A Java Framework for Computer Vision

Stephen Sheridan

A Java Framework for Computer Vision

Stephen Sheridan
Institute of Technology Blanchardstown
Stephen.sheridan@itb.ie

Abstract

This paper outlines a framework implemented entirely in Java that attempts to give students exposure to computer vision systems from a practical standpoint. Various tools and technologies are introduced that will allow a student to acquire an input image through a WebCam, extract useful information from that input image and finally, attempt to make sense of the input.

1 Introduction

The field of computer vision is a diverse and interdisciplinary body of knowledge and techniques that attempt to understand the principles behind the processes that interpret signals from various sensors. Computer vision is often studied at an undergraduate level as part of the wider area of Artificial Intelligence or as a topic in its own right. Much of the work in teaching a subject like computer vision is theory based, as building complete vision systems takes an enormous amount of time and expertise. The problem that lies at the heart of computer vision is that of recovering information about the world from images. As with human vision, computer vision starts with the problem of image acquisition. Nature has provided humans with an excellent image acquisition device, the eye. Light enters the eye through a lens and is focused on a special layer of light sensitive cells at the back of the eye called the retina. From there, the light signals are converted to electrical impulses that travel down the optic nerve to an area of the brain called the striate cortex [1]. In comparison to what we know about the human eye, little is known about the function of the parts of the brain dedicated to vision. By using the human visual system as a model for computer vision we can break the problem down into three basic stages.
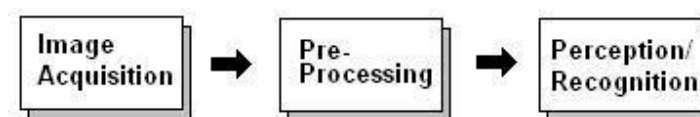
Figure 1: Simplification of stages for computer vision

The simplification of the computer vision problem shown in figure 1 does not suggest that this is what happens in humans, however it does provide a useful decomposition of work for a computational model.

2 Image Acquisition

Image acquisition is usually the first problem that students encounter when starting a computer vision project. Five or more years ago, this may have been more problematic because of the limited availability and high cost of devices such as digital video cameras and WebCams. Today however, the starting point for a computer vision project is much cheaper. Because of the growing interest in video conferencing, highly functional WebCams have become widely available and can be bought for very reasonable prices. From a practical standpoint, WebCams provide a quick and easy means of capturing images as they conform to the TWAIN specification. TWAIN defines a standard software protocol and API for communication between software applications and image acquisition devices [2]. The latest TWAIN specification can be downloaded from http://www.twain.org. All image acquisition devices that comply with the TWAIN specification can be accessed programmatically through the TWAIN API. The TWAIN drivers and API usually reside on the host operating system as native code, therefore, a Java wrapper API must be written to make the TWAIN API available to Java programmers. Luckily enough for Java programmers, a Java TWAIN API can be downloaded from http://www.gnome.sk and at the time of this writing the API is free for non-commercial use.

2.1 Using the Java TWAIN API

Once the Java TWAIN API has been downloaded and installed, students can begin to write simple programs that capture images from TWAIN compliant devices such as Scanners and WebCams. The TWAIN API is easy to use and does not require the student to have any prior knowledge of the actual hardware being used.

Listing 1 shows the main steps involved in capturing an image from a WebCam.

```
// Imports
import SK.gnome.twain.*;

// Step 1:Create a new TWAIN Object
Twain twain = new Twain();
// Step 2:Allow the user to select a data source (Scanner,WebCam,...)
twain.selectSource();
// Step 3:Pop up Twain user interface.
twain.setVisible(true);
// Step 4:Grab an image from the TWAIN device
image = Toolkit.getDefaultToolkit().createImage(twain);
```

Listing 1: Code sample showing steps to capture an image from a WebCam

In listing 1 the first step simply creates a Java TWAIN object that is used to control the image acquisition device. By calling the *set*XXX methods provided by the TWAIN class, the programmer can set various properties of the image acquisition device. A detailed listing of the available *setXXX* methods can be found in the Java TWAIN API documentation and further information about each of the device properties can be found in the TWIAN specification document [3]. Step 2 of listing 1 allows the user to select an image acquisition source. This is a useful feature as more then one TWAIN compatible source may be attached to the system. Figure 2 below, shows the source selection window that appears when this method is called.

