Intelligent Fish Classification in Underwater Video

Joseph Anderson
Saint Vincent College
joe.anderson@email.stvincent.edu

Brian Lewis
Clemson University
blewis593@msn.com

Colin O'Byrne
Harvey Mudd
cobyrne@hmc.edu

November 15, 2011

Abstract

We present an exposition on the evolution of image processing in the context of gray-level images with the goal of applying such techniques to detect and identify fish in natural, underwater environments. Automatic image thresholding methods are explained, implemented, and applied - in conjunction with bac subtraction - to achieve accurate segmentation. A method of improving segmentation via an edge thinning procedure is presented and studied. After segmentation, data determined to represent a fish is saved and used for classification. Biometric principles such as WARP and Gabor filters are applied to extract feature data and machine learning techniques are used to identify the processed objects. Boltzmann machines, convolutional neural networks, and deep belief networks are trained to perform classification. The system performance is analyzed and future improvements are planned and described.

Contents

Preface

This work was conducted as part of a Research Experience for Undergraduates (REU) and the University of New Orleans funded by a grant from the National Science Foundation. The intention of the following report is to provide a comprehensive account of the material studied and problems addressed by the authors.

This work was written with the assumption only that the reader has knowledge of fundamental probability and statistics - Chapter 1 will provide the background necessary to Chapter 2 and Chapter 3. Chapters 2 and 3 are the record of the specific work completed during the course of the REU. Chapter 4 addresses the possibility and course of future study and publication.

In the experimental results, the reader should note that all experiments were performed on a Dell Optiplex 745 running Windows XP. The computer had an Intel Core 2 CPU 6700 which ran at 2.66 GHz and 1.57 GHz. There also was 1 GB of RAM. In this environment all experiments were conducted in Matlab R2009a.

We would like to thank Madhuri Gundam, Arjun Joginipelly, Milton Quinteros, Dr. Huimin Chen, Dr. Edit Bourgeois, and especially Dr. Dimitrios Charalampidis for their invaluable guidance and support for the span before, during, and after the project. We would also like to thank National Marine Fisheries Center for providing the video data, and also the Lousiana Board of Regents and Stennis Space Center for providing the funding for a related research performed by faculty and students in the Electrical Engineering and Physics programs at University of New Orleans.

New Orleans
November 15, 2011

Chapter 1

Introduction

The scientific community has the ability to capture and store images in digital form; this data has been subjected to similar exhaustive information extraction as have our other digital resources. We take for granted our inherent, human ability to interpret the visual data we receive in meaningful ways and we now strive to develop systems capable of such extraction and interpretation. Innovative image processing techniques hold the potential to radically change the way scientists and engineers find and approach problems. For decades, image processing has been recognized and studied as it has become more practicable. The benefits of such studies extend as the field itself advances, and they benefit varied applications such as medicine, security, entertainment, among others.

The specific applications of image processing and modeling vastly differ based on the goals of the project and the type of data provided. For instance, different approaches must be considered for analyzing medical images than for digitizing printed materials or for detecting defects in industrial products. These diverse applications must apply concepts including foreground detection, noise reduction, and feature extraction. When incorporating images taken from a video feed, one might have to implement tracking and occlusion compensation. Tracking strategies would be used to know that a segmented "blob" in one pictures is the same one in a different position at a different time in the video sequence. Occlusion occurs when an object is obscured or blocked by another object in the foreground. Naive processing methods would detect the object twice - before and after it has been occluded. These multitudinous hurdles become motivation for more novel approaches in segmentation and processing algorithms that account for variations in data employed by the system. Steps have even been taken to develop automated methods of algorithm selection or user-assisted selection.

Unfortunately, other issues arise in image processing - statistical or otherwise - that inhibit effective knowledge from being attained by our methods. Perhaps the most daunting has been computational cost: many approaches require the use of every information unit (typically pixel value) of an image; others will gather histograms to generate probability distributions of image qualities and be analyzed for patterns in background and foreground. Filtering techniques are employed that are required to operate over the entire scope of an image and need to store copious auxiliary data. Other requirements of satisfactory image processing include mathematically well-defined images which are difficult to gather from *in natural* data sets. Such images would be those which have distinctly multi modal histograms or have well-distributed noise[10].

In general, a preprocessing step may be applied in order to reduce image noise and other

artifacts. After an image has gone through preprocessing, it may be necessary to extract the pertinent information from the image in order to execute some sort of detection or classification on the images. One of the most common approaches is by identifying and extracting features from the image, known as feature extraction. These features are unique identifying marks and can represent a number of different qualities about the image. These features are always represented by numbers in the form of a feature vector, and each feature vector is associated with an image, an image block, or even a single pixel. A feature can range from something as simple as the pixel value at a specific location to a set of more complex image descriptors, such as the Fourier descriptors.

Our use of feature extraction allowed us to identify distinct species of fish based on a single image. The original data collected was in video form, and was provided to us in still image form. By trying to identify the fish species based on a single image, the dependency on other areas such as fish tracking for identification would be reduced.

These features must then be classified and the species of fish determined. Artificial neural networks are a very powerful approach for classifying data. Neural networks are trained by repeatedly presenting the network with representative data and the desired network output, and the network parameters are updated so that network output matches the target value. Several approaches have been employed in the literature for the purpose of parameter updating. One of the most frequently used methods is gradient descent.

The aforementioned topics were the focus of this research on image processing in the context of fish identification and classification. This report is structured to reflect the organization that was used in the work's execution. The research described in this report was divided in three main components, namely automatic thresholding, feature extraction, and classification. For the sake of perspicuity, these three matters will be introduced independently in the systematization of the associated research. Finally, the coalescence of our various endeavors is presented with our experimental results and the possible future progress that can be studied *de futuro* in correspondence with the advisors and project leaders.

1.1 Background Subtraction

In the ideal case, a sequence of images taken of a moving object will share a common, static, background environment; such images may be sequentially analyzed and the common background derived from the temporal progression. In practice, however, "real" images are full of noise, occlusion, and varying illumination. The basic implementation of background subtraction is as it sounds - from a quantized version of an image (typically a matrix), subtract the computed matrix of background intensities. This will yield a matrix where higher values are analogous to objects moving through the environment blocking the background and having contrasting values - a concept relateable but not equivalent to how our minds actually distinguish objects.

In order to produce a more reliable, up-to-date background model however, techniques have been introduced involving statistical methods to generate data that is recent to the current frame being analyzed, so that illumination and shadows may be included in the non-object class of the image. Because of the complexity of such methods, "background subtraction" has become more theoretical and has been researched more than might be literally inferred from its name [32].

1.1.1 Running Gaussian Average

An early approach to background subtraction was by Wren *et al.* in [45] where they considered each pixel as being independent and fitting a Gaussian probability distribution function (pdf) on that pixel's last n values [32]. A *support map* was created to model each pixel's relation to a detected blob and the related statistics were updated recursively using the adaptive filter

$$\mu_t = \alpha y + (1 - \alpha)\mu_{t-1} \tag{1.1}$$

The recursion in (1.1) allows the system to compensate for changes in scene illumination. These considerations require a non-trivial amount of time to compute. In order to improve this, the sampling rate may be adjusted. However, a lower update rate delays the detection of a dynamic background.

1.1.2 Temporal Median Calculation

In [24], a background model was calculated using the mean of the previous n frames. The model was subtracted from the current image in the set. Further improvements were contributed in [7] by computing the median on n_k sub frames and a weighted version of the old mean value image. This method suffers primarily from a lack of deviation measure for adapting the background-subtracted result. Sufficient memory for the pixel buffers must also be provided, but is less of a problem with newer technology and can be easily attained for more "typical" buffers.

Several approaches to selection of the median have been proposed. Histogram based selection as in [14] uses the histogram values of successive frames to efficiently adjust the running median without having to recalculate for the entire frame buffer. Other algorithm methods such as divide-and-conquer in [8] have been proposed and shown to be efficient selection-based median methods. Recently, a faster temporal median method developed in [15] takes advantage of known repetition of a given median and the pixel correlation between consecutive frames. We will discuss the latter strategy in the context of automatic thresholding.

