly reduces the code complexity of data structure deallocation.

The declaration of `InplaceVector` is included in Appendix A.

BitMatrix, IntegerMatrix & DoubleMatrix

These structures are simply handy wrappers of two dimensional arrays of bits, `int`s and `double`s respectively. They perform both memory handling and range checking on arguments. The `BitMatrix` is interesting in that it only uses one bit to store each element, rather than the entire machine word per element found in most naive C bit array implementations. The `DoubleMatrix` is able to store and load its contents to and from a file, hence facilitat-

[1]The term vector is taken here to mean an array that dynamically resizes its self.

[2]A size of 1 would mean enough space to store a single element, rather that 1 byte of memory.

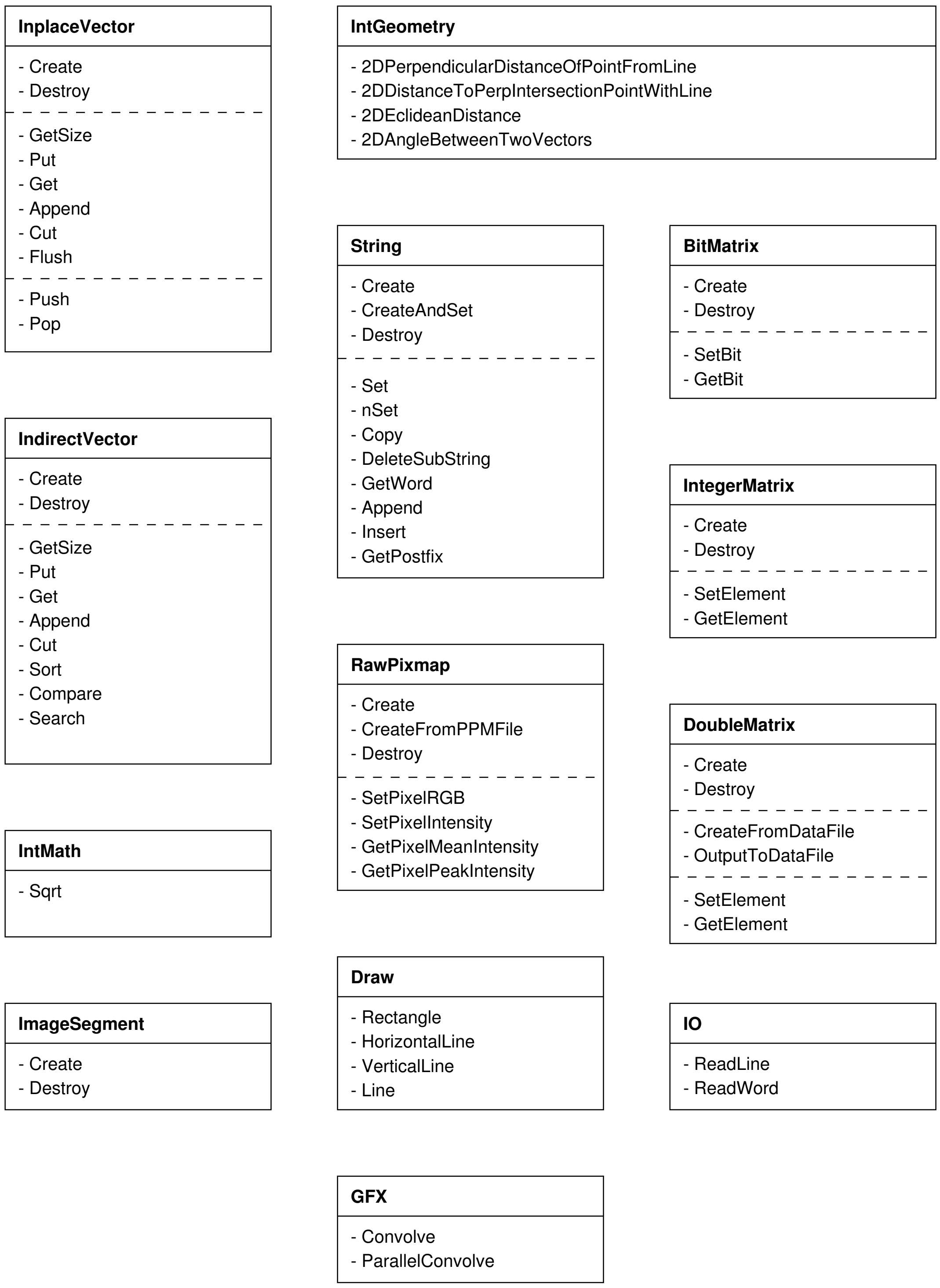


Figure 3.1: *Low level utility modules and data structure objects.*

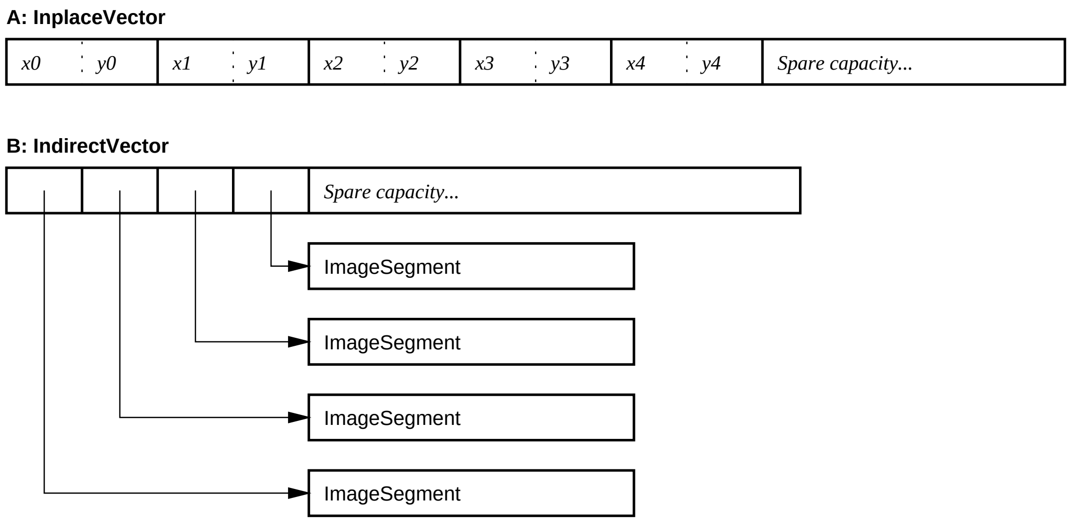


Figure 3.2: *A: The memory map of an* `InplaceVector` *that contains five XY coordinates. B: The memory map of an* `IndirectVector` *that contains pointers to four* `ImageSegment` *structures.*

ing the storing and retrieval of the feature vectors that make up the training sets.

RawPixmap

This is a wrapper for 24bit RGB image data. Individual pixels can be read and written and the object is capable of reading from and writing PPM format [10] image files.

ImageSegment

This is a simple C `struct` that is used to store data about a segment that is defined on an image. Its definition is given in Appendix B. It is designed to store three important sets of data about a segment. These are:

- A list of the XY coordinates of the pixels of the image covered by the segment.
- Statistics about the segment. These include its width, height, mid point, average colour and idensity[3]. These statistics are included in the segment definition so that procedures that process the segment after it has been created don't have to keep recomputing them from the list of pixels covered.
- Status flags. These are used when the segment is being processed to indicate whether the segment has already been discarded or is part of an object that has been classified (and hence can't be part of another object).

[3]This is the inverse density, which is defined as $(width * height)/area$, where $area$ is the number of pixels in the segment. Its purpose is to distinguish compact, dense segments which have an idensity close to 1 from segments which are fibrous or consist of a ring of pixels and hence have an idensity much greater than 1.

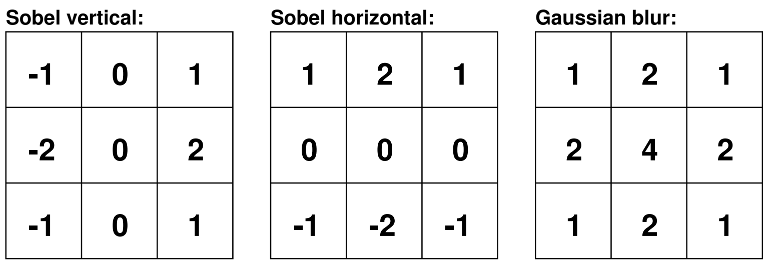
GFX

In the discrete two dimensional case applicable to image processing, the mathematical operator of convolution is defined as:

$$out(x,y) = \sum_m \sum_n kern(m,n) \cdot in(x-m, y-n)$$

Where *in* is the input image, *out* is the output image and *kern* is a small convolution kernel to be applied. By the application of appropriate kernels it is possible to perform processing operations such as edge detection and noise filtering. For example Figure 3.3 contains Sobel kernels that perform detection of horizontal and vertical edges and a Gaussian filter that performs a blurring operation.

The GFX module contains a very powerful convolution function called `ParallelConvolve`. The declaration of this function is included in Appendix C. The function has the following features



Sobel vertical:

-1	0	1
-2	0	2
-1	0	1

Sobel horizontal:

1	2	1
0	0	0
-1	-2	-1

Gaussian blur:

1	2	1
2	4	2
1	2	1

Figure 3.3: *A selection of common convolution kernels.*

which are not common to standard implementations of the convolution operation:

- It is capable of performing the convolution in place. E.g. the output image can be the same block of memory as the input image and a minimal quantity of workspace is used in the computation.

- As well as reading in and outputting an image. It can read in and output a sign plane which contains the sign of each of the pixels in the image. This is useful for performing an edge detect from which you need the direction of the changes in intensity as well as the magnitudes. Indeed it is essential to have this information in order to perform a Laplacian edge detection operation.

- It is capable of handling kernels of any shape and size, however it is inefficient for kernels above approximately 5 by 5 [5].

- It is capable of applying more than one kernel at a time and combining the results in a variety of ways. This is useful for performing operations such as a Sobel edge detection that finds both horizontal and vertical edges. Normally the horizontal kernel from Figure 3.3 would be applied to the image and the result stored in a block of temporary workspace. The vertical kernel would then be applied and the results for the two applications then merged. However with the `ParallelConvolve` function it is possible to simply pass in both the kernels and all of these operations are performed simultaneously, in place and with sign planes if necessary.

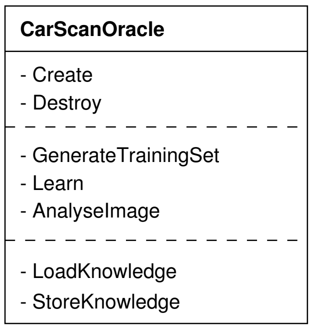


Figure 3.4: *The* `CarScanOracle` *object*

3.2 Top level structure

Exported interface

In accordance with the original specification the entire processing pipeline was wrapped in a black box library. This library exports the single object illustrated in Figure 3.4. The function of each of its methods is as follows:

- *Create:* Creates an untrained `CarScanOracle` object.
- *Destroy:* Destroys a `CarScanOracle` object.
- *GenerateTrainingSet:* Accepts an example image, the name of a directory to write an unannotated training set into and a collection of user settings from the caller. It passes the image through the pipeline configured in mode A - the training set generation mode from Figure 2.5. Hence writing out the training set produced from the image to the specified directory.
- *Learn:* Accepts the name of a directory containing an annotated training set and the name

of a directory containing an annotated validation set from the caller. It passes these into the pipeline configured in mode B - the neural net training mode from Figure 2.5. Hence training the neural net.

- *AnalyseImage:* Accepts an image to be analysed and a collection of user settings from the caller. It passes the image through the pipeline configured in mode C - the image analysis mode from Figure 2.5. Hence the image is analysed and a vector of segments that cover all of the cars in the image is returned to the user. It is also worth mentioning that this is actually returning more useful information than was originally specified, as it returns the list of pixels that each car occupies, wrapped in an `ImageSegment`, rather than just the XY coordinates of each car.
- *LoadKnowledge & StoreKnowledge:* Loads and stores the knowledge held in the `CarScanOracle`'s neural net.

Data flow

The flow of data inside this object in each of the three processing modes is illustrated in Figure 3.5. The key point from this diagram is that although conceptually the main pipeline is linear, it is not implemented in a purely linear fashion. This is for performance reasons, because if the segmenter was to complete its processing and then pass on a large list of segments to the downstream modules, then it would require a huge quantity of memory to store this list. Therefore instead, every time the segmenter generates a segment it is immediately passed to the downstream modules to either be classified or written out as part of a training set. If the segment is classified and is found to contain a car then the segmenter places it into the output vector of segments that contain cars that is to be returned to the caller, otherwise it discards it immediately.

3.3 Preprocessor

Noise filter

The noise filter is implemented simply by applying the Gaussian blur filter from Figure 3.3 using the `Convolve` function from the `GFX` module. This has the effect of rounding off any noise spikes in the input image but not completely removing them.

Edge detector

This sub module is implemented by simultaneously applying the two Sobel kernels shown in Figure 3.3 to the image input, discarding the signs and taking the maximum to the two results. This is achieved using a single call to the `ParallelConvolve` function in the `GFX` module. The output image therefore contains the magnitudes of the edges at each location irrespective of the direction of the edge.

3.4 Segmenter

Inputs and outputs

As illustrated in Figure 3.5, the segmenter accepts as input the blurred image and the edge image as `RawPixmaps`. If executing in image analysis mode, it outputs a vector containing `ImageSegment` definitions of all the segments that were deemed by the classifier to contain cars. However if executing in training set generation mode it has no output, as the segments generated are written out to disc after the Fourier invariance transforms. Each segment that is generated is passed to the Fourier invariance transforms wrapped in an `ImageSegment` definition.