Figure 2: TWAIN source selection window

Step 3 of listing 1 sets the visible property of the TWAIN object to *true,* causing a vendor specific UI to appear that allows the user to manipulate the image before it is captured. In some cases where this is not desirable, the value passed to this method can be set to *false* and the UI will not appear. However this feature will only work if the image acquisition device supports the TWAIN specification version 1.6 or higher. The code in step 4 of listing 1 shows how an AWT IMAGE object is returned from the TWAIN device. When this line of code is executed the TWAIN UI will appear and user can configure the image before it is captured. Figure 3 shows an example of the type of UI that is displayed when the *setVisible* method is used with a value of *true.*
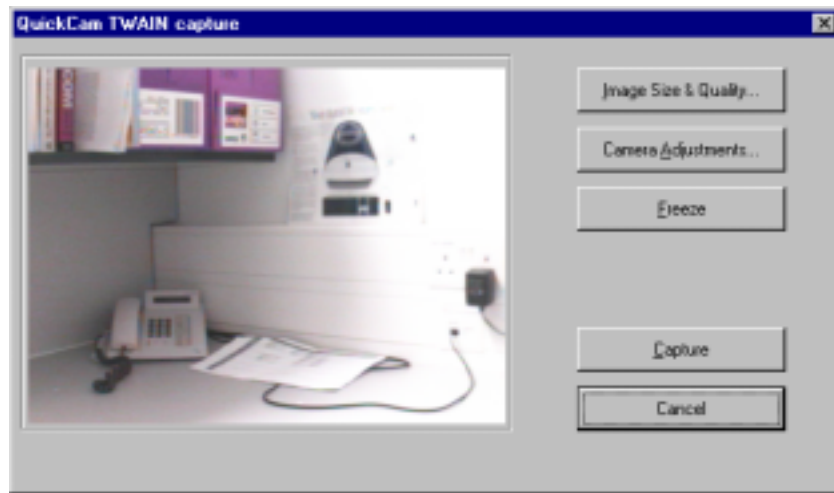
Figure 3: Example of TWAIN user interface

Once a valid AWT IMAGE object is returned from the TWAIN device, standard Java image handling techniques can be used to display or manipulate the image further. At this point in the computer vision process the pre-processing stage must take over and extract some useful information from the image.

3 Pre-processing

The pre-processing stage is concerned with extracting useful information from the image that can be passed on to the final perception or recognition stage. Many of the operations carried out at this stage are based on standard image processing techniques such as filtering and edge detection. These image-processing techniques have huge applications in the area of computer vision. By applying filters to an image we can change the colours used in an image, change the image contrast, brighten or darken the image or blur the image. Blurring is important because it can improve the results of the perception/recognition phase by reducing the amount of noise in an image. Images can contain noisy data for many different reasons, for example, there may be dirt on the camera lens, the lighting conditions may not be adequate or the quality of the image acquisition device may not be suitable for a computer vision application. Edge detection is really just a special type of filtering that can be applied to an image to find abrupt changes in pixel intensity values. This type of filtering is very important in the area of computer vision, as edges are what define the shape and structure of our world [4]. The implementation of the image processing techniques mentioned above could seen as a project in its own right and seeing as though there are many libraries that provide this functionality already, it is unnecessary to ask students to re-invent the wheel. This however, does not mean that having a deep understanding of the above image processing techniques is

unimportant; it simply means that students can participate in practical computer vision projects without having to get bogged down in implementation details. The JAI API (Java Advanced Imaging API) is one such library that can be used to carry out various image-processing operations and can be downloaded for free from http://java.sun.com/products/java-media/jai/.