Temporal median filters are effective in identifying rapid and continuous scene change. This method can be profitable in surveillance of moving traffic or monitoring athletic performance. In our case, the temporal median was chosen at an early stage - as the result of some assumption - to be the implemented method of background subtraction. This method, along with the running Gaussian average, is claimed to be the most practicable when compared with various other methods in [32].

1.2 Thresholding Methods

In the myriad problems and applications of image processing, a system must be developed to designate pixels of various regions as either background, foreground, or any one of a specified number of other layers. These layers are typically objects known to be occluded by a foreground. In this exposition we focus on gray scale images only - in the context of color images, we assume that they have been quantized into a gray scale composition. For bi-level thresholding, a function is constructed which assigns to any given pixel a constant (e.g. 0 for background and 1 for foreground) to designate its class membership relative to a threshold. This goal can be easily extended to multilevel thresholding.

The foundations of gray level thresholding were advanced by Otsu's method in [29] in which they use a class-variance technique that maximizes the separability of classes determined by the threshold. This method makes use of the image histogram [1] to generate probability functions and use the respective zeroth- and first-order means. Yanowitz [48] used locally adaptive methods that - similarly to those in [38] - proved to be one of the more effective local methods available. A method by Kittler in [20] uses cluster analysis - a form of histogram analysis - that evaluates the peaks of a histogram which are assumed to be distinct. According to the survey in [38], this method is effective in the context of NDT [2] images. Cheng *et al* in [6] use the "fuzzy entropy principal." This approach uses a two-dimensional histogram that, in addition to pixel intensity, takes into account average intensities. A point on this (surface) histogram would represent the number of pixels at a particular intensity that also have a particular average. From this, an entropy is defined and the optimal threshold is determined to be at a point of "maximum fuzziness." More recently, Tao *et al* in [42] use a graph cut method with improved efficiency over previous methods. As will be seen later, this method by Tao *et al* provides our project with the most reliable results. Xiao *et al* provide a new method with a two-dimensional histogram to define a locally-based *gray-level spatial correlation* and uses this property to compute image entropy [47]. In [2], Ajlan *et al* propose a cross-entropy solution using a gamma distribution function. Even now, more thresholding methods are being developed and applied.

Complications arise when formulating an approach to choosing such a threshold and have been thoroughly studied in many contexts. Because of varying needs and datasets, different methods are more well-suited to independent applications. The pre- and post-processing steps taken must also be carefully chosen to complement the thresholding technique and allow for more accurate segmentation. Noise, for instance, can have adverse effects on thresholding methods that use *global* as opposed to *local*[3] data. This distinction impacts the core strategies of threshold algorithms.

1.2.1 Histogram-Based Thresholding

We have already introduced briefly thresholding methods that operate over an image histogram with $L+1$ gray levels - denoted h -where $h(i)$ denotes the number of pixels at value $i \in \{0, 1, 2, ..., L\}$. From this data, a probability function can be constructed

$$p(i) = \frac{h(i)}{\sum_{n=0}^{L} h(n)} \tag{1.2}$$

In effect, (1.2) computes the (equally likely) probability of a single pixel having a particular value. Similarly, for a single, particular valued threshold $t \in \{0, 1, 2, ..., L\}$, one can dichotomize the image based on this value where the background, say $C_0 = \{(x, y)|f(x, y) \leq t\}$ and the foreground $C_1 = \{(x, y)|f(x, y) > t\}$ - where $f(x, y)$ is simply the gray-level intensity of the image at position (x, y). Of course, these definitions are assuming that the background is darker than the foreground - which will be the case in an image representing the difference between an image and the calculated

[1] A histogram is essentially a function representing the number elements in a set that possess common qualities. In the context of image processing, an image histogram is typically a discrete-valued function representing the number of pixels at each possible gray level.

[2] Non-destructive testing

[3] Global image data is information gathered from the entire image. Local data is information extracted only from a small subset or window at a time

background. These will serve as the bases for histogram-based thresholding methods and many of the methods discussed in this chapter.

As is common in statistical approaches, we will use the moments of the data and also variance of the classes C_0 and C_1. To obtain the moments, we take

$$\omega(t) = \sum_{i=0}^{t} p(i) \tag{1.3}$$

and

$$\mu(t) = \sum_{i=0}^{t} ip(i) \tag{1.4}$$

as the zeroth- and first- order moments of the histogram to the $t^{\text{th}+1}$ level.

From these we obtain the probability of individual class-occurrence [4]

$$\omega_0 = Pr(C_0) = \sum_{i=0}^{t} p(i) = \omega(t) \tag{1.5}$$

and

$$\omega_1 = Pr(C_1) = \sum_{i=t+1}^{L} p(i) = 1 - \omega(t). \tag{1.6}$$

The class mean levels of C_0 and C_1 are given respectively by

$$\mu_0 = \sum_{i=0}^{t} ip(i|C_0) = \sum_{i=0}^{t} \frac{ip(i)}{\omega_0} = \frac{\mu(t)}{\omega(t)} \tag{1.7}$$

and

$$\mu_1 = \sum_{i=t+1}^{L} ip(i|C_1) = \sum_{i=t+1}^{L} \frac{ip(i)}{\omega_1} = \frac{\mu_T - \mu(t)}{1 - \omega(t)} \tag{1.8}$$

Also, the total mean of the image is simply

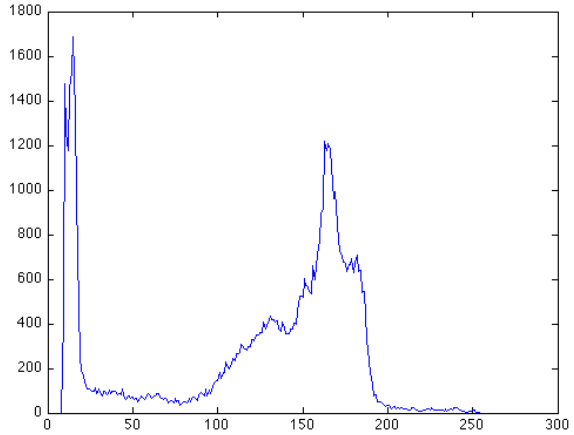
$$\mu_T = \sum_{i=0}^{L} ip(i) = \mu(L). \tag{1.9}$$

According to the method in [29], these results are then used to derive the optimal threshold which minimized the between-class variance:

$$T_{opt} = \underset{0 \le t \le L}{\arg\max}\, \sigma_B^2(t) = \underset{0 \le t \le L}{\arg\max}\, \frac{[\mu_T \omega(t) - \mu(t)]^2}{\omega(t)[1 - \omega(t)]} \tag{1.10}$$

composed only of the definitions from (1.3), (1.4), and (1.9). In the application of this theory, one may restrict the variable $t : 0 \le \omega(t) \le 1$ so that the denominator of (1.10) will be greater than 0. Since it is easily seen that on this restricted range, t will always take on a positive and bounded value, the maximum may always be obtained. The result of the threshold calculation of T_{opt} in

(a) Original Image



(b) Cameraman Histogram (intensity vs. number of pixels)

Figure 1.1: Cameraman image and its histogram

(1.10) is actually maximizing a discriminant criterion which represents the separability of gray level classes.

Taking advantage of the apparent simplicity with which this threshold may be calculated, Liao *et al.* develop an algorithm in [23] to efficiently extend Otsu's method to multilevel thresholding. This method will be later discussed at length in Chapter 2, but it should be noted that this can be accomplished with the use of lookup tables preventing the recalculation of means and variances at different points of the histogram.

Another histogram analysis method proposed by Ramesh *et al.* in [35] is used to calculate and minimize one of two error functions:

$$E(t) = \sum_{i=0}^{t}(i-\mu_0)^2 + \sum_{i=t+1}^{L}(i-\mu_1)^2 \tag{1.11}$$

measuring a sum of square errors or the sum of variances:

$$E(t) = \frac{1}{N_0-1}\sum_{i=0}^{t}(i-\mu_0)^2 + \frac{1}{N_1-1}\sum_{i=t+1}^{L}(i-\mu_1)^2. \tag{1.12}$$

In (1.12), N_0 is the total number of pixels in C_0 and N_1 is the total number of pixels in C_1. The method in [35] calculates the optimal threshold as

$$T_{opt} = \underset{0\le t\le L}{\arg\min}\, E(t) \tag{1.13}$$

While the sum of square errors is computationally more desirable, the variance method is shown in [35] to achieve a "better" threshold by their criterion of correctness. To establish this metric,

[4]In our case: the probability that a pixel will be part of a particular class such as foreground or background

they employ a user-assisted comparison of correctly threshold pixels to incorrectly thresholded ones. They also assert that when the image histogram tends to be unimodal, the method achieves even more accurate threshold results than previous algorithms.