Method overview

A number of experiments led to the conclusion that it is not possible to create a useful segmentation system that directly produces segments that might contain cars. This is because cars are so diverse that any heuristic that finds all cars also appears to find everything else in the image, which defeats the point of the segmenter. However further experimentation led to the development of a two stage process that is capable of locating all the cars in the image without outputting everything. Stage one is to locate objects that might be components of cars, such as the boot, roof and bonnet. Stage two is an aggregation phase where a scan is performed for clusters of these candidate components that are stereotypical of the presence of a car.

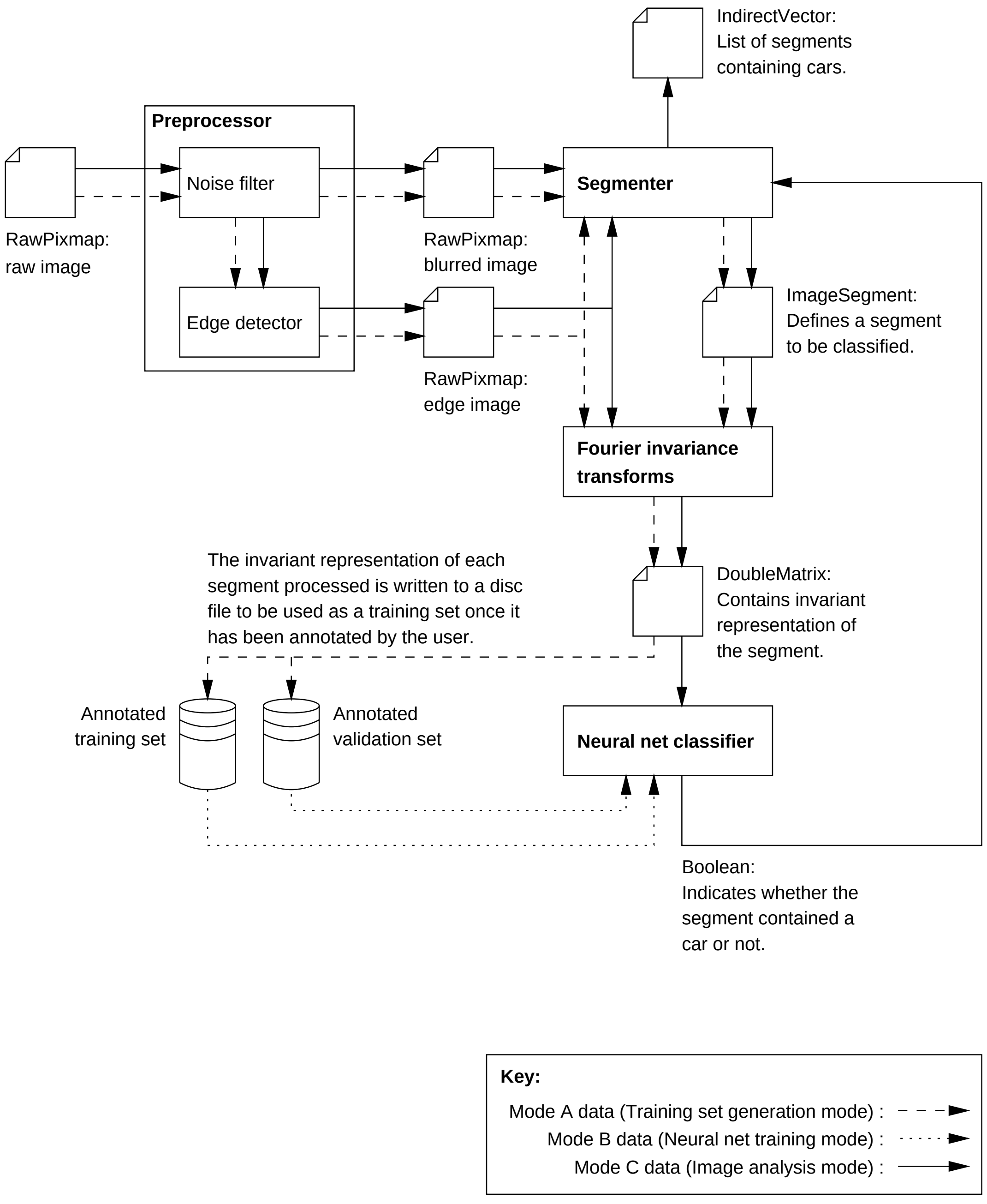


Figure 3.5: *Data flow within the* `CarScanOracle` *object*

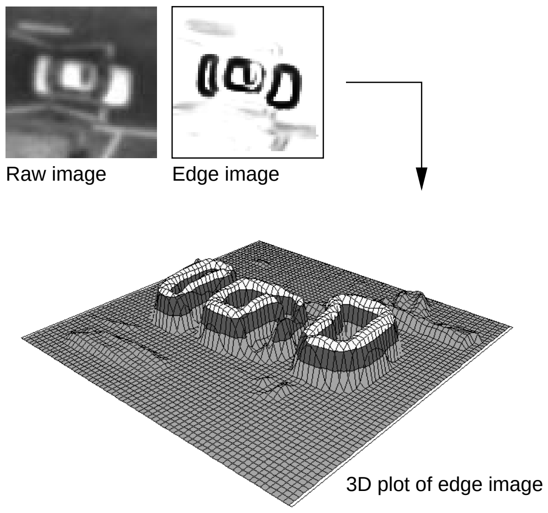


Figure 3.6: *Some different representations of an image containing a car.*

Component location theory

After looking at car components in a number of different representations, such as those shown in Figure 3.6, it became clear that they generally have two things in common:

- Most components are areas of relatively continuous colour surrounded by areas of different colours.
- The center of a component is usually a local minima in the edge domain.

This led immediately to a method for finding car components using segment growth[4]. E.g. if segment growth is started from all local minima and stopped when the pixels reached have a colour that is deemed to be significantly different from the seed pixel, then a collection of segments that represent car components is produced. In this document these segments will be termed as “micro segments”.

Of course the definition of a car component that is being used is so general that other objects such as trees, bits of road and houses will also be output as micro segments. Hence the need for the aggregation phase to work out which of the micro segment groups actually represent cars.

[4]See Section 2.5.1.

Aggregation theory

It was determined that cars have two different micro segment signatures. They are either a single micro segment that is approximately rectangular and of the correct length and width to be a car[5], or they are a cluster of micro segments that together cover an area that is again approximately rectangular and of the correct length and width to be a car. Both these signature types are illustrated in Figure 3.7. The aggregation system therefore simply needs to look for these signatures to find things that are possibly cars.

Note that for all the cars in Figure 3.7 that have signatures that are clusters of micro segments, regardless of how many segments there are in the cluster, there is always a segment that spans across the whole back of the car and another that spans across the whole front of the car. These lateral spanning segments are a feature common to almost 100% of cars that are made up of a cluster of micro segments, and are generally the boot and the bonnet of the car. As will be seen later this fact is useful when trying to perform aggregation.

Internal pipeline

The implementation of the segmenter hence has three conceptual stages to its internal pipeline. Firstly it generates the micro segments. It then performs aggregation. Then finally it converts any aggregates found into individual segments that represent possible cars to be passed downstream for classification. The data flow and structure of the pipeline that achieves this is illustrated in Figure 3.8. The next few sections describe the function of each of its modules.

Micro segmenter

This module is responsible for generating the micro segments. It outputs two data structures, the micro segment list and the micro segment map.

The micro segment list is an `IndirectVector` containing a list of `ImageSegments`, each of which

[5]The size of a car in the image can trivially be computed from the altitude at which the image was obtained and the lens size of the camera. This system is not concerned with this computation and so reads in the approximate size of a car from the caller. Hence the size can either be computed or entered directly by the user.

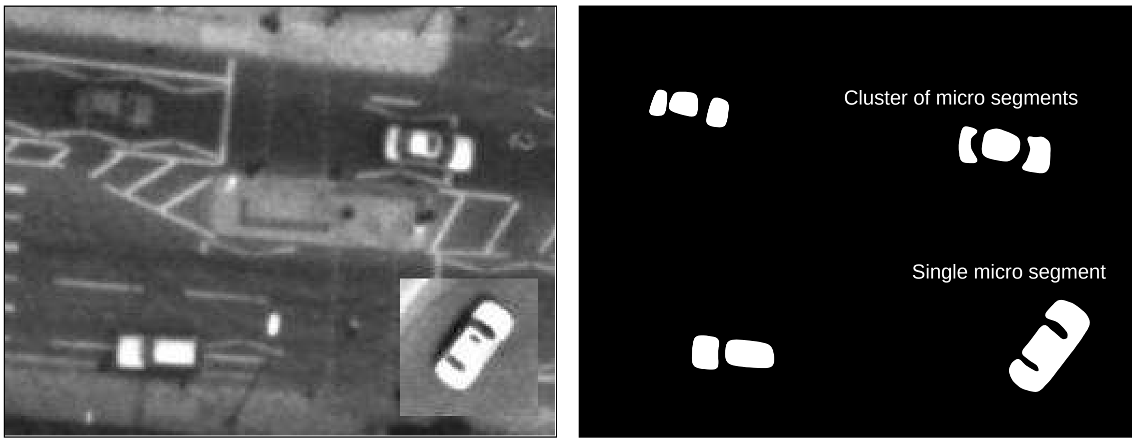


Figure 3.7: *An illustration of the different micro segment signatures of cars.*

defines a micro segment. However while the micro segment list holds definitions of all the segments generated by the micro segmenter, it is not efficient to use it to work out which micro segment covers a particular pixel of the input image[6]. To do this would require a linear search through all the pixel lists of all the `ImageSegment`s held in the micro segment list.

Therefore the location of a micro segment in the micro segment list is defined as the ID of the micro segment and a pixel to segment lookup table is then generated. This lookup table, the micro segment map, is an `IntegerMatrix` of the same dimensions as the input image. Each element contains the ID of the micro segment that covers the corresponding pixel or `-1` if there is no segment covering the pixel.

The micro segmenter implements the component location technique discussed earlier. The algorithm designed to estimate the local minima is given in Figure 3.9 and the algorithm to grow the micro segments from each local minima is given in Figure 3.10. Each segment growth that occurs generates an `ImageSegment` with all its statistics appropriately filled in. This `ImageSegment` is then added to the two output data structures.

The reason for the algorithm in Figure 3.9 only estimating the location of local minima is that finding local minima is a time consuming process. Therefore it simply starts segment growth from all the zero points of the edge image, which are by definition local minima. It then starts segment growth from all points with a value of one that have no segment covering them and so on. This doesn't exactly achieve the effect of growing segments only from the local minima. For example it tends to produce ring segments around the rims of the volcano like structures in Figure 3.6. However this doesn't matter as the results are close enough, faster to obtain and the aggregation routines that follow can actually make use of these extra segments.

A number of other local minima estimation techniques were also experimented with, but this one tended to give the best set of micro segments to work with during aggregation.

An example of the typical output generated is given if Figure 3.11. The different tones of shading represent different micro segments. It should be visible from this that the majority of the components of the cars have been detected.

Segment culler

It can be seen from Figure 3.11 that the micro segmenter outputs segments that are obviously not components of cars. For example in Figure 3.11 the road that the cars are sitting on has been output as a segment. The purpose of the culler is therefore to discard any of the micro segments that are obviously not relevant.

It achieves this by scanning through the micro segment list and setting the discarded flag on any

[6]Micro segments are defined as being mutually exclusive of one another, hence there exists a mapping from each input image pixel to zero or one micro segments.