3.1 Using the Java Advanced Imaging API

JAI is a set of classes that allow sophisticated high-performance image processing to be incorporated into Java applets and applications. JAI implements a set of core image processing capabilities including:

- Filtering
- Edge extraction
- Statistical operators
- Image I/O
- Region of interest control

- Logical image operators
- Fourier transforms
- Image interpolation
- Histogram operators
- Image tiling

The first stage in using the JAI API is to create a parameter block. Parameter blocks are used to hold information on how the image is to be processed. The code listing below shows how a parameter block can be constructed to blur an image.

```
// Imports
import javax.media.jai.*;
import java.awt.image.renderable.ParameterBlock;

// Step 1: Setup blur kernel
float[] blurData      = {0.0F,        1.0F/ 8.0F,  0.0F,
                          1.0F/ 8.0F,  4.0F/ 8.0F,  1.0F/ 8.0F,
                          0.0F,        1.0F/ 8.0F,  0.0F
};
// Step 2: Create a new kernel object using the blurring data
KernelJAI kernel = new KernelJAI(3,3,1,1,blurData);

// Step 3: Create a param block with the source image and blurring kernel
ParameterBlock paramBlock = new ParameterBlock();
paramBlock.addSource(pimage);
paramBlock.add(kernel);

// Step 4:Call the appropriate JAI create method using the paramBlock
PlanarImage processedImage = JAI.create("convolve",paramBlock);
```

Listing 2: Code sample showing how to construct a parameter block for blurring

Step 1 of listing 2 creates a mask that will be used to average the pixel values in the source image. Different values can be used here to create various effects or levels of blurring. Figure 4 shows the effects of applying two different levels of blurring to a source image that contains noise. Step 2 creates a KernelJAI object; the KernelJAI class takes the values of the blurData and creates a 3x3 matrix that will be used during the blurring process. Step 3 creates a parameter block object and adds the source image and the blurring kernel. Step 4 makes a call to the JAI.create method with the appropriate operation name. When this method is called, the image added to the parameter block is processed using the given values. The JAI.create method returns a new PlanarImage that is the result of processing the source image with the values contained in the paramBlock object.
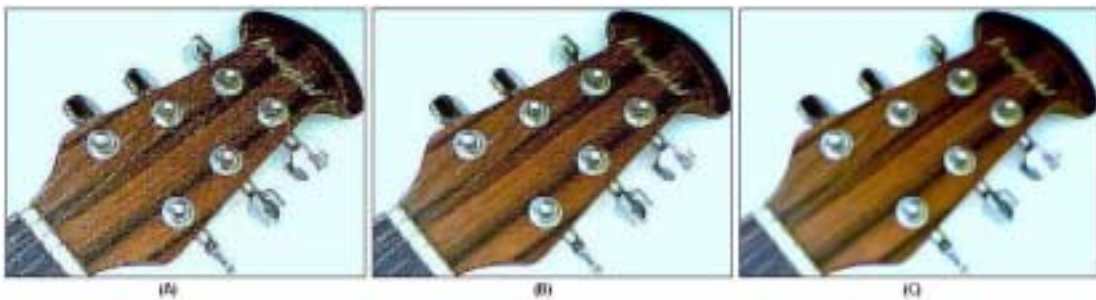

Figure 4: (A) source image with noise, (B) image blurred once (C) image blurred twice

Using the JAI API to detect edges follows almost the same steps as the blurring example in listing 2. The only difference is that instead of constructing a kernel that will average the intensity values in the image we must construct horizontal and vertical kernels that will detect abrupt changes in pixel intensity. Many different types of edge detection masks can be used for this purpose [5]; some popular masks are shown in figure 5.

The process of constructing the edge detection kernels is exactly the same as before, except this time it is necessary to construct both horizontal and vertical masks that can be added to the parameter block. Listing 3 shows how this can be done for the Sobel masks shown in figure 5.