Overall, histogram-based approaches have inspired and reinforced past and current thresholding methods. Authors have incorporated histogram analysis (as will be seen) into methods that focus on entropy calculation, fuzzy methods, and clustering.

1.2.2 Cluster-Based Thresholding

Continuing the approach of statistical modeling, thresholding techniques like that of Kittler and Illingworth in [20] focus on creating a criterion function to minimize over the threshold. This is similar to the approach of Ramesh in [35] except that the clustered data is modeled against two Gaussians.

In Kittler and Illingworth's method [20], the common assumption that the variances will be equal is removed and a criterion function is created that minimizes the error of the Gaussian density-fitting problem. Similar to in the histogram-based methods, a variance is calculated as

$$\sigma_0^2 = \frac{\sum_{g=0}^{t} (g - \mu_0)^2 h(g)}{\omega_0} \tag{1.14}$$

and

$$\sigma_1^2 = \frac{\sum_{g=t+1}^{L} (g - \mu_1)^2 h(g)}{\omega_1}. \tag{1.15}$$

These variances will serve as density parameters with ω_0 and ω_1. From these parameters, a criterion function $\mathrm{J}(t)$ is constructed[5] as

$$\mathrm{J}(t) = 1 + 2[\omega_0 \log \sigma_0 + \omega_1 \log \sigma_1] - 2[\omega_0 \log \omega_0 + \omega_1 \log \omega_1]. \tag{1.16}$$

Because older methods such as those in [28] require solving a rather complicated quadratic equation using histogram estimation, the proposed criterion function improves both the overall simplicity and is trivially minimized for bilevel thresholding. Kittler and Illingworth's method may also be extended to multilevel thresholding and, when applied iteratively, further reduce the criterion function.

Clustering methods are not immune to pitfalls, however, and they can produce a nonsensical threshold under several conditions: when the criterion function $\mathrm{J}(t)$ converges to an internal minimum, it does not guarantee its uniqueness also, because the algorithm may tend to converge to the intensity boundary points of the image in certain cases. The most typical solution for such errors is to repeat the algorithm several times starting at different thresholds and comparing the results [20].

1.2.3 Entropy-Based Thresholding

Originally conceived by Claude Shannon in [39] - originally published in 1948 - the entropy in information theory is a measure of uncertainty associated with the probability of a set of random

[5] For the full steps of the derivation, see [20]

variables. He treats a message as a sequence of uniformly distributed random variables. From such assumptions, he derives a theorem to represent the possibility of error in the message as

$$H = -\sum_{i=1}^{n} p_i \log p_i \tag{1.17}$$

where each p_i is a particular probability in the sequence. Shannon clearly models the entire concept after that of the entropy in statistical mechanics developed by Boltzmann. One can easily see that $H = 0$ (the outcome is certain) if and only if all but one probability in the set is zero. This would mandate that the only non-zero probability be a certain event (or have a value of 1) - resulting in zero uncertainty.

An excellent exposition can be found in the method of Guo *et al.* [11] where the Shannon entropy is adopted into an adaptive particle swarm algorithm. In this implementation, the actual entropy concept is faithful to the original theorem introduced in [39]. To use the Shannon entropy in this way, we define the distributions of C_0 and C_1 to be

$$C_0(i) : \frac{p(i)}{\omega_0} \quad \text{for } i = 0, 1, 2, ..., t \tag{1.18}$$

and

$$C_1(i) : \frac{p(i)}{\omega_1} \quad \text{for } i = t+1, t+2, ..., L. \tag{1.19}$$

The Shannon entropy of these classes would then be

$$H(C_0) = -\sum_{i=0}^{t} \frac{p(i)}{\omega_0} \log \frac{p(i)}{\omega_0} \tag{1.20}$$

and

$$H(C_1) = -\sum_{i=t+1}^{L} \frac{p(i)}{\omega_1} \log \frac{p(i)}{\omega_1}. \tag{1.21}$$

After defining $\phi(t) = H(C_0) + H(C_1)$, the optimal threshold would be one that maximized a sum of entropy between the two classes C_0 and C_1:

$$T_{opt} = \arg\max_{t} \phi(t). \tag{1.22}$$

The above method is easily extended in [11] to multilevel implementation minimizing a function

$$\phi(t_1, t_2, ...t_k) = \sum_{i=0}^{k} H(C_i)$$

to threshold the image into $k+1$ levels. One can easily see that in order to segment $k+1$ levels, only k thresholds are needed. The separability of the entropy functions provides, as seen in Chapter 2, for an algorithmically simple implementation. However, because a linear search is used to find the maximum threshold argument, the computational cost becomes $O(\mathrm{n}^k)$

While [11] uses the original definition of the Shannon entropy, Xiao *et al.* uses the combination of a two-dimensional histogram and an associated weight function to adjust the class entropy based

on local pixel similarities. In [47], Xiao *et al.* construct a two-dimensional histogram using a gray-level spatial correlation (GLSC) calculation. This GLSC is a measure of how many pixels (m) in a neighborhood of a predefined size (N×N) are within a predefined range of gray values (ζ). The resulting histogram $h(i,m)$ can then be used to construct a probability distribution representing the probability that any given pixel will be have intensity i and have m similar neighbors. To construct the probability function as in [47], we take

$$p(i,m) = \frac{h(i,m)}{\sum_{k=0}^{L}\sum_{m=1}^{N^2} h(k,m)}. \tag{1.23}$$

where $m \in \{1,2,3,...,N^2\}$ and N being a positive integer. Now to weight the entropy, we construct a function

$$W(m,N) = \frac{1+e^{\frac{-9m}{N^2}}}{1-e^{\frac{-9m}{N^2}}}. \tag{1.24}$$

We observe that the authors in [47] intended to apply more weight to entropy corresponding to GLSC with lower probability. The entropy of pixels at gray level k with a lower number of similar neighbors are multiplied by the greater weight.

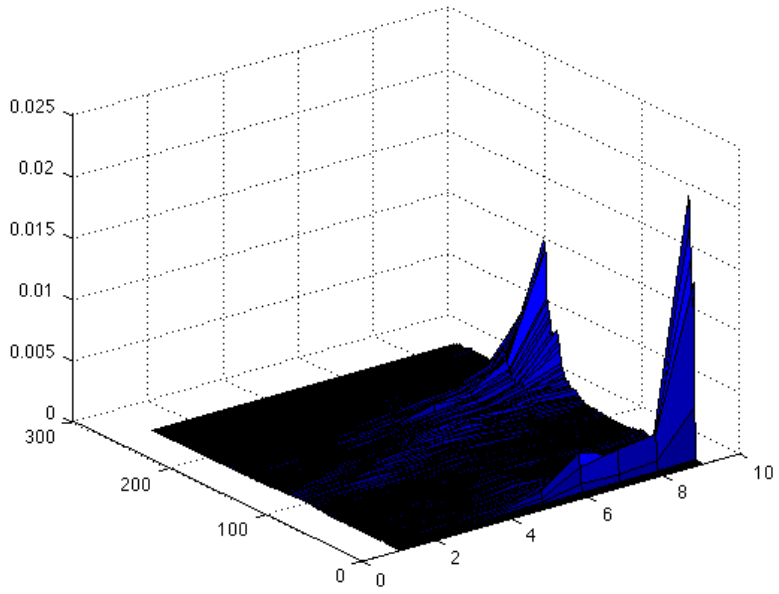


Figure 1.2: The GLSC histogram for the image in 1.1a

The entropy calculation uses for its distributions of background and foreground

$$C_0 : \frac{p(0,1)}{P_0(t)}, ..., \frac{p(0,N^2)}{P_0(t)}, ..., \frac{p(1,1)}{P_0(t)}, ..., \frac{p(t,N^2)}{P_0(t)} \tag{1.25}$$

and

$$C_1 : \frac{p(t+1,1)}{P_1(t)}, ..., \frac{p(t+1,N^2)}{P_1(t)}, ..., \frac{p(t+2,1)}{P_1(t)}, ..., \frac{p(L,N^2)}{P_1(t)} \tag{1.26}$$

where

So, using (1.24) and the distributions the entropy equations for the background and foreground respectively are

$$H_0(t) = -\sum_{k=0}^{t}\sum_{m=1}^{N^2}\left[\left(\frac{p(k,m)}{P_0(t)}\log\frac{p(k,m)}{P_0(t)}\right)W(k,m)\right] \tag{1.27}$$

and

$$H_1(t) = -\sum_{k=0}^{t}\sum_{m=1}^{N^2}\left[\left(\frac{p(k,m)}{P_1(t)}\log\frac{p(k,m)}{P_1(t)}\right)W(k,m)\right]. \tag{1.28}$$