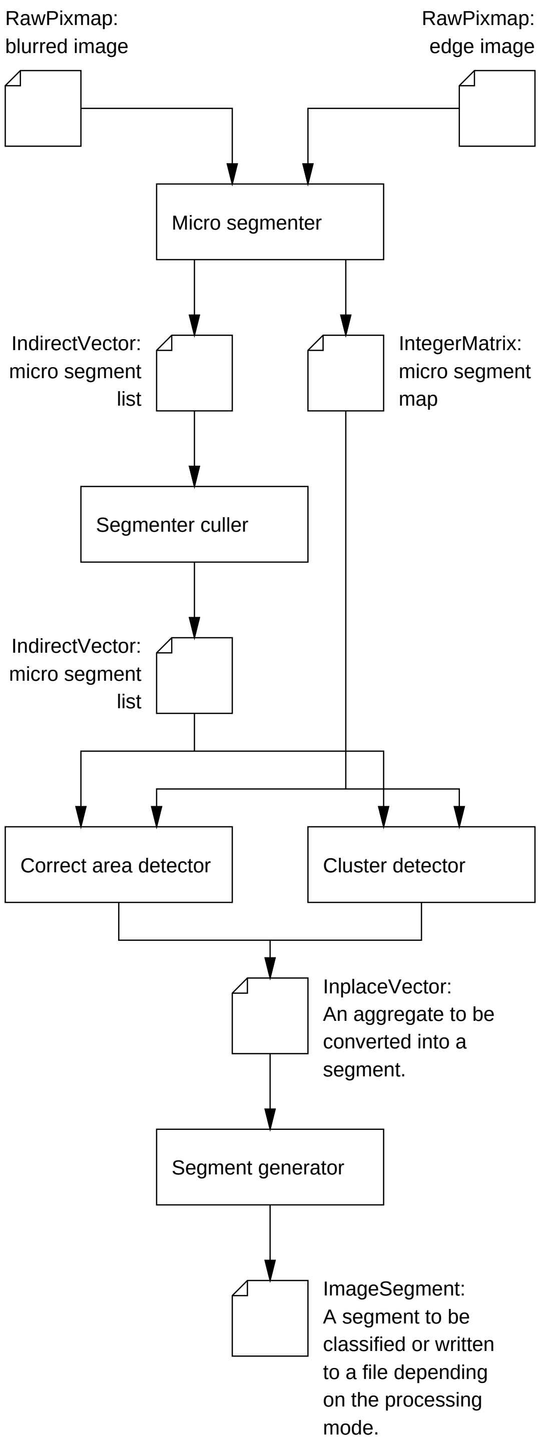


Figure 3.8: *The internal pipeline of the segmenter.*

```
For gradient = 0 to gradient_threshold
  do
  For every pixel in input image
    do
    If value stored for pixel in the edge image
       equals gradient and pixel is not already
       part of a micro segment
       then
       - Commence a region growth from pixel
       endif
    done
  done
```

Figure 3.9: *Simple local minima estimation algorithm.*

```
- Generate an empty pixel stack
- Generate a new micro segment

- Push seed pixel onto stack

While pixel stack not empty
  do
  - Pop pixel from top of pixel stack

  If difference in colour between pixel and seed
     pixel is less than user threshold and pixel
     is not already part of a micro segment
     then
     - Add pixel to micro segment
     - Push the 8 pixels surrounding the pixel
        onto the pixel stack
     endif
  done

- Write statistics into micro segment
```

Figure 3.10: *Segment growth algorithm.*

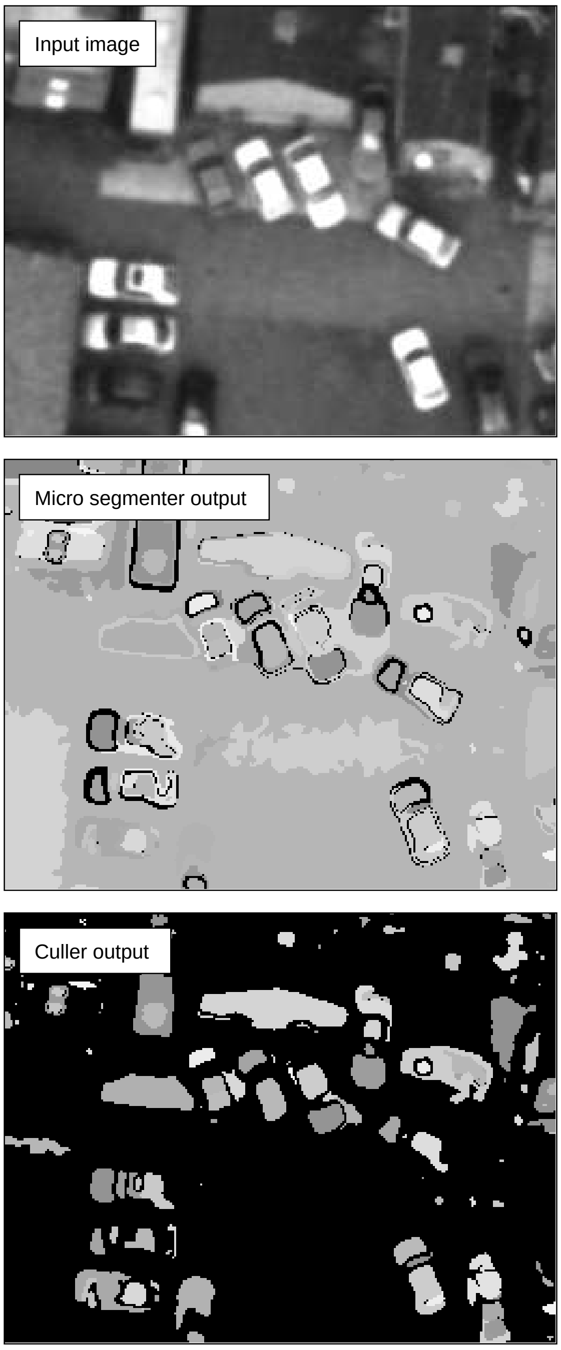


Figure 3.11: *Example of output from the micro segmenter and culler.*

of the segments that have a width or height greater than 1.5 times the approximate length of a car. The results of this are also shown in Figure 3.11, the black areas representing pixels with no micro segments covering them. From this it can be seen that the components of the cars have become very obvious.

Aggregation

To perform the micro segment aggregation the segmenter pipeline splits into two, a branch for each of the different micro segment signatures a car can have. The two branches are executed serially one after the other. When a branch locates something that it thinks might be a car it outputs a list of the IDs of the micro segments that might be the car, this is termed here as an aggregate, which is then passed on down the segmenter pipeline for conversion into a true segment and then on to the invariance transforms etc.

Correct area detector

This module tries to locate the micro segments that on their own could represent a car. It does this by scanning through the micro segment list looking for segments that have not been discarded, have an idensity less than three[7] and have and area greater than a caller specified threshold. Any segments that meet these criteria are passed out as a single segment aggregate.

Cluster detector

This module tries to locate the clusters of micro segments that could represent cars. The ideal cluster detection algorithm would be to locate clusters of micro segments that will fit inside a rectangle of the size of a car and fill more than a threshold percentage of the rectangle. However it was decided that the mathematical and computational complexity of trying to fit an arbitrarily oriented rectangle to a cluster of micro segments, where there are a large number of possible combinations of micro segments, makes the method infeasible.

Fortunately it was realised that the feature noted earlier that cluster cars have segments that laterally

[7]This is a purely heuristic number that was found to be about the upper bound for cars consisting of a single micro segment.

span their back and front can be used to make the problem tractable. In brief the operation of the implemented cluster detector is as follows (How it performs some of these tasks will be explained in a moment):

- Initially it finds a pair of micro segments which look like they could be the front and back of (the ends of) a car.
- It then derives from them the location of the bounding rectangle that the car would have if the pair actually were the ends of a car.
- It then tests the hypothesis that the pair are the ends of a car by scanning the bounding rectangle for other micro segments that could represent components of the car, such as the roof and the windows. If a sufficient quantity of other components are found then it accepts the pair and the other components as a car, wraps them together as an aggregate and passes them downstream for further processing. Otherwise if an insufficient number of other components are located, then it concludes that the pair are not the ends of a car and moves on.
- It then repeats this process until there are no more pairs of segments that look like they could be the ends of a car.

Some notes on this process are:

- To locate pairs of segments which look like they could be the ends of a car, the cluster detector looks for pairs where the segments are both about the right size to be a bonnet or boot, approximately the correct distance apart and of similar colour[8].
- To derive the location of a car bounding rectangle from a pair of end segments the cluster detector makes use of the fact that if the segments are the ends of a car then they also generally laterally span the car. This means that their midpoints will lie on the longitudinal centerline of the car. The sides of the bounding rectangle are therefore parallel to this centerline at half a car width on either side. The ends of the rectangle are then lines that are perpendicular to the centerline that push hard up against the two end segments.

[8]Although this is not rigorously enforced to allow for multi coloured cars.

Segment generator

The purpose of the segment generator is to convert an aggregate that is just a list of micro segments into a single segment that can be passed to the invariance transforms. As a car when viewed from above in real life is a convex object the segment generator computes the convex hull of the aggregate. It then converts the hull into an `ImageSegment` by performing a polygon scan conversion operation to compute the pixels that the hull encloses. Some of the special cases involved in polygon scan conversion are helpfully avoided as the hull is convex. Finally the segment is expanded by a couple of pixels in all directions to ensure that the car is completely included in the segment.

The segment definition is then passed on down the main pipeline to the invariance transforms.

3.5 Fourier invariance transforms

Inputs and outputs

As illustrated in Figure 3.5, the Fourier invariance module accepts as input a segment defined by an `ImageSegment` and the edge image as a `RawPixmap`. Its output is a `DoubleMatrix` which contains a rotation, scaling and translation invariant representation of the segment.

Internal pipeline

The data flow and structure of the internal pipeline of this module is illustrated in Figure 3.12.

Masker and range normalisation

The invariance transforms need to work on the data held in the region of the edge image defined by the `ImageSegment` passed in. However the `ImageSegment` only contains a list of the XY coordinates of the pixels that make up the segment not the data its self. The masker therefore cuts out the region of the edge image defined by

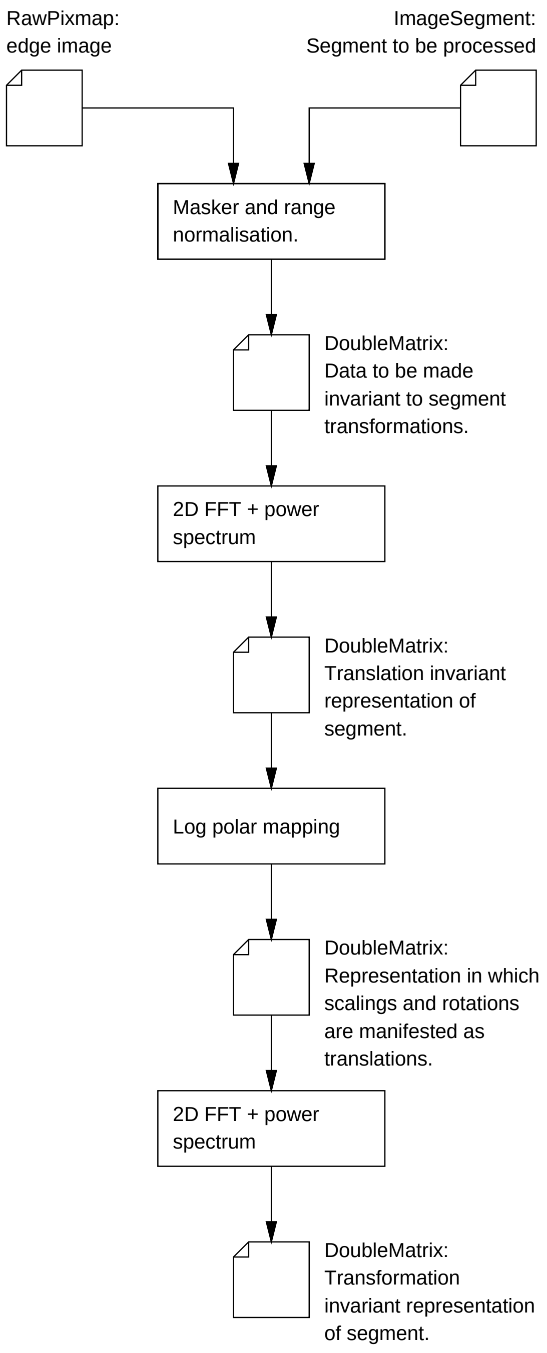

RawPixmap:
edge image
ImageSegment:
Segment to be processed
Masker and range normalisation.
DoubleMatrix:
Data to be made invariant to segment transformations.
2D FFT + power spectrum
DoubleMatrix:
Translation invariant representation of segment.
Log polar mapping
DoubleMatrix:
Representation in which scalings and rotations are manifested as translations.
2D FFT + power spectrum
DoubleMatrix:
Transformation invariant representation of segment.

Figure 3.12: *Internal pipeline of the Fourier invariance module.*

the `ImageSegment` and passes it on down the invariance transform pipeline as a 64 by 64 element `DoubleMatrix`. This process is illustrated in Figure 3.13.

The shift to a real valued `DoubleMatrix` is necessary because the Fourier transform and classifier routines all operate on real valued data rather than discrete pixel values. It is of fixed 64 by 64 size because all the feature vectors that are passed to the classifier must be the same size, hence it is logical to fix the size at this point and gain performance through the Fourier transforms by using matrix dimensions of a power of 2 which are optimal for an FFT. Segments that are too large to fit into the 64 by 64 matrix are scaled to fit. This can be performed without loss of generality as the data is about to be made invariant to scalings.

Finally this module performs range normalisation. What that means is that each `DoubleMatrix` object output has its elements scaled so that they range from 0.0 to 1.0. This means that a dark car on a dark background which will hence have very weak edges will have the magnitude of its edge increased so that is has the same representation as a bright car on a dark background which will have very strong edges.

2D FFT + power spectrum

The FFTs are implemented simply by making calls to the MIT *"Fastest Fourier Transform in the West"* (FFTW) library. This provides the exceptionally high performance necessary to perform the large number of Fourier transforms required to process all of the segments output by the segmenter. A power spectrum computation is then performed on the results.

Log polar mapping

The mapping of the output from the first Fourier transform onto a log polar basis utilises super sampling to ensure that as accurate as possible representation of the data is obtained.