0.0	**-1.0**	**0.0**
-1.0	**4.0**	**-1.0**
0.0	**-1.0**	**0.0**

Simple horizontal

0.0	**0.0**	**-1.0**
0.0	**1.0**	**0.0**
0.0	**0.0**	**0.0**

Roberts horizontal

1.0	**0.0**	**-1.0**
2.0	**0.0**	**-2.0**
1.0	**0.0**	**-1.0**

Sobel horizontal

0.0	**-1.0**	**0.0**
-1.0	**4.0**	**-1.0**
0.0	**-1.0**	**0.0**

Simple vertical

-1.0	**0.0**	**0.0**
0.0	**1.0**	**0.0**
0.0	**0.0**	**0.0**

Roberts vertical

-1.0	**-2.0**	**-1.0**
0.0	**0.0**	**0.0**
1.0	**2.0**	**1.0**

Sobel vertical

Figure 5: Selection of popular edge detection masks

```
// Step 1: Setup horizontal & vertical masks
float[] sobel_h_data  = { 1.0F,  0.0F, -1.0F,
                          2.0F,  0.0F, -2.0F,
                          1.0F,  0.0F, -1.0F
};
float[] sobel_v_data  = { -1.0F,  -2.0F, -1.0F,
                           0.0F,   0.0F,  0.0F,
                           1.0F,   2.0F,  1.0F
};

// Step 2: Create horizontal & vertical JAI kernel objects
KernalJAI kern_h = new KernelJAI(3, 3, sobel_h_data);
KernalJAI kern_v = new KernelJAI(3, 3, sobel_v_data);

// Step 3: Create a param block with the source image and Sobel kernels
ParameterBlock paramBlock = new ParameterBlock();
paramBlock.addSource(pimage);
paramBlock.add(kern_h); paramBlock.add(kern_v);

PlanarImage processedImage = JAI.create("gradientmagnitude", paramBlock);
```

Listing 3: Code sample showing how to create Sobel masks

Once the JAI kernel objects and parameter block have been created, the JAI.create method can be called to process the source image. This time the JAI.create method is called with the operation name "gradientmagnitude", this will cause both the horizontal and vertical masks to be used during the operation. Figure 6 shows the results of using the Simple, Roberts and Sobel masks respectively.
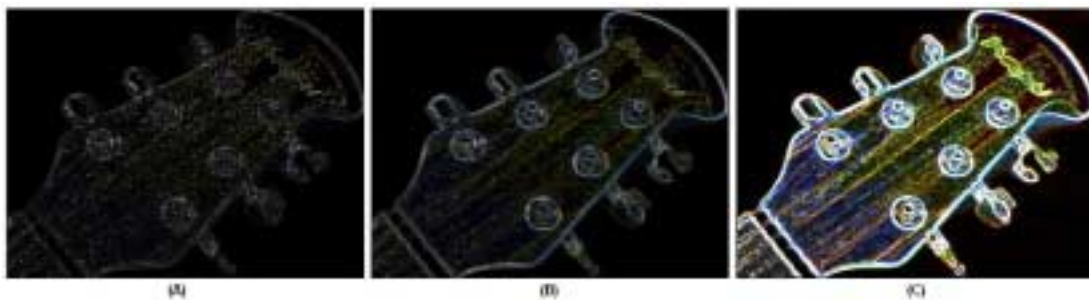

Figure 6: (A) Simple mask, (B) Roberts mask (C) Sobel mask

3.1 Conversion to raw data

The final phase of the pre-processing stage is to convert the edge image into raw data. This is necessary because the perception/recognition stage cannot deal directly with the images produced by the JAI API. There are many ways in which this can be done; one simple approach is to apply a threshold operator on the edge image. Since each pixel in the edge