As above described, the proposed threshold is obtained by maximizing $\phi(t) = H_0(t) + H_1(t)$ to get

$$T_{opt} = \arg\max_t \phi(t). \tag{1.29}$$

While this method used entropy in an interesting way, the novelty of this algorithm is the construction of the GLSC histogram and using it to calculate a more dynamic entropy that depends on local image data.

Another interesting algorithm that takes advantage of entropy methods is proposed in [2] to use the cross-entropy definition by Pal [30]. A gamma distribution is used in place of a Gaussian one because it works with both symmetric and non-symmetric histograms; it is used instead of K- or Beta-distribution because of its simplicity [2].

To make use of the additional distribution, the sets C_0 an C_1 are constructed as in (1.18) and (1.19) respectively. Next, the sets Q_0 and Q_1 represent the distributions based on the poisson model and are

$$Q_0(i) : \frac{e^{-\mu_0}\mu_0^i}{i!} \text{ for } i = 0, 1, ..., t \tag{1.30}$$

and

$$Q_1(i) : \frac{e^{-\mu_1}\mu_1^i}{i!} \text{ for } i = t+1, t+2, ..., L. \tag{1.31}$$

The cross-entropy for the image then becomes

$$\begin{aligned} D(t) = &\sum_{i=0}^{t} C_0(i)\log\left(\frac{C_0(i)}{Q_0(i)}\right) + \sum_{i=0}^{t} Q_0(i)\log\left(\frac{Q_0(i)}{P_0(i)}\right) \\ &+ \sum_{i=t+1}^{L} C_1(i)\log\left(\frac{C_1(i)}{Q_1(i)}\right) + \sum_{i=t+1}^{L} Q_1(i)\log\left(\frac{Q_1(i)}{P_1(i)}\right). \end{aligned} \tag{1.32}$$

However because μ_0 and μ_1 (1.30) and (1.31) are based on Gaussian methods, they are instead estimated respectively by

$$\mu_0 = \sqrt{\frac{\sum_{i=0}^{t} h(i)i^2q^2}{\omega_0}} \tag{1.33}$$

and

$$\mu_1 = \sqrt{\frac{\sum_{i=t+1}^{L} h(i) i^2 q^2}{\omega_1}} \tag{1.34}$$

where $q = \frac{\tau(N+0.5)}{\tau(N)\sqrt{N}}$ and $h(i)$ the histogram of image. The value q is computed with the standard gamma function using N as a shape parameter. This cross-entropy is minimized with respect to t to obtain the optimal threshold.

The method in [2] was developed in the context of processing data from Synthetic Aperture Radar (SAR) images. This type of data typically contains noise that is a consequence of the coherent processing of radar echoes and manifests itself in the image as a salt-and-pepper-like speckle. Segmenting detailed edges of objects is complicated by the presence of dispersed noise and so using a gamma distribution, the noise can be more easily smoothed.

Even though they are adapted from the field of communication, entropy-based approaches have provided new means of effective image thresholding using both global and local methods. The key to a strong and stable entropic approach is one that calculates over a probability distribution which accurately represents the relationship between pixel intensities and the class of a particular pixel. Even more localized approaches have been developed, but the use of a strictly local threshold (one that only thresholds a window of an image) can be destabilized by inconsistent illumination or shadow artifacts.

1.2.4 Fuzzy Thresholding

Though criterion functions and entropy provide a convenient and inexpensive way to calculate an image threshold, there are clearly disadvantages to such statistical dependence. Using principles of fuzzy logic and focusing on gray level similarity, algorithms have been developed to build a similarity model from which an effective threshold may be extracted. These methods can accommodate images whose histograms are multimodal and would cause other methods to err.

In the field of fuzzy logic, one assigns to each element x_i in the universe X a value between 0 and 1 given by a function $f_A(x_i)$; this is called the membership degree of x_i. A fuzzy set A in X is then defined as

$$A = \{(x_i, f_A(x_i)) | x_i \in X\}. \tag{1.35}$$

An S-function is used to model to the various membership degrees and to represent the classes of pixels [26]. The S-function is defined as:

$$f_{AS}(x) = S(x; a, b, c) = \begin{cases} 0 & x \leq a \\ 2\left(\frac{x-a}{c-a}\right)^2 & a \leq x \leq b \\ 1 - 2\left(\frac{x-a}{c-a}\right)^2 & b \leq x \leq c \\ 1 & x \geq c \end{cases} \tag{1.36}$$

where $b = \frac{a+c}{2}$ and is referred to as the "crossover point" because $f_{AS}(b) = 0.5$. The S-function can be controlled through the appropriate adjustment of parameters a and c. The S-function is sigmoidal in nature and will assign a higher value to pixels with a higher (lighter) intensity. This is similar to the binarization used by previous methods, except that some pixels are "maybe" in

one class or the other. To represent a function that assigns a higher membership degree to pixels of lower (darker) intensity we will use

$$f_{AZ}(x) = Z(x; a, b, c) = 1 - S(x; a, b, c). \tag{1.37}$$

One can see that by this criterion, a pixel is "fuzzier" if its membership degree is closer to 0.5. In [17], an index of fuzziness (IF) is introduced to measure a fuzzy set with a crisp set. A fuzzy A^* is crisp if $\forall x \in X$

$$\mu_{A^*}(x) = \begin{cases} 0 & \text{if } \mu_A(x) < 0.5 \\ 1 & \text{if } \mu_A(x) \geq 0.5 \end{cases} \tag{1.38}$$

and the IF is then calculated by a normalization of the distance between A and A^*

$$\psi_k(A) = 2n^{-\frac{1}{k}} \left[\sum_{i=1}^{n} |\mu_A(x_i) - \mu_{A^*}(x_i)|^k \right]^{1/k} \tag{1.39}$$

where n is the number of elements in A and A^* and $k \geq 1$. If $k = 1$ or $k = 2$, then the index is respectively called linear or quadratic. The result of this index is directly related to the ambiguity of the set - higher IF implies a higher ambiguity of the set data.

Once an IF function is established, the two fuzzy sets C_0 and C_1, as before, represent background and foreground levels respectively. One uses sets B and W defined to the levels modeled by the Z-function and S-function respectively. The region between these two sets where the Z-function and S-function are both 0 is called the *fuzzy region*. To determine the optimal threshold, one sequentially adds points to each class and assigns the point to the region for which the IF is lower. Because this method uses a measure of fuzziness, a normalization is required. The normalization factor is computed by

$$\alpha = \frac{\psi(W)}{\psi(B)}. \tag{1.40}$$

Once the normalization factor is obtained, one can assign each x_i to C_0 if

$$\psi(B \cup \{x_i\}) < \alpha\psi(W \cup \{x_i\})$$

and assign x_i to C_1 otherwise. With the continuation of this process, the threshold is obtained when

$$\psi(B \cup \{x_i\}) = \alpha\psi(W \cup \{x_i\}).$$

In [26], Lopes *et al.* remove some limitations of fuzzy thresholding by assigning a minimum number of pixels to the initial subsets B and W. Lopes *et al.* also use a similarity process to find the threshold point [26]. The method in [26] also uses a misclassification error to evaluate the new process.