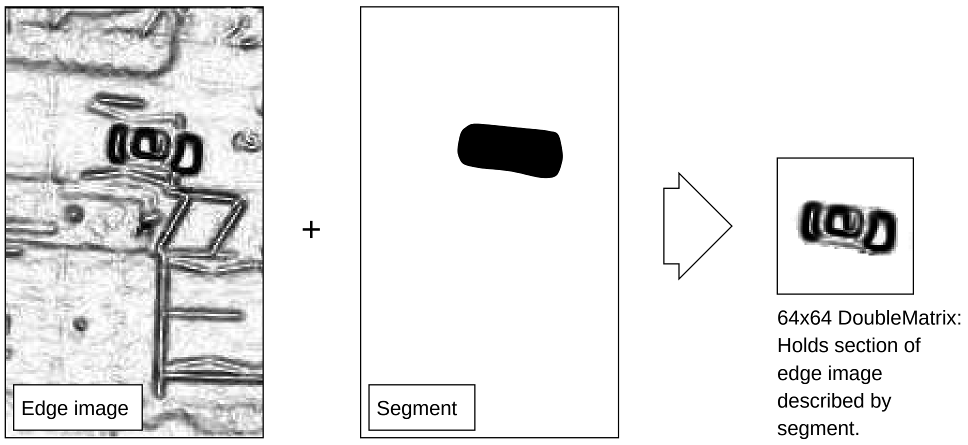

Edge image
+
Segment
64x64 DoubleMatrix: Holds section of edge image described by segment.

Figure 3.13: *Illustration of masker operation.*

3.6 Neural net classifier

Inputs and outputs

As illustrated in Figure 3.5, the classifier module accepts as input a transformation invariant representation of a segment in a `DoubleMatrix`. It then classifies the segment and outputs a boolean that specifies whether the segment contains a car or not.

Overview

The classifier is implemented using the classic three layer neural net architecture illustrated in Figure 3.14. The implementation has the following features[9]:

- Each element of the input `DoubleMatrix` is taken as a parameter of the feature vector fed into the net.
- The net has a single output neuron that outputs the likelihood of the feature vector being fed into the net being a representation of a car.
- The number of neurons in the hidden layer is configurable by the user, hence it can be tailored to the complexity of the user's data.
- Training is via gradient descent methods [8, chapter 4], using back propagation [8, chapter

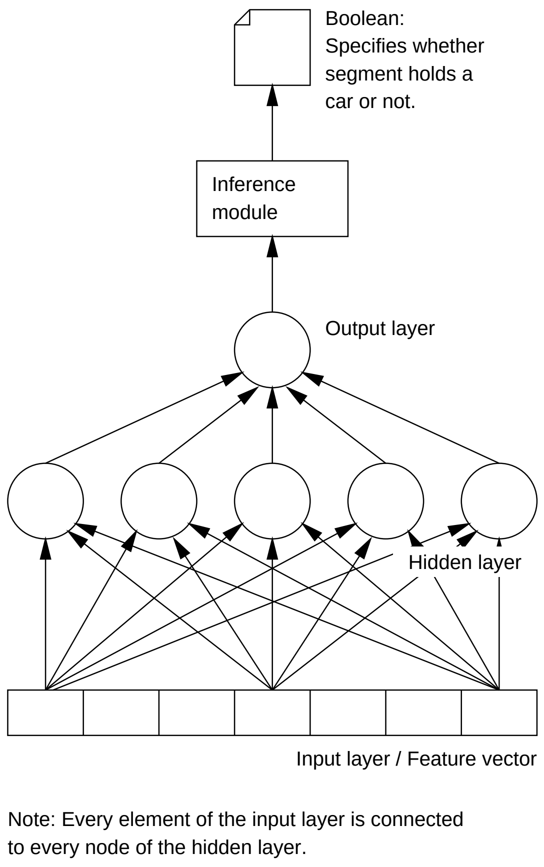

Boolean: Specifies whether segment holds a car or not.
Inference module
Output layer
Hidden layer
Input layer / Feature vector
Note: Every element of the input layer is connected to every node of the hidden layer.

Figure 3.14: *Architecture of neural net classifier.*

[9]Some of the points in this list assume a good familiarity with neural net techniques. The reader is asked to refer to a good text [8, 9] on the subject for more details.

6] to feed output errors back to the hidden layer nodes.

- The output of each node is fed through a sigmoid function to allow a smooth gradient descent [8, page 19].
- The net is trained by use of the “online” technique in which the synapse weights and neuron activation thresholds are updated after presentation of each training set feature vector.
- The training of the net makes use of simulated momentum to allow the gradient decent to cross local minima [8, section 6.5].
- Due to the sigmoid function the single output layer node outputs a number between 0.0 and 1.0. This is converted into a boolean by the inference module which is in fact simply a configurable threshold. If the output from the output node is greater than the threshold then the input is deemed to be a car and the inference module outputs `true`, otherwise it outputs `false`. After the net has been trained the threshold is set simply by selecting the value that gives the best tradeoff between the number of cars missed and the number of other objects incorrectly identified as cars.

Data structures

Training a neural net is a very computationally expensive task. Therefore the data structures used to implement the net were influenced purely by performance considerations. To this end the data structures consist solely of two linear arrays of doubles. One of these arrays stores the synapse weights and activation thresholds of the hidden layer neurons[10], the other stores the synapse weights and activation thresholds of the output layer neurons (Which in this case is only a single neuron). Two other arrays are then used during training to store the rate at which each of the weights in both the layers are changing. These rates are known as weight velocities and are used in computing the momentum of a weight. The format of these arrays is illustrated in Figure 3.15.

[10]The implementation makes use of the mathematical properties of the artificial neurons that allows an activation threshold to be treated as a weight on an extra synapse that has it’s input permanently set to a value of 1.0

The use of linear arrays rather than say an object for each neuron, allows fast pointer arithmetic transversal of the structures during evaluation and training. This provides a huge performance advantage over making multiple indirect reads and writes to the elements of objects.

Implementation

The evaluation and training/back propagation algorithms are not presented here as they are implementations of standard textbook [8, 9] algorithms. However for the interested the feature vector evaluation function is included in Appendix D.

3.7 Facilities for testing

Most of the objects in the system have extra debugging methods that allow their internal state to be dumped to disc.

3.8 Third party code

The third part code and libraries used in the system are:

- *Clib:* The standard C library.
- *FFTW:* The MIT “Fastest Fourier Transform in the West” library. Used for performing two dimensional FFTs.
- *Integer sqrt:* An Integer square root routine from Dr. Dobb’s Journal [11].

3.9 Source statistics

The statistics of the source code produced to implement the system are:

```
Lines of source      : 15654

  Lines in headers   : 3541
    Comments         : 3099
    Code             : 442

  Lines in body      : 12113
    Comments         : 2374
    Code             : 9739
```

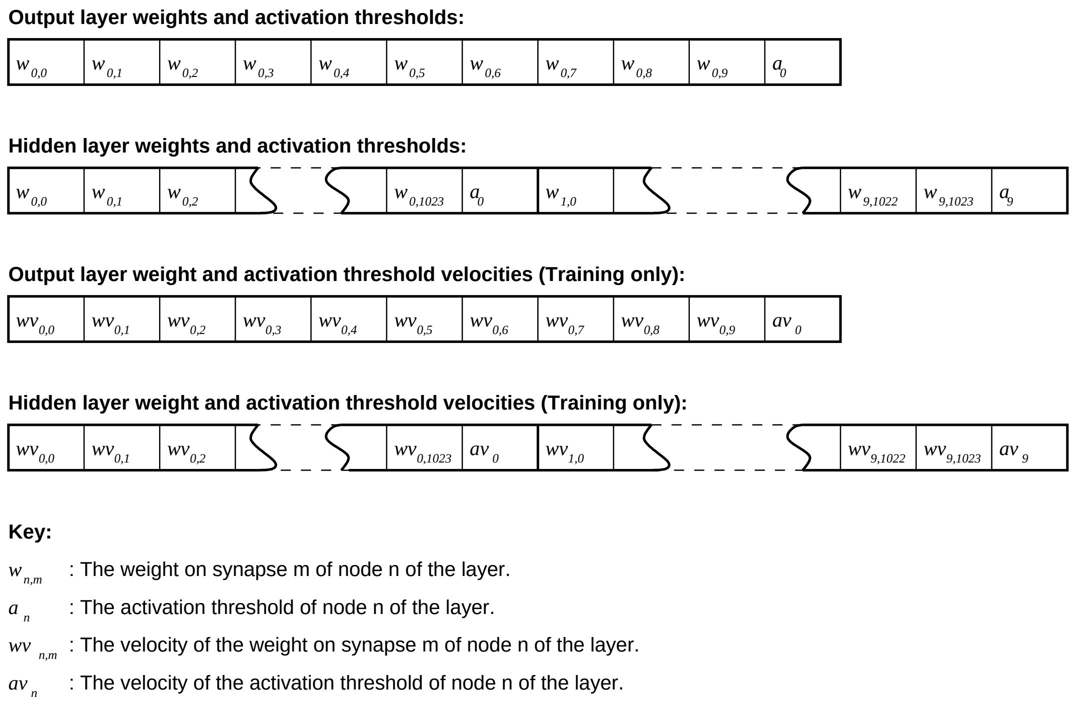

Output layer weights and activation thresholds:
$w_{0,0}$ $w_{0,1}$ $w_{0,2}$ $w_{0,3}$ $w_{0,4}$ $w_{0,5}$ $w_{0,6}$ $w_{0,7}$ $w_{0,8}$ $w_{0,9}$ a_0
Hidden layer weights and activation thresholds:
$w_{0,0}$ $w_{0,1}$ $w_{0,2}$ $w_{0,1023}$ a_0 $w_{1,0}$ $w_{9,1022}$ $w_{9,1023}$ a_9
Output layer weight and activation threshold velocities (Training only):
$wv_{0,0}$ $wv_{0,1}$ $wv_{0,2}$ $wv_{0,3}$ $wv_{0,4}$ $wv_{0,5}$ $wv_{0,6}$ $wv_{0,7}$ $wv_{0,8}$ $wv_{0,9}$ av_0
Hidden layer weight and activation threshold velocities (Training only):
$wv_{0,0}$ $wv_{0,1}$ $wv_{0,2}$ $wv_{0,1023}$ av_0 $wv_{1,0}$ $wv_{9,1022}$ $wv_{9,1023}$ av_9
Key:
$w_{n,m}$: The weight on synapse m of node n of the layer.
a_n : The activation threshold of node n of the layer.
$wv_{n,m}$: The velocity of the weight on synapse m of node n of the layer.
av_n : The velocity of the activation threshold of node n of the layer.

Figure 3.15: Illustration of the arrays used to implement the neural net. Shown here is the layout for a net with a 4096 parameter feature vector input, a 10 node hidden layer and a 1 node output layer. This is in fact the configuration that it was found worked best for this application.

Chapter 4

Evaluation

This chapter contains an evaluation of the performance of the system and each of its sub modules. Details of the testing of every low level object in the system have not been included.

4.1 Segmenter

A small section of test code was written to output segments generated by the segmenter to an image file for analysis. Figure 4.1 contains a test image and the segments produced for it. The white areas are the segments output. The black line a few pixels in from the boundary of each segment is the convex hull that is computed as part of the segmentation process[1]. From this and other such test images it is clear that the segmenter is performing exactly to specifications. Specifically:

- Every whole car in the image has a segment which exactly covers it.

- The number of other segments defined on the image is low. This image contains only 104 segments. This means only 104 invariance transforms and classifications have to be performed, which it turns out is about the ball park needed to get the performance of the system to meet its specifications.

[1]Remembering that the segments are grown by a few pixels in all directions after they have been generated from the convex hull. Hence the convex hulls lie several pixels in from the boundaries of each segment.

4.2 Fourier invariance transforms

Tests performed

Initially the transforms were tested with data that checked that they met their mathematical specifications. So for example Fourier basis functions were fed into the FFTs to check that they output a single point in the Fourier plane etc. From these tests it was determined that the transforms were functioning to their mathematical specification.

Tests were then performed to find out how well the transforms actually achieved invariance. Details and results of this testing are given in Appendix E. From these it was determined that while translation invariance worked perfectly, rotation and scaling invariance hardly worked at all.

Discretisation errors

It was decided that these problems were due to small discretisation errors that were being amplified by the log polar transforms. To explain this Figure 4.2 shows the first two stages of the processing of two images that are rotations of each other. Note the way that the Fourier plane is simply rotated by the same amount as the images. Now consider the discretisation errors that must necessarily occur in the eight elements that surround the element at the origin of this Fourier plane. For example if the image and hence the Fourier plane are rotated by 40 degrees, then there is no possibility that these eight elements can correctly represent the rotation and so the errors are going to be large. Now note the way that the log polar transform maps these elements onto large areas of the log polar plane.