image has an intensity value between 0 and 255, we can isolate those pixels that are part of an edge and those that are part of the background. For example, the edges shown in figure 6 diagram C) are primarily made up of pixels whose intensity values are greater then 150, so it is possible to construct a raw data file where a pixel that is higher then the threshold value of 150 is represented by a 1 and a pixel that is lower then the threshold value is represented by a 0. Figure 7 shows how this can be done for an edge image of a handwritten alphabetic character.
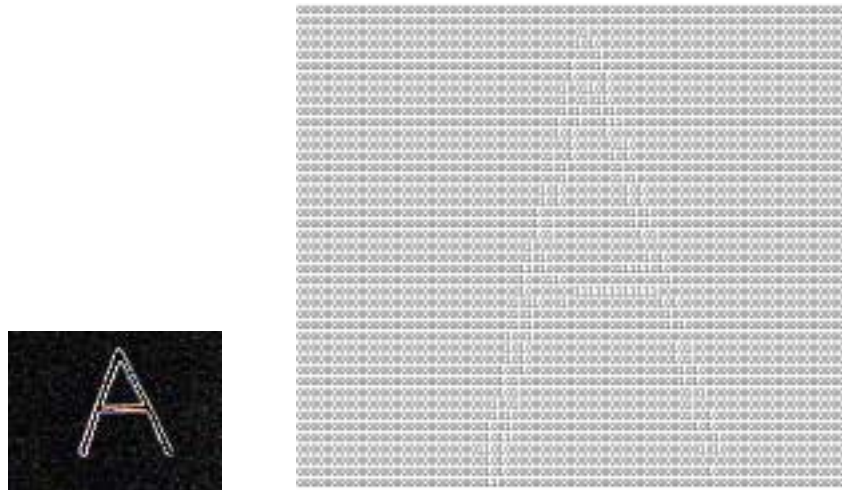
Figure 7: Raw data output of handwritten character

Once the raw data for the edge image has been produced it can be passed onto the perception/recognition stage. It is at this stage that the computer vision system attempts to make sense of the data. There are many different techniques that can be used in order to make sense of this raw data and the choice of technique will depend on the requirements of the computer vision system being developed. For the purpose of this paper we will focus on one such technique that is particularly adept at recognising patterns within the raw data.

4 Perception/Recognition

Since the 1980's it has been has been widely accepted that neural networks are excellent at solving pattern recognition problems. Although a full description of neural networks is beyond the scope of this paper some concepts and terminology must be introduced that will allow for a description of how they can be used in computer vision projects. Neural networks are vastly simplified computational models of the biological human neuron [6]. Each biological neuron is made up of three main parts, the axon, dendrites and the cell body or soma. The dendrites detect electrical impulses from neighbouring cells a pass these impulses

into the cell body. The neuron is said to 'fire' if the sum of arriving impulses is above a certain threshold value. When the neuron fires it sends another electrical impulse down its axon, this electrical impulse is then detected by the dendrites of other neighbouring cells. In 1958 Frank Rosenblatt developed a simple computation device called the Perceptron that was based on the work carried out by Hebb, McCulloch and Pitts in 1943. Rosenblatt's work still forms the basis of the neural networks that are used today. Figure 8 shows the similarities between a biological neuron and Rosenblatt's Perceptron.
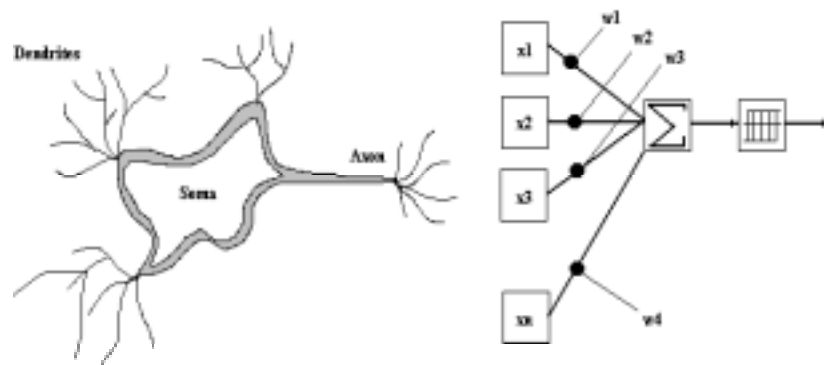