1.2.5 Graph Cut Thresholding

A final talking point for our study of image thresholding is the idea of graph cuts; in this algorithm an image is modeled as a weighted graph where each node has fixed coordinates corresponding to its position in the image and a value corresponding to its gray level. Mathematically, this model is expressed as $G = (V, E)$ with V being the nodes in the graph and E representing the edges of the

graph. The strength of such a method is derived from data clustering techniques such as in [46]. The data model is used to calculate the cut between two classes of nodes (that form a partition of V) - our C_0 and C_1 for instance - and attempts to find the minimal cut between the two classes. This cut function is defined as

$$\text{cut}(C_0, C_1) = \sum_{u \in C_0, v \in C_1} w(u, v) \tag{1.41}$$

where $w(u, v)$ is the weight of the edge between node u and node v. However, only minimizing the cut between classes shows a bias toward smaller and more isolated clusters of data - something not typical in "natural" images. To correct for this, Shi and Malik in [40] proposed a *normalized cut* to measure group disassociation as

$$\text{Ncut}(C_0, C_1) = \frac{\text{cut}(C_0, C_1)}{\text{asso}(C_0, V)} + \frac{\text{cut}(C_0, C_1)}{\text{asso}(C_1, V)}. \tag{1.42}$$

where

$$\text{asso}(C_0, V) = \sum_{u \in C_0, v \in V} w(u, v) \tag{1.43}$$

is the total edge weights between C_0 and the rest of the graph. The major contribution in the method proposed by [42] is that the Ncut becomes

$$\text{Ncut}(C_0, C_1) = \frac{\text{cut}(C_0, C_1)}{\text{asso}(C_0, C_0) + \text{cut}(C_0, C_1)} + \frac{\text{cut}(C_0, C_1)}{\text{asso}(C_1, C_1) + \text{cut}(C_0, C_1)} \tag{1.44}$$

which - as we will see in the algorithm's implementation in chapter 2 - can be made substantially more efficient because the Ncut calculation depends not on the size of the image, but the number of possible pixel intensities. One can see that the normalized cut in (1.42) does not bias smaller node clusters because $\text{asso}(C_0, V)$ will always grow with more nodes in C_0; then when one minimizes the result of $\text{Ncut}(C_0, C_1)$ the tendency will be to find a balance between strongly weighted sets and larger ones. The weight function proposed in [42] is

$$w(u, v) = \begin{cases} e^{-\left[\frac{||\text{F}(u) - \text{F}(v)||_2^2}{d_I} + \frac{||\text{X}(u) - \text{X}(v)||_2^2}{d_X}\right]} & \text{if } ||\text{X}(u) - \text{X}(v)||_2 < r \\ 0 & \text{otherwise} \end{cases} \tag{1.45}$$

where $\text{F}(x)$ is the intensity at node x and $\text{X}(x)$ is the location of node x. The parameters d_I and d_X in (1.45) are used to control how much each property contributes to the weight; r is a predefined constant to control how close two nodes must be to have a non-zero weight. Also, in (1.45) $||...||_2$ denotes the Euclidean vector norm. Looking at (1.45), we can see that the smaller the value of r, the more sparse the representation graph will become. Tao *et al* manage to control the computational cost using their derivation of the normalized cut so that it does no grow exponentially like a typical attempt to compute all the weights of a dense graph. They reduce their complexity from that of normal solutions $O(\text{N}^3)$ to $O(\text{N})$ where N is the number of pixels in the image.

Newer methods such as in [9] are used to modify the method of normalized cut for an attempt at a more accurate segmentation. In [9], the weight function is modified to take into account visible edges in the original image and if an edge lies on one of these lines, it will have a greater weight and therefore resist separation. The method in [9] also uses a component tree as an alternative representation of the image to which the normalized cut would be applied.

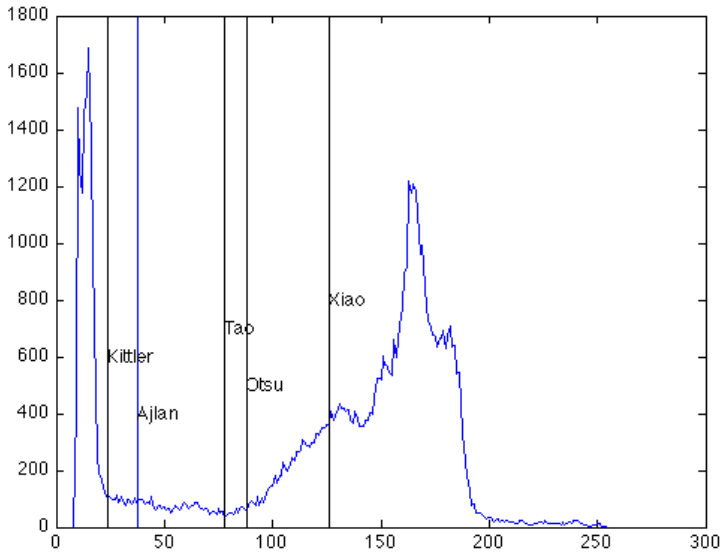


Figure 1.3: Figure 1.1a thresholds calculated by the methods in [20], [2], [42], [29], [47]

1.3 Feature Extraction

Every image is made up of a distinct set of features that can be used in classifying the image in relation to similar images. These features can represent any number of items such as, but not limited to shape, color, edges, and blobs [25]. Each of these features is represented by a set of numerical values in the form of a feature vector. A feature vector is used to help reduce the large amount of information present in the original image that would normally take an extremely long time to classify.

Once the feature vector has been extracted, it can be further reduced through certain techniques such as Principal Component Analysis (PCA) [36], or Latent Semantic Analysis [49]. Dimensionality reduction algorithms are done in the hope of finding the fewest number of features with the highest amount of variance to improve the chances of classification.

1.3.1 Gabor Filters

The Gabor filter [43] is used in several image processing applications. In our application, the output from the filter allows us to locate any important edges in an image which we can then use in identification. The filter is useful for representing texture in part because all Gabor filters are derived from the same basic wavelet, and are changed through rotation and scaling. The basic Gabor filter equation is as follows:

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x\prime^2 + \gamma^2 y\prime^2}{2\sigma^2}\right) \exp\left(i\left(2\pi\frac{x\prime}{\lambda} + \psi\right)\right) \tag{1.46}$$

The λ in this equation represents the wavelength of the sinusoid, θ is the orientation of the normal, ψ represents the phase offset, σ is the sigma of the Gaussian envelope, and γ is the spatial aspect ratio. x$\prime$ and y$\prime$ are calculated using the following two equations:

$$x\prime = x\cos(\theta) + y\sin(\theta) \tag{1.47}$$

$$y\prime = -x\cos(\theta) + y\sin(\theta) \tag{1.48}$$

Due to the relationship between the Gabor filter and Gabor wavelets, it is beneficial to run an image through several Gabor filters that have different orientations and sizes. When using the different Gabor filters on each image, the relationship between these outputs is unique and can be very useful in identifying specific features. An example of this is looking for an area of the image that has a distinctly different color from the rest of the image, and is unique to a specific class.

1.3.2 Fourier Descriptors

Fourier descriptors are used to describe the shape of an object with regards to its frequency. Descriptors in lower frequencies represent general information about the shape while descriptors at higher frequencies correspond to finer details regarding the shape [50]. In addition to the different frequencies, the Fourier descriptors have a zero frequency descriptor that is known as the DC component. This component is unique in that it does not describe the shape of the object, but rather the location.[16] The DC can be extremely useful when trying to normalize an image because it can help center the image on the origin. Because of the difference in information depending on the frequency, Fourier descriptors can easily be reduced in dimensionality [50]. A common way to do dimensionality reduction is through the energy of the Fourier descriptors. The goal is to capture the largest amount of energy in the fewest number of points [16]. The equation for the energy of a set of descriptors is as follows:

$$E(M) = \sum_{m=-\frac{M}{2}}^{\frac{M}{2}-1} \sum_{m\neq 0} |Z_m|^2 \tag{1.49}$$

As shown, this equation takes the sum of the magnitude squared for each point in the set of Fourier descriptors. The DC component is left out of this calculation due to its lack of information regarding the shape of the specified object. In order to calculate the proper dimensionality reduction the total energy for all Fourier descriptors is calculated, and then the energy of smaller subsets of Fourier descriptors is calculated. A ratio of the smaller subset's energy to the full set's energy is calculated, and the ratio between this subset of points and the energy of the entire set of points is used to determine the dimensionality reduction. [16]If the energy of 64 points is .933 of the total energy, while 128 points has .941 of the total energy, it would be more beneficial to use 64 points because using 128 points does not provide enough of an improvement to merit the increase in computational resources.