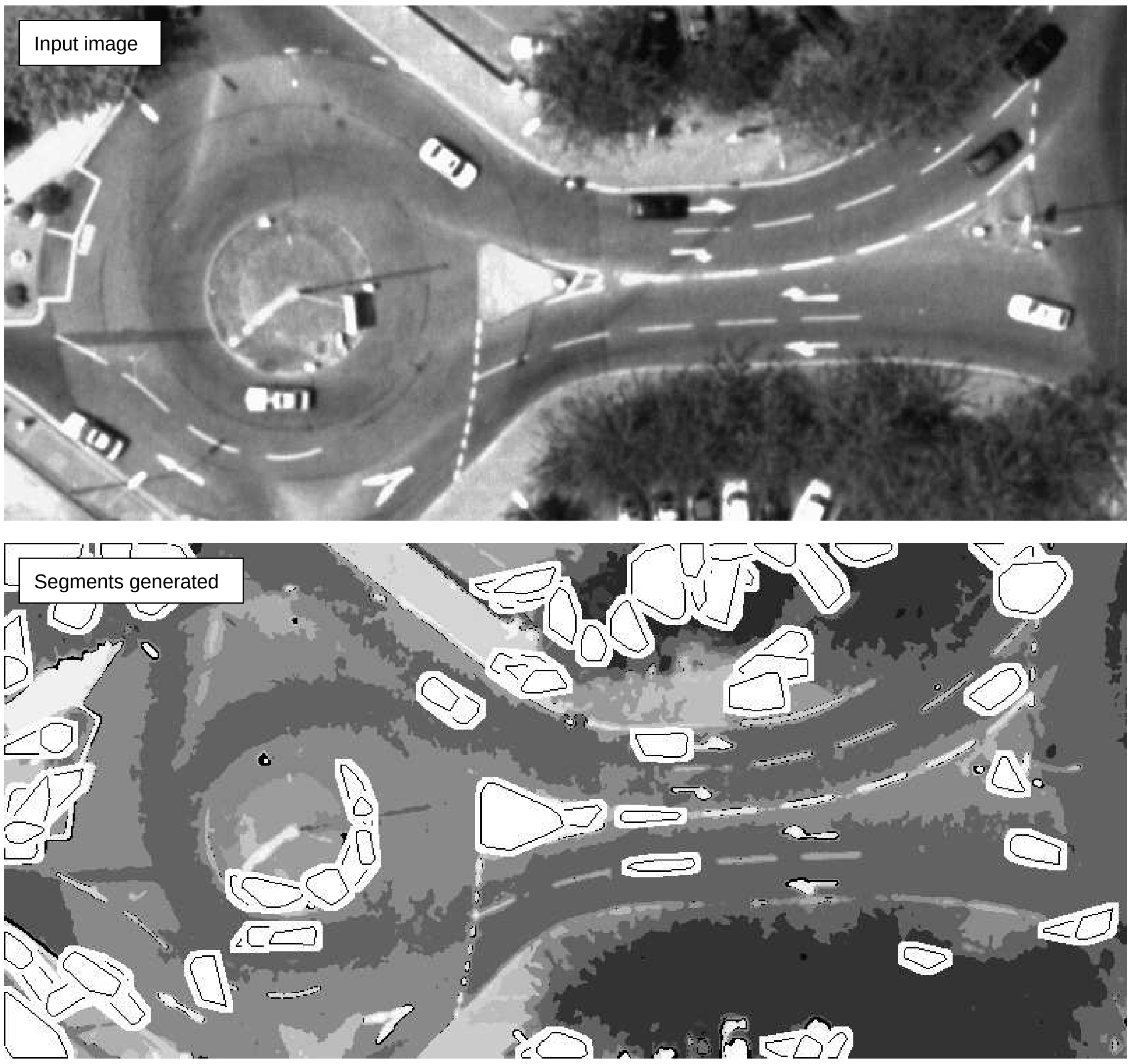


Figure 4.1: *Example of the segmenter's output*

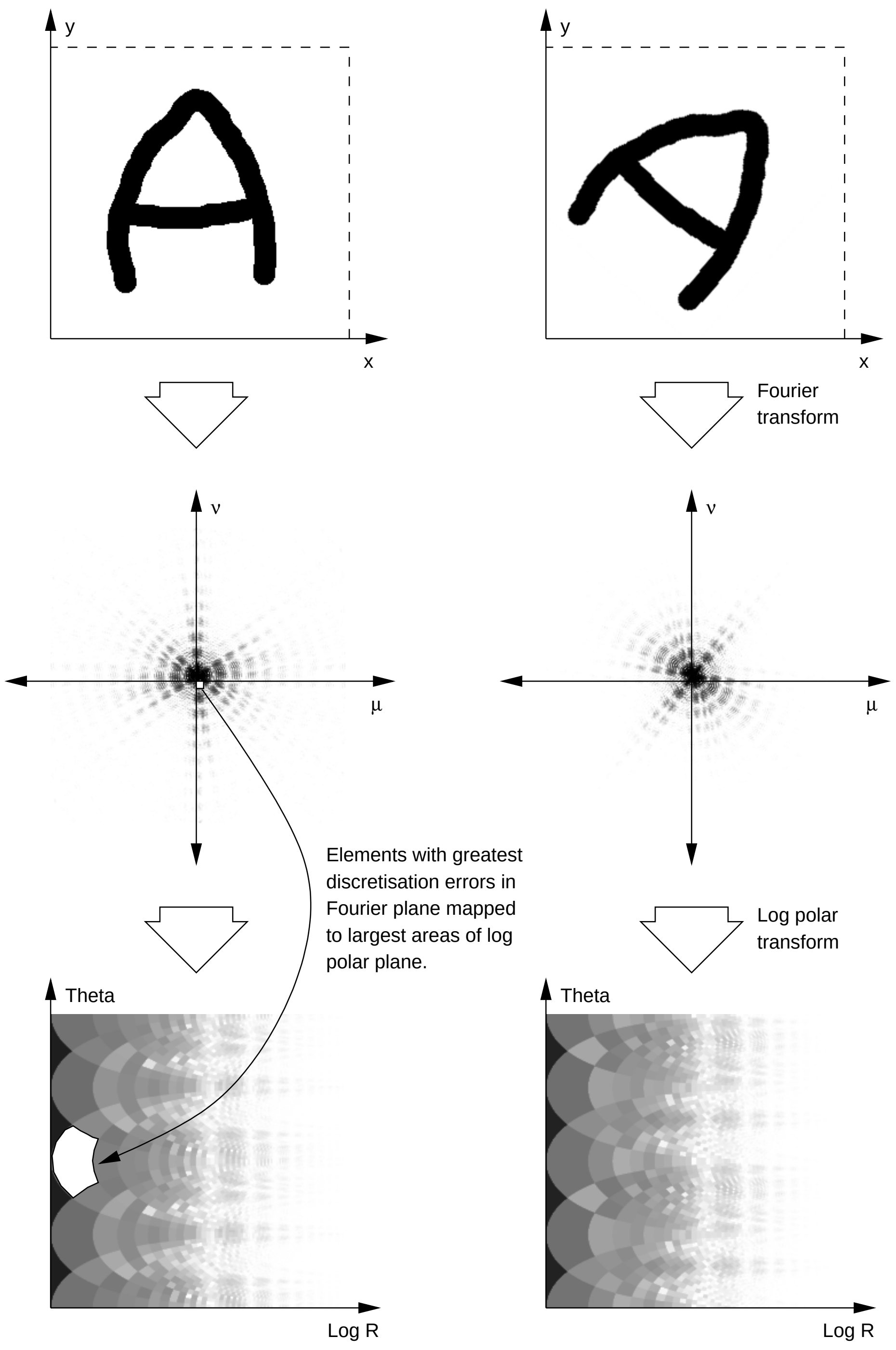

y
x
y
x
Fourier
transform
ν
μ
ν
μ
Elements with greatest discretisation errors in Fourier plane mapped to largest areas of log polar plane.
Log polar
transform
Theta
Log R
Theta
Log R

Figure 4.2: *Illustration of discretisation errors*

Hence the elements of the Fourier plane that have the largest discretisation error are mapped onto the largest areas of the log polar plane.

Reference back to the original papers [2, 3, 4, 7, 12], revealed that those that had performed the transforms optically using lenses and hence at an effectively infinite resolution had reported good results. Meanwhile those that had implemented it in the discrete case had been remarkably quiet about how well the technique had actually worked. This led to the conclusion that perhaps the discrete case had a few more implementation problems than the papers extolling its virtues were willing to let on.

The polar mapping

The simple solution implemented was to convert the log polar mapping into a simple polar mapping. This served to remove the massive dilation of the central elements along the r axis of the polar plane caused by the log function. Of course the polar mapping still dilates these elements along the *theta* axis but the overall amplification is greatly reduced.

With this new implementation the rotation invariance tests were repeated and the results are included in Appendix F. These results suggested that rotation invariance was showing signs of functioning. However presumably due the the discretisation errors that are intrinsic to the system its performance was still poor.

To overcome the fact that removing the log function also removes the ability for the transforms to achieve scaling invariance, the segments being fed into the transformations were all prescaled so that they had the same dimensions. This is an exceptionally crude way of achieving scaling invariance, but as will be see in Section 4.4, even though the rotation invariance has poor performance and the scaling invariance is crude, the complete system actually manages to achieve surprisingly good results.

4.3 Neural net classifier

The neural net was tested on the well defined problem of recognising hand written characters such as those shown in Figure 4.1. It obtained almost perfect performance on the classification of unseen examples[2], however it did understandably have a small amount of confusion about letters such as capital Q and O. This test therefore proved that the net is capable of solving complex classification problems.

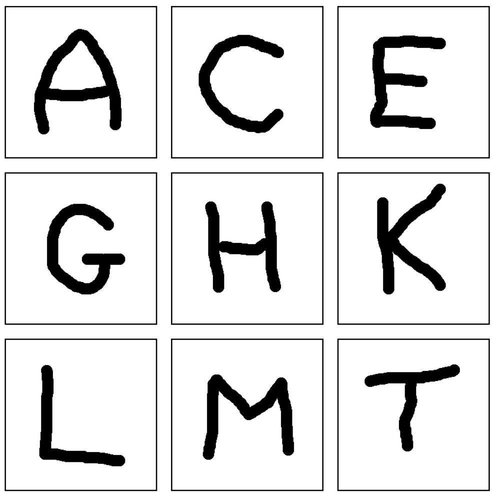


Figure 4.3: *An example of the hand written characters used to test the neural net.*

4.4 The complete system

Average complexity images

Even though there appear to be some problems with the Fourier invariance transforms, the system as a whole produces remarkably good results for the majority of the images given to it for processing. For example, Figure 4.4 contains an image in which the system has located[3] all but two of the un-occluded cars and has not misclassified a single piece of background scenery as a car. This is the typical level of performance obtained from the system when it is used to process images of this level of complexity.

[2]Note that it was trained directly on the characters with no invariance transforms involved anywhere. Hence would only classify character in one orientation. However this it enough to show that the net its self is functioning correctly.

[3]Note that the test harness that was written to call the `CarScanOracle` object has rendered the list of segments containing cars returned as rectangles centered on the midpoints of these segments.

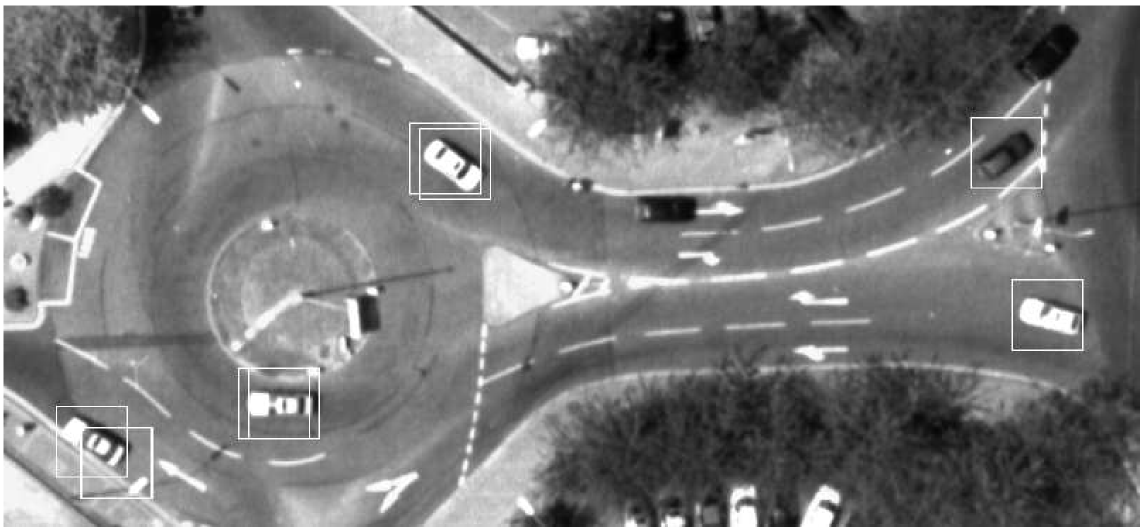


Figure 4.4: *Example of the output from the system for an input image of average complexity.*

High complexity images

In this application a high complexity image is an image which contains a large quantity of background features such as buildings and other objects that bare a close resemblance to cars. Figure 4.5 contains an example of such an image.

When the system was run on this image the segmenter output 627 segments. The performance of the classifier at classifying these segments is detailed in Figure 4.6[4].

These results show that the classifier correctly classified 92% of the segments that contained cars and 96% of the segments that contained scenery. This represents amazingly high accuracy for a complex real world problem such as this.

There is however a problem. Due to the complexity of the background data in this image, the segmenter output many times more segments that contained scenery than it did segments that contained cars. The results above show that 4% of these scenery segments were miss classified as cars. In real terms this means that almost as many scenery segments were misclassified as cars as there were cars in the image. Figure 4.7 contains the output from a section of the image which contains some of these misclassifications.

Hence even though the classifier has a very high accuracy, the shear complexity of certain input images, which cause the segmenter to output a lot of scenery segments, can make the classifier's inaccuracies significant. However this really is a worst case situation. For the majority of images the complexity is low enough that the inaccuracies don't become significant.

The "Yellow box" problem

During testing it was found that the system does have one specific problem. This is that it gets completely confused by "Yellow box" junctions and the other forms of cross hatch marking that are often found at road intersections. This problem is illustrated in Figure 4.8. Note that it is not confused by the dashed lines normally found in the middle of a road, only markings that form particular shape boxes in the image.

Attempts to solve this problem by adding extra examples of these road markings to the training set, resulted in an overall reduction of the number of cars detected. Hence these road marking evidently contain some of the features that the net is using to identify cars, which causes the net to have difficulty telling them apart.