Figure 8: Comparison of biological neuron and Perceptron

Each Perceptron has a number of inputs that are connected to a summation unit via weighted links. The summation unit calculates the weighted input and passes the result to a threshold unit. The threshold unit outputs a 1 if the weighted input is above or equal to some value, otherwise it outputs 0. A learning algorithm is usually used to optimise the weight values associated with each link in order to give the desired result. Complex networks can be built by connecting many of these simple computational structures together, however, implementing these networks in software can be quite difficult and time consuming. The approach taken by many neural network researchers is to develop a software library of classes that can be used and modified over time to speed up the implementation and testing of new types of networks. Computer science students studying at ITB have access to one such library, the library is comprised of a set of Java classes that can be used to implement various kinds of neural networks [7]. The library provides a set of primitive neural network classes that students can use to build their own networks or alternatively, students can use a set of existing network classes such as the BackProp (backpropagation) class to solve problems [8].

4.1 Using the Java Backpropagtion Class

By using the Backprop Java class, students can concentrate on the problem the network is attempting to solve rather then the implementation details of the network itself. Students can

build and train a backpropagation network by constructing a topology and training file and passing these files as parameters to a Backprop object. The Backprop object also provides *learn* and *run* methods that the student can call to operate the network in training mode or in normal mode. Using the network in training mode will cause the Backprop object to read the training file and begin readjusting its weights using the backpropagtion algorithm. Once the network has been trained and its weights have been saved, it can be used in normal mode to solve problems such as pattern recognition. Although a lot of the more difficult implementation details are taken care of by the neural network Java classes, the student is still required to build training files and decide on an appropriate topology for the network. Figure 9 shows the contents of a typical topology and training file.

```
// Topology file                          // Training file
                                          0 a1.trn 1 0 0 0
0.45  // Learning rate parameter          1 a2.trn 1 0 0 0
0.9   // Momentum term                    2 b1.trn 0 1 0 0
0.1   // Tolerance                        3 b2.trn 0 1 0 0
3     // Number of network layers         4 c1.trn 0 0 1 0
4800  // Number of input nodes            5 c2.trn 0 0 1 0
100   // Number middle layers nodes       6 d1.trn 0 0 0 1
3     // Number of output nodes           7 d2.trn 0 0 0 1
```

Figure 9: Topology and training files used by Backprop class

The topology file allows the student to set parameters associated with the backpropagation learning algorithm such as the learning rate and momentum term. The student can also define the number of layers that the network contains and the number of units on each layer. These values can be adjusted after each training session to improve the overall performance of the network. The construction of the training file is less straightforward and will depend heavily on the problem the network is required to solve. The training file shown in figure 9 is used to present the network with some positive examples of alphabetical characters. Each line of the training file must start with a pattern number followed by the name of a file that contains a raw data representation of the letter to be learned (see figure 7), finally each line of training file must end with the desired output for that pattern. The desired output data informs the network which output unit should fire for each pattern. So for example, if the network is presented with an A, the first output node should fire, if the network is presented with a B, the second output unit should fire, and so on. Once the topology and training file have been constructed the student can use the Backprop class by constructing a Backprop object and passing the topology and training file as parameters. The code listing below shows how this can be done.

```
// Build the network
Backprop bp = new Backprop("topology.txt", "alphabet.trn");
bp.createNetwork();

// Train the network and save its weights
bp.learn()
bp.saveWeights("weights.dta");

// Load the saved weights and a new pattern to test the network
bp.loadWeights("weights.dta");
bp.resetInputPatterns("test.trn", 1);
bp.run();
```

Listing 4: BP network construction, training and running

Using the Backprop class as part of a computer vision project requires the student to write two separate programs, a training program and an application program. The training program must acquire images that can be presented to the network during the training phase, and the application program must use the weights saved during the training phase to recognize patterns that the network has not seen before. The software libraries introduced in sections 2 and 3 of this paper can be applied to both the training program and the application program.