1.4 Classification

Classification is the process of taking features extracted from the raw video data and using them to determine the species of fish present in the data. Neural networks are powerful classifiers and

several architectures were explored in this research.

Restricted Boltzmann machines and the deep belief networks that can be composed of them are interesting for their ability to learn models of input data through unsupervised learning. In particular, they may be useful for reducing noise and recovering missing data in extracted features.

Convolutional neural networks are extremely well suited to processing visual information and a natural fit for classifying image data. They are also robust to noise and small distortions.

1.4.1 Restricted Boltzmann Machine

Boltzmann machines are different from traditional neural networks. Rather than learning a function to separate classes of data, Boltzmann machines are trained to model a data distribution. A Boltzmann machine, when trained on a certain distribution of data, when presented with random input and let run to equilibrium will produce output within that distribution. The same machine when presented with data slightly different from that which was trained and then run only briefly, will produce data more similar to the training data.

Boltzmann machines are highly connected, parallel networks consisting of two fully connected layers of binary neurons, one "visible" layer, and one "hidden" layer[1]. In a restricted Boltzmann machine (RBM), each neuron is connected to every neuron in the other layer, but the intra-layer connections are eliminated.

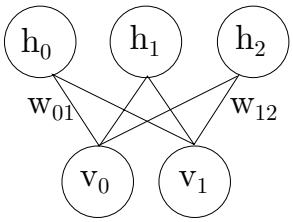


Figure 1.4: RBM Architecture

The network encodes an energy function,

$$E = -\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i \tag{1.50}$$

where w_{ij} is the symmetric weight between units i and j, s_i is the binary state of neuron i, and θ_i is its threshold.

Neuron k is "on," that is $s_k = 1$, with probability p, where

$$p = \frac{1}{1 + \exp\left(-\sum_i w_{ik} s_i\right)} \tag{1.51}$$

We can update the state of one layer by calculating its activation probabilities based on the state of the other layer and then sampling. By performing repeated iterations of this Gibb's sampling one can find the minima of this energy. This stochastic process makes the Boltzmann machine more resilient to local minima than its deterministic counterpart the Hopfield network.

RBMs can be efficiently trained by minimizing the contrastive divergence of the network[12]. Gibbs sampling is used to update the states of the network, and by comparing the initial and

equilibrium derivatives the network parameters can be adjusted. Fortunately, in practice good approximations can be achieved by performing only a small number of iterations of Gibbs sampling, rather than finding the equilibrium state, and even a single iteration yields good results.

1.4.2 Deep Belief Network

Deep belief networks are another deep learning architecture. They consist of multiple layers with directed connections, and a final layer with symmetric connections forming an associative memory. The most common form of deep belief network (DBN) is a stack of multiple RBMs. The first layer learns a representation of the inputs and its weights encode features of the input space, and subsequent layers learn representations of these representations, encoding features of the features. In this way, DBNs can learn better representations of data than a simple RBM can.

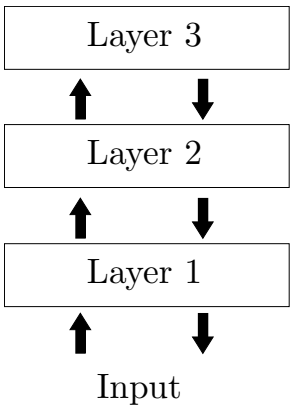


Figure 1.5: DBN Architecture

Such networks can be efficiently trained in a greedy fashion[13]. The lowest layer is trained using the RBM training algorithm, and then the next layer is trained, using the output (hidden units) of the first layer as its inputs. This process can be repeated for subsequent layers.

DBNs can be extended to directly discriminate by learning labels associated with inputs. During training, the output of the penultimate layer is concatenated with the desired label (which is clamped) when training the associative memory. To then infer a label from unknown data, labels can be initialized to 0.1 and Gibbs sampling can be performed, or the free energy of the vectors possible by considering label possibilities can be calculated directly, yielding more accurate results. Network performance can be improved even further by following the greedy training with a more refined "up-down" training pass.

1.4.3 Convolutional Neural Network

Convolutional neural networks (CNNs) are particularly well suited to classifying image data, outperforming many other architectures at recognizing handwritten digits [41]. They are similar to traditional feed-forward networks, however they use local connections and weight sharing to create an efficient, highly connected architecture which makes use of spatial information. They are made up of alternating convolution and down sampling layers, followed by standard MLP layers [22].

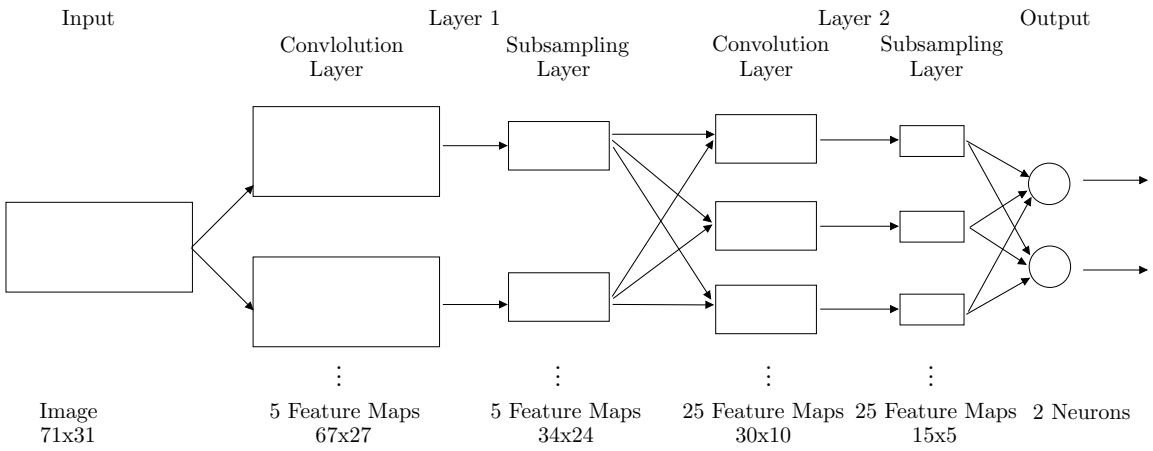


Figure 1.6: CNN Architecture

Each convolutional layer consists of multiple feature maps, and each node in the feature map is connected to a small region around the corresponding nodes in the previous feature maps. These weight kernels are shared across that feature map, so updating the map is merely a matter of convolving the kernels with the input feature maps. The convolutional layer is followed by a down sampling layer, usually using either the local maximum, or simply sub sampling.

In this way, early layers in the network extract general image features, and later layers extract more specific and higher level features. The last layers of the network are standard fully-connected MLP layers and the network output is the classification.

Chapter 2

Application in Fish Detection

To integrate the components of the project, the thresholded results were used in the feature extraction and classification algorithms independently (Fig. 2.1).

Figure 2.1: The project data flow

2.1 Thresholding

In practice, one would not threshold a raw data image such as 1.1a or 2.2a; in our experiments, we incorporated the thresholding methods in one of two ways: 1) threshold the difference of a single frame and the background (as in 2.2c) or 2) threshold the difference of a single frame and background as before, but then threshold the difference image in a band around the detected edges of the binary image. By sampling only the band of the original, the threshold would be able to more easily distinguish object differences and in general provide a sharper and more accurate segmentation.

The process of fish segmentation is detailed in algorithm 1. In Algorithm 1 perhaps the most important step is to apply the thresholding algorithm. Originally, the threshold was manually chosen and hard-coded into the project. For this reason, we chose to investigate and implement existing thresholding methods that could handle images with varying illumination, incomplete foreground data, etc. Of the methods mentioned in section 1.2, the methods of Otsu [29], Ramesh [35], Kittler [20], Guo [11], Xiao [47], Ajlan [2], and Tao [42] were implemented. Time was the largest restriction on how the breadth of our algorithmic testing and so those listed were chosen as judiciously as possible - taking into account complexity, efficiency, and potential benefit. For the most part, each

implemented thresholding algorithm was written as described by their respective authors. The specific code for these implementations may be found in appendix C.