[4]Note that these figures have been corrected by subtracting special cases such as the line of cars outside the car show room and the yellow box junction, so that they are not unnaturally skewed.

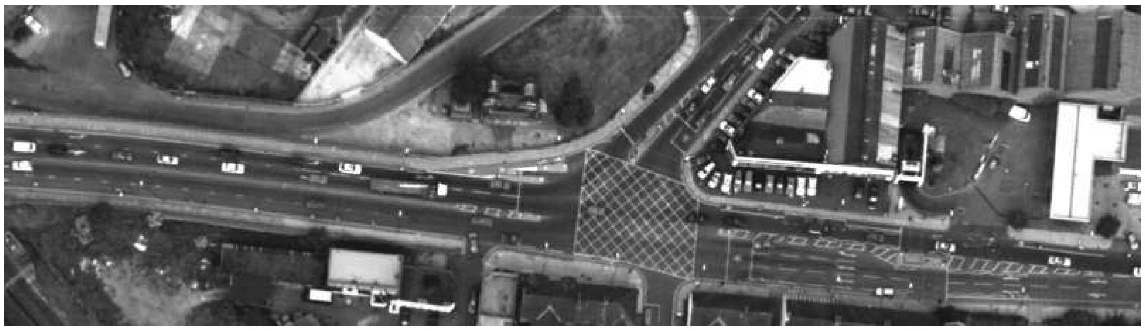


Figure 4.5: *Example of a high complexity image.*

Segments containing scenery that were correctly classified as such (Correct rejections):	582
Segments containing cars that were correctly classified as such (Correct accepts) :	24
Segments containing cars that were classified as containing scenery (False rejections):	2
Segments containing scenery that were classified as containing cars (False accepts) :	19

Figure 4.6: *The performance of the classifier when classifying the segments produced for Figure 4.5.*

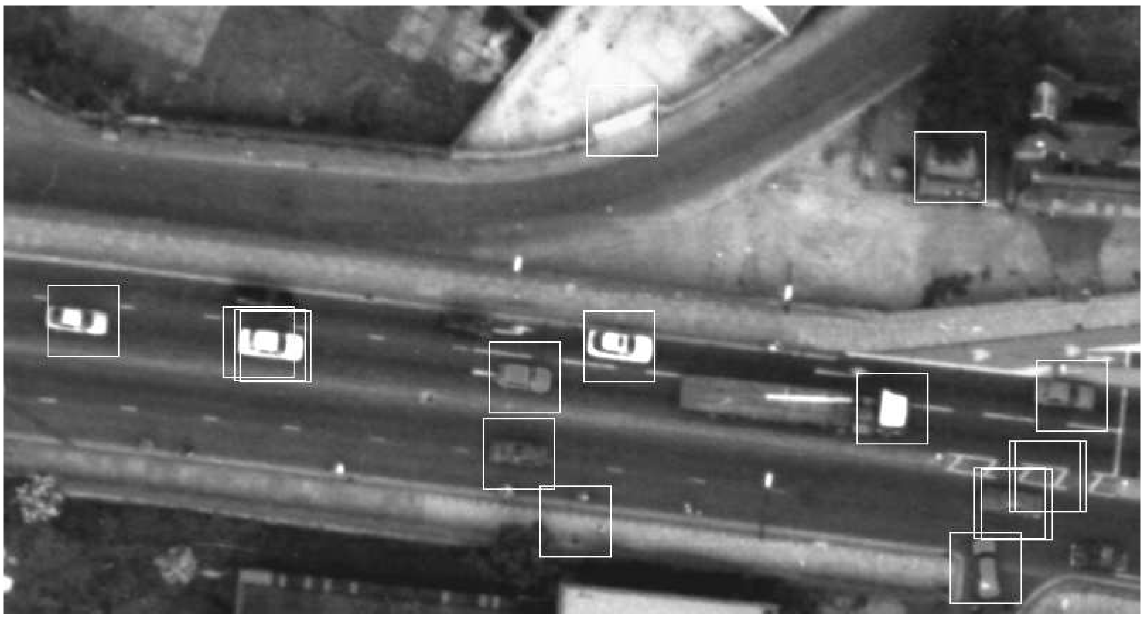


Figure 4.7: *Example of misclassifications in a complex scene.*

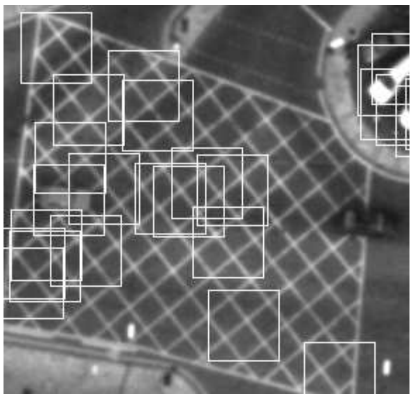


Figure 4.8: *An example of how the system is confused by "Yellow box" junctions.*

Machine	Figure 4.4	Figure 4.5
PentiumII-350	37s	186s
Pentium-166	88s	395s

Figure 4.9: *Rough performance results for the system, giving the time taken to process some of the figures in this chapter.*

Speed

Some rough performance figures for the system are given in Figure 4.9. These show that the system is very close to its target of being able to perform the task at least as fast as a human. For example a human probably wouldn't be able to obtain an average of 186 seconds for images such as Figure 4.5 if you include the fact that a human is only likely to work a maximum of 12 hours in a 24 hour day.

Chapter 5

Conclusions

5.1 Comparison with aims

The basic aim of this project was to implement a back end library that was capable of locating cars in vertical aerial photography. As detailed in the Implementation and Evaluation sections, a back end library has been implemented and for the majority of images it accurately locates the cars contained within them.

5.2 Design improvements

Invariance transforms

It is clear from the Evaluation that there are some serious problems with the Fourier log polar invariance technique. It is therefore probable that High Order Neural Nets (HONN) or hybrid Fourier/HONN techniques may be utilised to obtain a higher performance.

Handling complex images

There is a possibility that improvements to the invariance transforms could improve the accuracy of the classifier. Hence alleviating the problem of the classifier inaccuracies becoming significant in complex images. However without such improvements, experience with the training of the classifier suggests that it is unlikely that its accuracy can be improved past the 96% mark. This means that the solution would be to improve the segmenter by adding further heuristics to cut down the number of segments containing scenery output. It is likely that this would be an achievable objective.

Solving the “Yellow box” problem

As the road markings that confuse the classifier are always yellow or white lines on a dark background. It should therefore be possible to add special case heuristics to the segmenter, that detect and discard all segments containing such features before they are passed to the classifier.

Bibliography

[1] Donald Lewine. *POSIX programmer's guide.* O'Reilly, 1994.

[2] Invariant pattern recognition: A review. *Pattern Recognition*, 29:1–17, 1996.

[3] D Casasent and D Psaltis. Position, rotation and scale invariant optical correlation. *Applied Optics*, 15:1793–1799, 1976.

[4] D Asselin and H H Arsenault. Rotation and scale invariance with polar and log polar coordinate transformations. *Optics Comm.*, 104:391–404, jan 1994.

[5] J G Daugman. Computer vision. Lecture notes from Part II of the Cambridge University Computer Science Tripos, 1999.

[6] Rotation and scale invarient pattern recognition using a multistaged neural network. *SPIE*, 1606:241.

[7] H Y Kwan, B C Kim, D S Cho, and H Y Hwang. Scale and rotation invariant pattern recognition using complex-log mapping and augmented second order neural network. *Electronics Letters*, 29(7):620–621, 1993.

[8] Kevin Gurney. *An introduction to Neural Networks.* UCL press, 1997.

[9] Christopher M Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[10] Portable pixmap file format. WWW: http://www.dcs.ed.ac.uk/~mxr/gfx/2d/PPM.txt.

[11] Peter Heinrich. Algorithm alley. *Dr. Dobb's Journal*, April 1996.

[12] R A Messner and H H Szu. An image processing architecture for real time generation of scale and rotation invariant patterns. *CVGIP*, 31:50–66, 1985.

[13] Application of wavelet and neural procesing to automatic target recognition. *SPIE*, 3069, ?

[14] Giorgio Bonmassar and Eric L. Schwartz. Space-variant fourier analysis: The exponential chirp transform. *IEEE transactions on pattern analysis and machine intelligence*, 19(10):1080–1089, oct 1997.

[15] Mary L Boas. *Mathematical methods in the physical sciences.* Wiley, 1983.

[16] Brian W Kernighan and Dennis M Ritchie. *The C programming language.* Prentice Hall, 1988.

Appendix A

Header file for InplaceVector object

```
/*******************************************************************/
/* Source :                                                        */
/* InplaceVector.h                                                 */
/*                                                                 */
/* Description :                                                   */
/* A self maintaining inplace vector that increses and decreases   */
/* its size as required.                                           */
/*                                                                 */
/* Language :                                                      */
/* C                                                               */
/*                                                                 */
/* Authors & Modifications :                                       */
/* 1999, Richard Lancaster, original.                              */
/*                                                                 */
/* Copyright (c) 1999 Richard Lancaster.                           */
/* See LICENSE file for full license details.                      */
/*******************************************************************/

#ifndef __InplaceVector_h
#define __InplaceVector_h

#include "StdDef.h"



/* STRUCTURE AND DATATYPE DEFINITIONS */

/*
 * InplaceVector descriptor
 *
 * THIS STRUCTURE CONTAINS PRIVATE DATA AND SHOULD NOT BE ACCESSED
 * DIRECTLY.  ACCESS SHOULD ONLY BE THROUGH THE METHODS PROVIDED .
 */

struct InplaceVectorStruct
  {
  /* The size of each element held in the vector in bytes */
  int element_size;

  /* The minimum number of spaces that are to be allocated for */
  /* elements in the vector and the number of elements by which */
  /* the size should be incremented when the vector becomes full */
  int default_size;
  int inc_step;

  /* The number of elements currently held in the vector and the */
  /* number of elements the vector currently has the capacity */
  /* to hold without memory reallocation */
  int number_of_elements;
  int capacity;

  /* The actual data */
  /* If capacity is zero then this pointer is NULL */
  byte *data;
  };

typedef struct InplaceVectorStruct InplaceVector;



/* CONSTRUCTOR AND DESTRUCTOR ROUTINES */

/*
 * This routine creates an inplace vector.
 *
 * NOTES:
 *
 * - The vector produced uses zero based addressing.
 *
 * - The vector automatically reallocates the amount of memory it
 *   is using depending on the number of elements stored in it.
 *
 * PARAMETERS
 *
 * element_size:
 *     The size of the elements to be stored in the vector
 *     in bytes.
 *
 * default_size:
 *     The initial number of elements to be allocated in the
 *     vector.
 *
 * inc_step:
 *     The step size by which to increase and decrease the
 *     size of the vector when required.
 *
 * RETURNS
 *
 * The vector or NULL on failure.
 */

extern InplaceVector *InplaceVectorCreate
    (int element_size, int default_size, int inc_step);

/*
 * This routine destroys a vector.
 * If the vector is NULL then it is ignored.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to be destroyed.
 */

extern void InplaceVectorDestroy(InplaceVector *vector);



/* PROPERTY MANIPULATION ROUTINES */

/*
 * Returns the number of elements in the vector.
 * Note that this is not the current capacity of the vector the
 * data about which is private.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to query size of.
 *
 * RETURNS
 *
 * The size of the vector
 */

extern int InplaceVectorGetSize(InplaceVector *vector);



/* VECTOR STYLE DATA MANIPULATION ROUTINES */

/*
 * Places an new element into a vector at a specified index.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to place element into.
 *
 * index:
 *     The index to place element at.
 *
 * value:
 *     A pointer to the value to be placed into the index.
 *
 * RETURNS
 *
 * True if sucessful or false otherwise.
 */

extern boolean InplaceVectorPut
    (InplaceVector *vector,
    int index, void *value);

/*
 * Gets an element from a vector.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to get an element from.
 *
 * index:
 *     The index to get the element from.
 *
 * value:
 *     A pointer to the location to copy the value into.
 *
 * RETURNS
 *
 * True if sucessful or false otherwise.
 */

extern boolean InplaceVectorGet
    (InplaceVector *vector,
    int index, void *value);

/*
 * Appends an element to the end of the data currently
 * held by the vector.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to append an element to.
 *
 * value:
 *     A pointer to the value to append.
 *
 * RETURNS
 *
 * true if sucessful or false otherwise.
 */

extern boolean InplaceVectorAppend
    (InplaceVector *vector,
    void *value);

/*
 * Removes an element from a vector, elements above will be shuffled
 * down.  The memory held by the vector will also be compressed
 * if that is a sensible thing to do.
 *
 * NOTES:
 *
 * - If memory compression fails then no error is returned,
 *   the memory just remains uncompressed.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to remove an element from.
 *
 * index:
 *     The index to remove.
 *
 * RETURNS
 *
 * True on sucess or false otherwise.
 */

extern boolean InplaceVectorCut
    (InplaceVector *vector,
    int index);

/*
 * Flushes all the elements from a vector.
 *
 * This function is designed to be far more efficient than
 * calling "Cut" on every single element.
 *
 * NOTES:
 *
 * - If memory compression fails then no error is returned,
 *   the memory just remains uncompressed.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to flush.
 *
 * RETURNS
 *
 * True on sucess or false otherwise.
 */

extern boolean InplaceVectorFlush
    (InplaceVector *vector);



/* STACK STYLE DATA MANIPULATION ROUTINES */

/*
 * Pushes a value onto the end of a vector.
 * Eg. This is a stack Push operation.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to push element onto.
 *
 * value:
 *     A pointer to the value to push onto the vector.
 *
 * RETURNS
 *
 * true if sucessful or false otherwise.
 */

extern boolean InplaceVectorPush
    (InplaceVector *vector,
    void *value);

/*
 * Reads the last element from a vector and then removes it
 * from the vector.  Eg. This is a stack pop operation.
 *
 * Popping from an empty stack results in true being returned,
 * no value is written out and the stack_empty boolean is set.
 *
 * PARAMETERS
 *
 * vector:
 *     The vector to pop element from.
 *
 * value:
 *     A pointer to the location to place the poped element.
 *     If the stack is empty then this value is unaltered.
 *
 * stack_empty:
 *     A pointer to a boolean to be set to true if an attempt
 *     is made to pop from an empty stack.  Otherwise it is set to
 *     false.
 *
 *     A NULL pointer can be passed in if you don't want to know.
 *     However that would be considered bad programming style.
 *
 * RETURNS
 *
 * True if sucessful or false on an error.
 * Poping from an empty stack is defined as sucessful.
 */

extern boolean InplaceVectorPop
    (InplaceVector *vector,
    void *value, boolean *stack_empty);

#endif
```