5 Computer Vision Application

This section introduces a demonstration application that was developed using the software libraries introduced in sections 2, 3, and 4. The application is capable of recognising three categories of everyday objects, Pens, Apples and Phones. A flowchart of the program execution can be seen in figure 10.
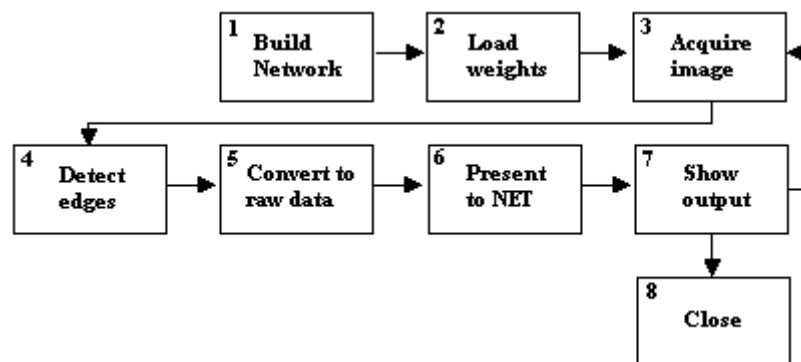
Figure 10: Flowchart showing program execution

Stage 1: The program begins by constructing the neural network that will be used during stage 6. In this example a 3-layer network with 4800 inputs, 150 middle units and 3 outputs is used. Each of the 4800 inputs represents one pixel of the input pattern. Each of the 3 output

units represents the category that the input image belongs to. Figure 11 shows the topology of the network used.

Stage 2: This stage of the program execution loads the weight values of a previous backpropagation training session. In this example the network was trained using 39 patterns containing 12 examples of pens,13 examples of apples and 14 examples of phones.

Stage 3: This stage uses the Java TWIAN API to acquire an input image from a USB WebCam attached to the system.
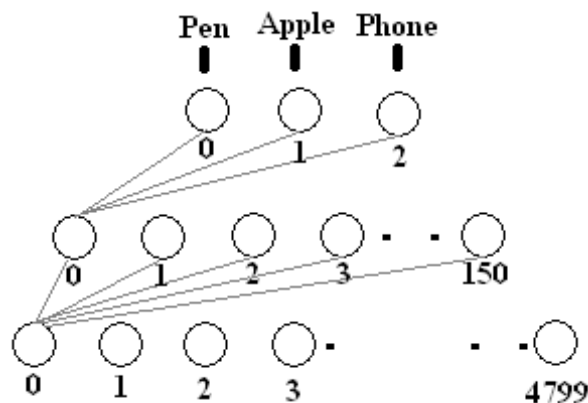
Figure 11: Neural network topology

Stage 4: This stage extracts an edge image from the input image using the Sobel masks introduced in section 3 of this paper.

Stage 5: Once an edge image has been produced it is converted into raw data using the thresholding operator discussed in section 3. Once the raw data has been produced it is written to a file called 'edge.trn'. This file will then be used as input to the network.

Stage 6: In this stage the information contained in the edge.trn file is presented to the network. As the network is operating in normal mode it only has to deal with one image at a time. The network takes each input value and loads the appropriate input unit with that value. It then feeds the input values forward and set the values on the output nodes.

Stage 7: Once the network has completed its feed-forward cycle the program checks the value of each output node to see which one is the highest. For example, if output node 0 has the highest values the network has recognised the input image as a pen, if the second output node has the highest value then the network has recognised the input image as an apple, and so on. Once the highest values have been determined the program notifies the user of the

result. From here the user can choose to quit the application or test the network again by acquiring another image.

Figure 12 shows the application's user interface. The user interface consists of a panel to display the edge image, a panel to show the network output and a panel that contains buttons that allow the user to acquire an image, recognise the current image and close the application. The user can test the network by clicking the 'Capture' button to acquire an image and then clicking the 'Recognize' button to present the pattern to the network. When the user clicks the 'Recognize' button, the network output will be displayed by switching on the appropriate radio button.