Our implementation of Tao *et al*'s method did have a computational improvement over what was proposed in [42]. IN [42], the authors claim that the operation of computing the matrix M[1] was the most expensive because of the many exponents and multiplications needed. Recall that in 1.2.5, the weight between two nodes was

$$w(u,v) = \begin{cases} e^{-\left[\frac{||\mathrm{F}(u)-\mathrm{F}(v)||_2^2}{d_I} + \frac{||\mathrm{X}(u)-\mathrm{X}(v)||_2^2}{d_X}\right]} & \text{if } ||\mathrm{X}(u)-\mathrm{X}(v)||_2 < r \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

We can expand this to be

$$w(u,v) = \begin{cases} e^{-\frac{||\mathrm{F}(u)-\mathrm{F}(v)||_2^2}{d_I}} e^{-\frac{||\mathrm{X}(u)-\mathrm{X}(v)||_2^2}{d_X}} & \text{if } ||\mathrm{X}(u)-\mathrm{X}(v)||_2 < r \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

and taking $||\mathrm{X}(u)-\mathrm{X}(v)||_2 < r$, we can form

$$w_I(u,v) = e^{-\frac{||\mathrm{F}(u)-\mathrm{F}(v)||_2^2}{d_I}} \tag{2.3}$$

and

$$w_X(u,v) = \begin{cases} e^{-\frac{||\mathrm{X}(u)-\mathrm{X}(v)||_2^2}{d_X}} & \text{for } ||\mathrm{X}(u)-\mathrm{X}(v)||_2 < r \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

However, as can be seen in the code in Appendix C for the Tao function, we store only sum of the weight component s in the M matrix. To see the advantage, consider any

$$\begin{aligned} \mathrm{M}_{i,j} &= \sum_{u\in V_i, v\in V_j} w(u,v) \\ &= \sum_{u\in V_i, v\in V_j} w_X(u,v) w_I(u,v) \\ &= \sum_{u\in V_i, v\in V_j} w_X(u,v) e^{-\frac{||i-j||_2^2}{d_I}} \\ &= e^{-\frac{||i-j||_2^2}{d_I}} \sum_{u\in V_i, v\in V_j} w_X(u,v) \end{aligned} \tag{2.5}$$

So in our implementation, once the entire M matrix has been constructed, we simply multiply each $\mathrm{M}_{i,j}$ by $e^{-\frac{||i-j||_2^2}{d_I}}$ since the intensities in M are given by the indexes i and j independent of where the pixels are in the image. The Ncut function can be calculated exactly according to [42].

By extracting this exponential process, the computation time is reduced because one only needs one additional multiplication per non-zero element in the M matrix (assuming that $e^{-\frac{||i-j||_2^2}{d_I}}$ is

[1]Recall that M is the matrix in section 1.2.5 where $\mathrm{M}_{i,j} = \mathrm{cut}(V_i, V_j)$

already computed at the time of multiplication). For that purpose, the values of $e^{-\frac{||i-j||_2^2}{d_I}}$ are stored in a lookup table. Also stored in a lookup table is the distance between any pixel in a $(2r+1) \times (2r+1)$ window with the center of the window. This lookup table is used as a mask to compute w_X because it is independent of the location of the nodes in the actual graph; only their relative distance affects the weight calculation. Taking advantage of this, we do not need to consider v nodes outside of a $(2r+1) \times (2r+1)$ window centered at a particular u node.

Another approach we took to improve the accuracy of the graph cut algorithm was to the sum of M matrices for a number of frames before, after, and including the frame currently being processed. This would hopefully adjust the result for inconsistencies such as illumination and shadow. Regions that might be unclear in one frame could be sharpened by this method because the fish might soon move to a region of higher contrast (or was recently in such an area).

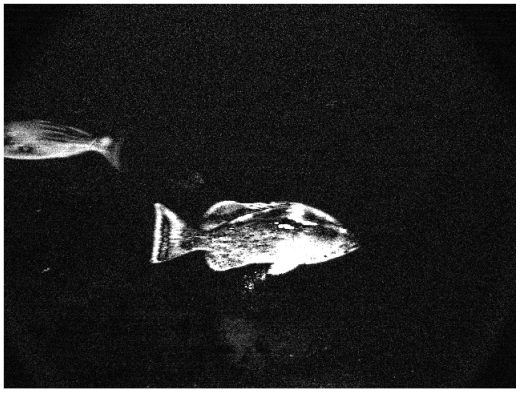
Another novel technique we employed (referred to as method (2) in Algorithm 1) in the overall structure of the project was an edge-thinning procedure intended to remove a "halo" that was getting included with some fish segmentations. Using only a band around detected blobs allowed us to use a less fuzzy filter to preprocess the data before imaging; this would improve the precision of the threshold. Because we repeat the thinning process with smaller bands, less filtering is required to removed noise that would bias the threshold. This method was run over the same data set as the original method and the results will be discussed in 3.1.

Algorithm 1 Algorithm for Segmentation

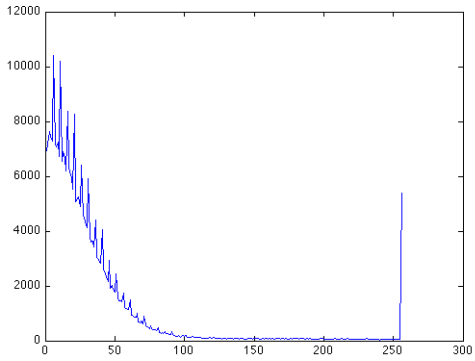
```
Down-sample images (for speed)
Calculate background
for all images in the set do
  Subtract background from current image
  Filter subtraction results to remove noise speckle
  Threshold result image and retrieve binary image
  if using method (2) then
    for all band sizes: x pixels above, below, left, and right do
      Detect edges of binary image
      Read original image in region within x pixels of edges.
      Apply a less blurry filter to the edge band to improve segmentation precision
      Threshold only the band region
      Replace into binary image the result of edge band thresholding
    end for
  end if
  for all separate objects in binary image do
    Extract objects in fixed-size bounding box and replace with values from original image
    Write object extractions to file
  end for
  Write original image (segmented according to binary) to file
  Write edge image of binary image to file
end for
Pass individual object extractions to feature extraction and classification network
```

(a) The raw frame

(b) The calculated background



(c) The difference image between (a) and (b)



(d) The histogram of 2.2c

Figure 2.2: An Original Image and Calculations

2.2 Image Segmentation

Once the image has been segmented according to the thresholding results, the objects assumed to be fish are submitted for feature extraction and classification . Because our images are of "natural", they do not have consistent illumination so the background substitution is not perfect. Some fish objects may have holes, or part of the fish near the edge may appear to be cut out. This can also be caused by obscure angles of fish exposure to the camera. To retrieve the most data possible, we dilate a copy of the original image to correct these discrepancies. The first step is to start with the largest structuring element to fill the biggest holes. We then use the dilated image to compare with the original image. We do not dilate the original image because a large dilation creates an unnecessary 'halo' of background around the fish. By dilating with smaller structure elements, we fill in as much of the fish as possible without enlarging this halo.

After dilation, we extract the fish data from the original image by identifying every object that is at least 50×50 pixels. We eliminate small blobs present due to deficiencies with the background subtraction. For each of these blobs a separate 281×121 image is created. The center of the blob is then mapped to the center of this new image, and the rest of the image is then filled in with pixels from its corresponding bounding box in the original frame. If the bounding box is smaller than 281×121 the rest of the new image is filled by black pixels. The segmentation is saved as well as the original image with the background subtracted.

2.3 Feature Extraction

As mentioned previously, after an image is preprocessed we then use a variety of methods to extract features that can then be used for classification. We looked at two different processes to extract unique features for classification. The first method we used was the WARP: Accurate Retrieval based on Phase (WARP) process. WARP uses Fourier descriptors to represent the shape of the object, and normalizes these descriptors with the phase and magnitude of them.