Appendix B

Definition of ImageSegment structure

```
/*
 * ImageSegment descriptor
 *
 * This structure provides public data and so may be accessed
 * directly.
 */

struct ImageSegmentStruct
  {
  /* -- FLAGS -- */

  /* Segment status flags */
  boolean discarded;
  boolean checked;

  boolean classified;
  int class;
  int parent_object;

  /* -- DATA -- */

  /* List of the pixels that make up the segment */
  /* Each pixel is stored in an XYCoordinate structure */
  InplaceVector *pixels;

  /* -- STATISTICS -- */

  /* Average colour of the segment */
  byte avg_red;
  byte avg_green;
  byte avg_blue;

  /* Segment bounds */
  int min_x;
  int max_x;
  int min_y;
  int max_y;

  /* Segment dimensions */
  int width;
  int height;

  /* Segment center point */
  int mid_x;
  int mid_y;

  /* Density details */

  /* NOTE: The idensity is the bounding volume of the segment */
  /* (max_x - min_x + 1) * (max_y - min_y + 1), divided by the */
  /* number of pixels in the segment.  Eg. it is the inverse */
  /* density */
  int idensity;
  };

typedef struct ImageSegmentStruct ImageSegment;
```

Appendix C

Header file for GFX module

```
/*******************************************************************/
/* Source :                                                        */
/* GFX.h                                                           */
/*                                                                 */
/* Description :                                                   */
/* Graphics effects routines.                                      */
/*                                                                 */
/* Language :                                                      */
/* C                                                               */
/*                                                                 */
/* Authors & Modifications :                                       */
/* 1999 Richard Lancaster, original.                               */
/*                                                                 */
/* Copyright (c) 1999 Richard Lancaster.                           */
/* See LICENSE file for full license details.                      */
/*******************************************************************/

#ifndef __GFX_h
#define __GFX_h

#include "BitMatrix.h"
#include "RawPixmap.h"
#include "StdDef.h"
#include "String.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>



/* Constant definitions */

#define GFX_EDGE_EFFECT_BLACK  (0)
#define GFX_EDGE_EFFECT_WHITE  (1)
#define GFX_EDGE_EFFECT_WRAP   (2)
#define GFX_EDGE_EFFECT_EXTEND (3)



/* CONVOLUTION KERNELS */

/*
 * A selection of standard 3x3 convolution kernels
 */

extern int GFX_kernel_gaussian[][];
extern int GFX_kernel_laplacian[][];
extern int GFX_kernel_sobelh[][];
extern int GFX_kernel_sobelv[][];



/* PROCESSING ROUTINES */

/*
 * Apply a convolution kernel to a pixmap.
 *
 * NOTES:
 *
 * - The result obtained for each channel of each destination pixel
 * after applying the kernel is truncated to the range -255 to 255
 * inclusive.
 *
 * - The result stored in each output element of the dest_pixmap
 * is abs(x).
 *
 * - The sign of each output element is discarded unless a sign
 * BitMatrix is passed to the function.
 *
 * PARAMETERS
 *
 * source_pixmap:
 *     The pixmap to apply the kernel to.
 *
 * source_sign_plane:
 *     A BitMatrix of size:
 *
 *         (source_pixmap->width * 3) by (source_pixmap->height)
 *
 *     That holds the sign of each element in source_pixmap.
 *     A 0 represents positive or zero, 1 represents negative.
 *
 *     If a NULL is passed as this parameter then all values in
 *     the source pixmap are considered to be positive.
 *
 * dest_pixmap:
 *     The pixmap to place the result of the convolution into.
 *     This must be the same size as the source pixmap.
 *     This will work if it is the same pixmap as the source_pixmap
 *     but it is likely to be slightly slower (probably not by
 *     very much though).
 *
 * dest_sign_plane:
 *     A BitMatrix of size:
 *
 *        (source_pixmap->width * 3) by (source_pixmap->height)
 *
 *     to hold the sign of each element in dest_pixmap.
 *     A 0 represents positive or zero, 1 represents negative.
 *
 *     If a NULL is passed as this parameter then the sign
 *     is discarded.
 *
 * kernel:
 *     Pointer to a 2D C array of integers that contains the
 *     convolution kernel.
 *
 * divisor:
 *     The divisor to apply to each result pixel.
 *
 * kernel_width:
 * kernel_height:
 *     The width and height of the kernel array.
 *     Both must be greater than 0 and odd.
 *
 * edge_effect:
 *     Specification of how to handle edges.
 *
 * RETURNS
 *
 * true if sucessful or false otherwise.
 */

extern boolean GFXConvolve
    (RawPixmap *source_pixmap, BitMatrix *source_sign_plane,
    RawPixmap *dest_pixmap, BitMatrix *dest_sign_plane,
    int *kernel, int divisor,
    int kernel_width, int kernel_height,
    int edge_effect);

/*
 * Apply a set of convolution kernels to a pixmap in parallel.
 *
 * If used with n kernels then the following effect is
 * obtained:
 *
 *   - n copies of the image are made.
 *   - One of the n kernels is applied to each image.
 *   - The n images are merged back together according to a
 *     merging function.
 *
 * In reality no copies of the image are made.
 *
 * The reason for this is to allow something along the lines of
 * a horizontal edge detect and a vertical edge detect to be
 * performed.  Then have their results merged together without
 * creating needless copies of the image.
 *
 * PARAMETERS:
 *
 * (The parameters are as for GFXConvolve with the following
 * exceptions)
 *
 * kernels:
 *     Points to an array of kernels to apply.  All kernels must
 *     be of the size specified in kernel_width and kernel_height.
 *     Each kernel is of the format accepted by GFXConvolve.
 *
 * divisors:
 *     Points to an array of divisors to apply to the result
 *     produced by each kernel.  divisors[n] is applied to
 *     the result from kernels[n].
 *
 * num_kernels:
 *     The number of elements in the kernels and divisors arrays.
 *
 * merge_style:
 *     How to merge the results of each kernel to produce the output
 *     pixel.
 *
 * RETURNS
 *
 * true if sucessful or false otherwise.
 */

#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_ADD_ABS    (0)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_ADD_SIGNED (1)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_MAX_ABS    (2)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_MAX_SIGNED (3)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_MIN_ABS    (4)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_MIN_SIGNED (5)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_AVG_ABS    (6)
#define GFX_PARALLEL_CONVOLVE_MERGE_STYLE_AVG_SIGNED (7)

extern boolean GFXParallelConvolve
    (RawPixmap *source_pixmap, BitMatrix *source_sign_plane,
    RawPixmap *dest_pixmap, BitMatrix *dest_sign_plane,
    int **kernels, int *divisors, int num_kernels,
    int kernel_width, int kernel_height,
    int merge_style, int edge_effect);

/*
 * Apply a threshold to a pixmap.
 *
 * This takes in a pixmap and for each pixel:
 *
 * - Reads the peak intensity.
 *
 * - If peak intensity is greater than or equal to the value of
 *   'threshold' then 255 is output into all 3 channels of the
 *   corresponding pixel of the destination pixmap.
 *
 * - If peak intensity is less than the value of
 *   'threshold' then 0 is output into all 3 channels of the
 *   corresponding pixel of the destination pixmap.
 *
 * PARAMETERS
 *
 * source_pixmap:
 *     The pixmap to apply the threshold to.
 *
 * dest_pixmap:
 *     The pixmap to output the bitmap image to.
 *     This must be the same size as the source pixmap.
 *     This will work if it is the same pixmap as the source_pixmap.
 *
 * threshold:
 *     The threshold to apply.  This does _not_ have to be in the
 *     range 0 to 255.
 *
 * RETURNS
 *
 * true if sucessful or false otherwise.
 */

extern boolean GFXApplyThreshold
    (RawPixmap *source_pixmap,
    RawPixmap *dest_pixmap,
    int threshold);

#endif
```

Appendix D

Neural net feature vector evaluation function

Neural net structure definition

```
/*
 * The ThreeLayerNN structure.
 *
 * This structure contains private data and should not be
 * accessed directly.
 *
 * The nodes in one layer are assumed to be fully connected to the
 * nodes in the layer above.
 *
 * This structure is designed with simple flat arrays so that
 * it is _fast_.
 */

struct ThreeLayerNNStruct
  {
  /* The number of nodes in each layer.  The input layer are just */
  /* distribution rather than processing nodes */
  int output_nodes;
  int hidden_nodes;
  int input_nodes;

  /* The weights of the node inputs */
  /* These are arrays of doubles.  The first n elements are the */
  /* weights of the first node in that layer.  The next n are the */
  /* weights of the next node etc.  There lengths are therefore */
  /* the number of nodes in the layer multiplied by the number of */
  /* nodes in the layer below plus 1.  The plus 1 is for the */
  /* threshold pseudo weight, which adds a weight to each node in */
  /* the layer */
  double *output_node_weights;
  double *hidden_node_weights;
  };

typedef struct ThreeLayerNNStruct ThreeLayerNN;
```

Evaluation function declaration

```
/*
 * Evaluates a feature vector presented to it.  It only returns
 * the raw output of the net.  Deciding how to classify the
 * results is left to the calling procedure.
 *
 * PARAMETERS
 *
 * net:
 *     The network to perform evaluation with.
 *
 * input_vector:
 *     The input vector presented as an array.
 *     This must be of length net->input_nodes.
 *
 * hidden_vector:
 *     A vector to hold the outputs from the hidden nodes.
 *     This is passed into the routine rather than being allocated
 *     internally so that it doesn't need to be continuously
 *     reallocated and so that these outputs can be interogated.
 *     This must be of length net->hidden_nodes.
 *
 * output_vector:
 *     The vector to place the processing results into.
 *     This must be of length net->output_nodes.
 *
 * hidden_raw:
 *     If not NULL this vector is filled with the raw version of
 *     hidden_vector that has not been passed through the sigmoid.
 *     This must be of length net->hidden_nodes or NULL.
 *
 * output_raw:
 *     If not NULL this vector is filled with the raw version of
 *     output_vector that has not been passed through the sigmoid.
 *     This must be of length net->output_nodes or NULL.
 */

extern void ThreeLayerNNEvaluateVector
    (ThreeLayerNN *net,
    double *input_vector,
    double *hidden_vector,
    double *output_vector,
    double *hidden_raw,
    double *output_raw);
```

Evaluation function definition

```
/*
 * See header file for description of routine
 */

extern void ThreeLayerNNEvaluateVector
    (ThreeLayerNN *net,
    double *input_vector,
    double *hidden_vector,
    double *output_vector,
    double *hidden_raw,
    double *output_raw)
  {
  int k;
  int j;
  int i;

  double *base_weight;

  double a;
  double y;

  /* Assertions */

    assert(net != (ThreeLayerNN *) NULL);

  /* Processing */

    /* Evaluate hidden nodes */
    for(k = 0; k < net->hidden_nodes; k++)
      {
      /* Set activation to zero */
      a = 0.0;

      /* Find the base of the weight list for this node */
      base_weight = net->hidden_node_weights
          + (k * (net->input_nodes + 1));

      /* Compute the scalar product of the inputs and the weights */
      for(i = 0; i < net->input_nodes; i++)
        {
        a += (*(input_vector + i)) * (*(base_weight + i));
        }

      /* Add in the threshold pseudo weight */
      a += (-1.0) * (*(base_weight + net->input_nodes));
    /* If requested store the raw activation energy */
    if(hidden_raw != (double *) NULL)
      {
      *(hidden_raw + k) = a;
      }

    /* Pass 'a' through sigmoid */
    y = ThreeLayerNNSigma(a);

    /* Store the output */
    *(hidden_vector + k) = y;
    }

  /* Evaluate output nodes */
  for(j = 0; j < net->output_nodes; j++)
    {
    /* Set activation to zero */
    a = 0.0;

    /* Find the base of the weight list for this node */
    base_weight = net->output_node_weights
        + (j * (net->hidden_nodes + 1));

    /* Compute the scalar product of the inputs and the weights */
    for(i = 0; i < net->hidden_nodes; i++)
      {
      a += (*(hidden_vector + i)) * (*(base_weight + i));
      }

    /* Add in the threshold pseudo weight */
    a += (-1.0) * (*(base_weight + net->hidden_nodes));

    /* If requested store the raw activation energy */
    if(output_raw != (double *) NULL)
      {
      *(output_raw + j) = a;
      }

    /* Pass 'a' through sigmoid */
    y = ThreeLayerNNSigma(a);

    /* Store the output */
    *(output_vector + j) = y;
    }
}
```

Appendix E

Fourier log polar invariance transform testing

This appendix details the testing performed on the Fourier log polar invariance transforms to discover how well they actually achieved invariance.