Input correctly identified as pen   Input correctly identified as phone
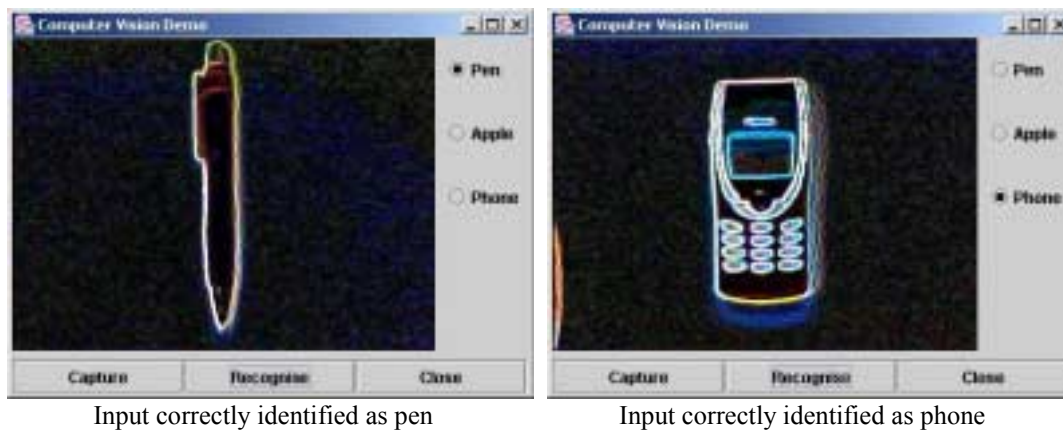
Figure 12: Application user interface

6 Conclusion

The application discussed in section 5 of this paper integrates the functionality of the software libraries introduced in sections 2, 3, and 4. The demo application follows the decomposition of work discussed in section 1 and applies the appropriate software tools at each stage. Although the software libraries used cut down on the amount of coding required, integration of each stage of the computer vision model is left up to the student. The demo application leaves a much room for improvement and is by no means a fully blown computer vision application. For example, the range of objects the application can recognise is limited and only very basic image processing is carried out during the pre-processing stage. The intention here is that it is up to the student to improve the system.

A list of possible areas where the student could improve the system is given below.

- Adjustment of WebCam parameters
- Colour analysis
- Input image orientation
- Dealing with multiple images

- Extraction of higher order data from images

- Better raw data conversion
- Optimisation of learning parameters
- More complete training sets
- Implementation of more sophisticated neural networks
- Experimentation with other problem domains

Overall the choice of software libraries has been successful, the libraries deliver the functionality required for computer vision projects and are easy to use. One possible improvement as regards the choice of libraries would be to use the JMF (Java Media Framework) for image acquisition as this library provides a pure Java interface to devices such as WebCams and can be used to capture time-based data, i.e. Digital Video.

References

[1] **Dean, Allen & Aloimonds (1995).** T. Dean, J. Allen, & Y. Aloimonds, Artificial Intelligence: Theory and Practice, *Addison Wesley*, 9.3:415-417.

[2] **TWAIN Working Group (2000).** TWAIN Working Group, TWAIN Specification Version 1.9, *TWAIN Group*, 1:1-3.

[3] **TWAIN Working Group (2000).** TWAIN Working Group, TWAIN Specification Version 1.9, *TWAIN Group*, 9:340-498.

[4] **Parker (1997).** J. R. Parker, Algorithms for Image Processing and Computer Vision, *John Wiley & Sons, Inc*, 1:1-7.

[5] **Marr, Hildreth (1980).** D. Marr, E. Hildreth, Theory of Edge Detection, *Proceedings of the Royal Society of London*, Series B. Vol. 207:187-217.

[6] **Arbib (1995).** M. A. Arbib, Introducing the Neuron, *Handbook of Brain Theory. MIT Press*, Part I.:4-11.

[7] **Sheridan (2000).** S. Sheridan, Non-Deterministic Processing in Neural Networks, *ITB Journal, Issue 2*:4-20.

[8] **Werbos (1995).** P. Werbos, Backpropagation: Basics and New Developments, *Handbook of Brain Theory. MIT Press*, Part III.:134-139.