Method

The set of hand written characters[1] shown in Figure E.1 was created[2]. In theory, if the invariant representation of a transformed example of one of these character is compared with the invariant representations of these characters. Then, if the invariance transforms are actually working, the difference between the transformed example's representation and the representation of the character it is a transform of, should be less than the difference between the transformed example's representation and the representations of any of the other characters. Rotated, scaled and translated versions of the characters were therefore created.

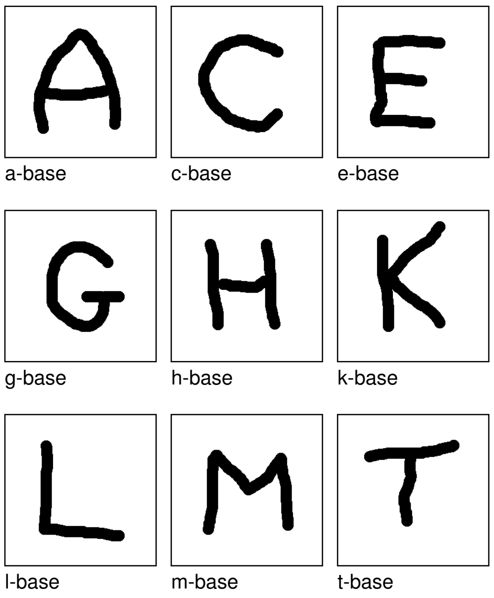


Figure E.1: *The base set of characters used for testing the invariance transforms.*

Results

The results obtained from performing the difference test are shown in Figure E.2.

Interpretation of results

These results showed two things:

- Translation invariance was working perfectly, as the results showed that there was never

[1]Hand written characters are well defined objects and hence provide a better test case than cars.

[2]These tests have been performed on larger datasets but this small sub set is illustrative of the kind of results obtained from the larger sets.

Variations	*Base characters*								
	a-base	c-base	e-base	g-base	h-base	k-base	l-base	m-base	t-base
a-rotated	$\underline{\mathbf{1.15}}$	2.21	2.86	1.93	2.23	2.21	2.48	2.98	2.48
c-rotated	1.70	$\underline{1.72}$	2.74	**1.51**	1.81	2.01	1.64	2.84	1.63
e-rotated	1.91	2.08	$\underline{2.43}$	**1.69**	2.15	1.82	1.89	2.58	2.27
g-rotated	1.58	1.53	2.49	$\underline{\mathbf{1.30}}$	1.65	1.74	1.82	2.63	1.79
h-rotated	2.04	2.08	2.56	**1.65**	$\underline{2.08}$	1.93	1.92	2.65	2.21
k-rotated	1.81	2.14	2.76	1.67	2.06	$\underline{2.01}$	**1.63**	2.90	1.88
l-rotated	2.37	1.68	2.19	1.57	2.08	**1.48**	$\underline{1.70}$	2.34	1.99
m-rotated	1.83	1.94	2.42	**1.62**	1.98	1.80	1.94	$\underline{2.54}$	2.14
t-rotated	2.42	2.67	3.98	2.22	2.68	3.13	**0.76**	4.10	$\underline{0.79}$
a-scaled	$\underline{2.62}$	2.91	4.33	2.50	2.88	3.47	1.01	4.45	**0.78**
c-scaled	2.89	$\underline{3.08}$	4.49	2.67	3.06	3.61	**1.38**	4.61	1.19
e-scaled	2.59	2.56	$\underline{3.77}$	2.24	2.59	2.90	1.46	3.90	**1.37**
g-scaled	3.02	3.28	4.69	$\underline{2.86}$	3.25	3.83	1.33	4.81	**1.22**
h-scaled	2.98	3.05	4.37	2.73	$\underline{2.96}$	3.54	1.52	4.47	**1.17**
k-scaled	2.69	2.89	4.22	2.50	2.85	$\underline{3.35}$	1.23	4.34	**1.07**
l-scaled	3.55	3.85	5.23	3.41	3.84	4.36	$\underline{\mathbf{1.69}}$	5.36	1.77
m-scaled	2.62	2.59	3.80	2.24	2.46	2.96	1.39	$\underline{3.91}$	**1.32**
t-scaled	3.86	4.19	5.61	3.77	4.14	4.74	2.10	5.73	$\underline{\mathbf{1.80}}$
a-translated	$\underline{\mathbf{0.00}}$	1.98	2.86	1.56	2.22	2.15	2.27	3.00	2.46
c-translated	1.98	$\underline{\mathbf{0.00}}$	1.59	1.10	1.65	1.23	2.45	1.64	2.65
e-translated	2.86	1.59	$\underline{\mathbf{0.00}}$	2.01	2.21	1.16	3.63	6.83	3.93
g-translated	1.56	1.10	2.01	$\underline{\mathbf{0.00}}$	1.56	1.26	1.96	2.09	2.29
h-translated	2.22	1.65	2.21	1.56	$\underline{\mathbf{0.00}}$	1.64	2.52	2.14	2.44
k-translated	2.15	1.23	1.16	1.26	1.64	$\underline{\mathbf{0.00}}$	2.76	1.26	3.09
l-translated	2.27	2.45	3.63	1.96	2.52	2.76	$\underline{\mathbf{0.00}}$	3.75	9.53
m-translated	3.00	1.64	0.84	2.09	2.14	1.26	3.75	$\underline{\mathbf{0.00}}$	4.00
t-translated	2.46	2.65	3.93	2.29	2.44	3.09	0.95	4.00	$\underline{\mathbf{0.00}}$

Figure E.2: This table shows the differences between the invariant representations of the base set of characters and the rotated, scaled and translated versions of this base set of characters. The differences are in arbitrary units, but the higher the number the greater the difference between the two representations. The minimum difference in each row is marked in bold type. The element that should be the minimum difference in each row if the invariance transforms are working is underlined.

any difference between a base character and its translated variation.

- Rotation and scaling invariance were not working at all, as there was hardly ever any correlation between each of the base characters and their rotated and scaled variations.

Appendix F

Fourier polar invariance transform testing

This appendix details the results of repeating the rotation invariance test from Appendix E on the polar (i.e. non log) invariance transform.

Results

The results obtained from performing this test are shown in Figure F.1.

Interpretation of results

These results showed that the use of the polar transform results in some correlation between the base characters and their variations. However it is still a fairly poor result.

Variations	*Base characters*								
	a-base	c-base	e-base	g-base	h-base	k-base	l-base	m-base	t-base
a-rotated	$\underline{\mathbf{0.53}}$	1.15	1.10	1.09	1.30	1.04	1.69	1.19	1.67
c-rotated	1.12	$\underline{\mathbf{0.52}}$	1.03	0.58	0.80	0.79	1.01	1.00	0.95
e-rotated	1.06	1.00	$\underline{0.83}$	0.90	0.90	**0.74**	1.48	0.83	1.55
g-rotated	0.97	0.49	0.90	$\underline{\mathbf{0.45}}$	0.78	0.66	0.98	0.90	1.00
h-rotated	1.32	0.85	0.96	0.75	$\underline{\mathbf{0.74}}$	0.82	1.38	0.92	1.46
k-rotated	0.97	0.72	0.82	**0.55**	0.80	$\underline{0.65}$	1.06	0.85	1.16
l-rotated	1.60	0.92	1.33	0.92	1.29	1.11	$\underline{\mathbf{0.47}}$	1.34	0.52
m-rotated	1.02	0.81	0.86	**0.75**	0.80	0.72	1.31	$\underline{0.80}$	1.35
t-rotated	1.60	1.03	1.64	1.03	1.48	1.37	0.38	1.63	$\underline{\mathbf{0.34}}$

Figure F.1: This table shows the differences between the invariant representations of the base set of characters and the rotations of this base set of characters. The differences are in arbitrary units, but the higher the number the greater the difference between the two representations. The minimum difference in each row is marked in bold type. The element that should be the minimum difference in each row if the invariance transforms are working is underlined.

Appendix G

Project proposal

G.1 Introduction

At all hours of the day a huge quantity of aerial surveillance photography is gathered by aircrafts and satellites. Extracting all of the available information content from this data by hand is impossible. Some form of automated processing of this data would therefore be beneficial.

The intention of this project is to develop a system to detect the locations of cars in aerial photographs. So for example, if given an aerial photo of a town it would return the coordinates of all the cars that are visible in the photo. Applications of such a system would include aiding in the investigation of traffic congestion, or if suitably adapted[1], tracking the movements of military vehicles.

G.2 Detailed specification

The core aim of this project is to implement a back end library which conceptually will have a single function call. The format of this function call will be along the lines of:

```
analyseImage(<image>)
returns <array of object locations>

  PARAMETERS:

  <image> : The image to be processed.

  RETURNS:

  An array of (x,y) coordinates describing
  where all the objects detected are to be
  found in the image.
```

[1]By retraining it to recognize military vehicles.

This low level implementation will allow the system to easily be integrated into any proposed applications. An example GUI and/or command line interface will also need to be implemented for testing purposes.

G.3 Metrics of success

When evaluating the success of the system there will be two levels of testing.

The first of these is unit testing. Here each of the components that make up the system will be fed artificial test data. If the results from the component match its design specification then the unit will be deemed to be functioning correctly.

The second level of testing is system testing. Here the whole system will be tested on its ability to perform it's specified function, e.g. to locate cars. This will be achieved by feeding the system carefully selected test images that will exercise its ability to detect cars in different circumstances and reject other objects that are not cars.

When evaluating the success at this level there are four metrics of importance:

Correct accept accuracy: The percentage of test images shown to the system that contain a car, in which the system correctly locates the car. This figure wants to be as high as possible.

Correct reject accuracy: The percentage of test images that contain no car, which the system rejects as containing no car. This figure wants to be as high as possible.

False accept accuracy: The percentage of test images shown to the system that don't contain a car, that the system reports as containing a car. This figure wants to be as low as possible.

False reject accuracy: The percentage of test images that the system is shown that contain a car, in which the system fails to locate the car. This figure wants to be as low as possible.

This is a computer vision problem and computer vision problems are notorious for not being able to achieve a 100% object recognition rate. For example if a car is partially obscured by a tree it is very unlikely that the system will be able to detect that car, whilst a human observer is likely to be able to infer that it's a car. Therefore the percentages that will be set as being "acceptable", will be selected after the initial reading weeks have determined what an acceptable success rate is for this type of application.

G.4 Project content

A basic design for the system will not be arrived at until this proposal has been submitted and several weeks of research and design have been completed. However it is likely that the system will involve some or all of the following components:

Basic image processing techniques, such as edge detection and image sharpening. These are likely to be used as a preprocessing operation to prepare the images for analysis.

Analytical image processing techniques, such as searching for rectangles in the image of approximately the correct size to be a car.

Domain conversion techniques, such as transformations into the wavelet or Fourier domains. This will be used to extract the main information components from the image or to obtain characteristics such as invariance of the data to rotation and scaling.

Feature classification systems, such as statistical decision, Bayesian inference or neural net classifiers. These will be used to take the features we've extracted from the image and decide whether the image is a car or not.

G.5 Implementation environment

The system will be implemented on the UNIX platform in the C++ language. This is to give a good combination of speed, expressive power and flexibility. Also implementation on a commonly available platform such as this will allow the continuation of the project even if a total hardware failure of the main development system is suffered.

G.6 Starting point

The language (C++) and the platform (UNIX) is already known by the author.

At the start of the project proposal period the author knew very little about computer vision and has not as of yet attended any substantial courses on the subject. Therefore before design begins the author will need to do substantial reading in the areas of computer vision, image analysis and target recognition.

G.7 Special resources required

- Aerial photographs for analysis have already been provided by the Cambridge University Committee for Aerial Photography.

- Development of system will be on author's personal machine.

G.8 Timetable

Weeks 1 and 2: *26th October 1998 - 8th November 1998*

Investigation into image processing and writing of small experimental routines for familiarization with techniques.

Weeks 3 and 4: *9th November 1998 - 22nd November 1998*

Further investigation, writing of experimental routines and high level design.

Deliverables:

- High level design
- Module interface specifications
- Basic design of modules

Weeks 5 to 10: *23rd November 1998 - 3rd January 1999*

Implementation of feature extraction modules.

Deliverables:

- Feature extraction modules

Weeks 11 to 13: *4th January 1999 - 24th January 1999*

Implementation of feature classification modules.

Deliverables:

- Feature classification modules.

Week 14: *25th January 1999 - 31st January 1999*

Preparing for and writing progress report.

Deliverables:

- Progress report

Weeks 15 and 16: *1st February 1999 - 14th February 1999*

Unit testing.

Deliverables:

- Unit testing results.

Week 17: *15th February 1999 - 21st February 1999*

System integration.

Deliverables:

- Integrated system.

Weeks 18 to 20: *22nd February 1999 - 14th March 1999*

Final testing and problem resolution.

Deliverables:

- Completed system
- Complete testing results

Weeks 21 to 23: *15th March 1999 - 4th April 1999*

Writing draft dissertation.

Deliverables:

- Draft dissertation

Weeks 24 and 25: *5th April 1999 - 18th April 1999*

Completing dissertation.

Deliverables:

- Completed dissertation