Compared to common Fourier descriptor methods, WARP is unique in two respects. The first is that WARP retains the phase information which allows for classification despite the rotation of objects in the image, and instead of using a standard distance measure such as Euclidean distance, Dynamic Time Warping (DTW) is used instead for classification.

As mentioned previously, a unique feature of WARP is that it retains and uses the phase information to help with classification. In most algorithms involving Fourier descriptors, the phase information is discarded because an invariance of rotation for all images is required [16]. (7,8) In addition any potential rotation using this phase information will shift the descriptor coordinates, which will add a significant amount of error to standard descriptor classification algorithms. The phase information could cause errors by using different starting points on the image shape. The problem that arises when discarding the phase information is the similarities that can exist between two completely different objects [16]. By retaining this phase information we can an additional parameter to classify these images by. Evidence of this is shown in Figure 2.3. The magnitude between Fig. 2.3a and Fig. 2.3b is much more similar than that of Fig. 2.3a and Fig. 2.3c despite the fact that Fig. 2.3a and Fig. 2.3c are of the same fish species. Looking at the phase plot however, Fig. 2.3a and Fig. 2.3c are much more similar to each other than Fig. 2.3a and Fig. 2.3b are. In order to preserve the phase information without causing any problems that would normally result in discarding the phase, the phase is modified with equation 2.3 to retain rotation and starting point invariances.

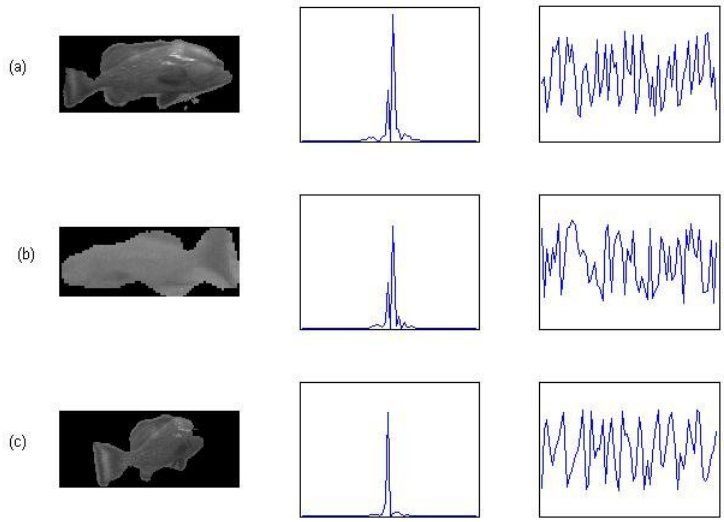


Figure 2.3: Fish images with phase and magnitude plots. A and C are same fish

The purpose for using Dynamic Time Warping [44] in these experiments is to allow for small discrepancies in objects that are the same, but are not properly aligned along the time axis. In classic Fourier descriptor comparisons, simple distance measures such as Euclidean distance are used, however this would not be ideal if any phase shifting occurs. DTW has been shown to be a superior method of classification with respect to time [19] , and has also been shown to improve on shape descriptor classification as well [16]. Although there are a variety of methods to extract shape descriptors, one of the most important characteristics is to retain as much shape information as possible in the fewest number of points. For this reason Discrete Fourier Transform (DFT) based points were chosen as being the most effective with respect to the frequency domain. [18] The DFT equation is as follows [16]:

$$Z_m = \sum_{l=0}^{N-1} z_l e^{-j\frac{2\pi lm}{N}} = R_m e^{j\Theta_m} \tag{2.6}$$

In this equation R_m represents the module or magnitude of the m$^{\text{th}}$ DFT coefficients, and Θ_m represents the phase of the DFT coefficients.

After all of the DFT coefficients are calculated, some form of dimensionality reduction must be performed. The initial set of DFT coefficients will be extremely large as each DFT coefficients will correspond to a pixel on the edge of the object that is trying to be classified. In most cases this results in a minimum of 400 coefficients, although it will vary depending on the size of the object in the image as well as the image's resolution. In order to do this we calculate the spectral energy of certain subsets of coefficients in the manner described in Section 1.4.2. This results in reducing

the number of coefficients to at least one quarter of the original number, if not more. Because the coefficients are ordered based on their frequency, only coefficients with a low frequency are used, but in almost all cases this is beneficial because the coefficients are more descriptive at these lower frequencies [33].

As mentioned previously, the changes in position, rotation, scale, orientation, and initial point on the boundary all modify the DFT coefficients. Because of this change there is a specific way of modifying the DFT coefficients to guarantee invariance among the scaling, rotation, translation, and starting point. The first step is to take the DC coefficients and set it equal to zero. As we mention multiple times, the value of the DC coefficients is of no consequence to the shape, and only matters for the location. By setting the DC to 0 we ensure that every image is translated to the origin. The next step is to modify the magnitude invariance amongst the scale. This is done with the following equation [16]:

$$\hat{R_m} = \frac{R_m}{R_1} \tag{2.7}$$

To modify the magnitude, the m^{th} magnitude is divided by the magnitude that is one place to the right of the DC coefficients.

In order to appropriately account for invariance in the phase and starting point, two different equations that both modify the phase value are used. We show them combined below because calculating both separately or together results in the same final phase value. The equation is as follows[16]:

$$\hat{\Theta_m} = \Theta_m - \frac{\Theta_{-1} + \Theta_1}{2} + m\frac{\Theta_{-1} - \Theta_1}{2} \tag{2.8}$$

In the first part of this equation we are modifying the original phase by subtracting the average between the phase values of the two coefficients on either side of the DC. This is the part of the equation that accounts for rotation invariance. After this the average of the difference between these same two phase values is multiplied by the m^{th} element we are modifying the phase of. For example if we are trying to modify the 5th element of the phase, then m would be equal to 5 so the average difference between the two phase values on either side of the DC would be multiplied by 5. This number is subtracted from the original phase instead of being added to the phase like the rotation part of the equation is. This second part ensures the invariance of the starting point for each set of coefficients.

Once the DC element has been modified, and the phase and magnitude values have also been appropriately changed, the normalized DFT coordinates must be calculated. This is done with the following equation [16]:

$$\hat{Z_m} = \hat{R_m} e^{j\hat{\Theta_m}} \tag{2.9}$$

Of note here is that the normalized DFT coefficients do not use the original DFT coefficients directly in their calculation. Instead they just used the modified phase and magnitude values. It is also important to remember that $\hat{Z_0}$ must be equal to 0 to ensure the translation is done properly, and it is also wise to make sure that the corresponding phase and magnitude for the DC coefficients are 0 as well to prevent any error being introduced.

After the normalized DFT coefficients have been calculated, it is necessary to perform an Inverse DFT on these coefficients. The Inverse DFT uses the following equation to come up with a

normalized signal. This signal will have all of the appropriate invariances and can then be used in classification through Dynamic Time Warping [16].

$$\hat{z_l} = \frac{1}{M} \sum_{m=-\frac{M}{2}}^{\frac{M}{2}-1} \hat{Z_m} e^{j\frac{2\pi lm}{M}} \qquad l = 0, ..., M-1. \tag{2.10}$$

Once two sets of normalized signals $\hat{z}$ and $\hat{\hat{z}}$ exist for comparison, the most common way to compare these signals is through Euclidean distance. The drawback to using Euclidean distance is that it does not take into account shifting along the time axis. This shifting is common when two similar signals exist, but are not completely aligned with each other. To counteract this issue, the WARP process uses Dynamic Time Warping to measure distance. DTW is a recursive function at its basic level, and then entire equation is as follows [3]:

$$\begin{aligned} d_{tw}(\langle\rangle, \langle\rangle) &= 0 \\ d_{tw}(\hat{z}, \langle\rangle) &= \infty \\ d_{tw}(\langle\rangle, \hat{z}) &= \infty \end{aligned}$$

$$d_{tw}(\hat{z}, \hat{\hat{z}}) = \sqrt{\varrho(\hat{z_0}, \hat{\hat{z_0}}) + min \begin{cases} d_tw(\hat{z}, tail(\hat{\hat{z}})) \\ d_tw(tail(\hat{\hat{z}}), \hat{z}) \\ d_tw(tail(\hat{z}), tail(\hat{\hat{z}})) \end{cases}}$$

In DTW, the part of the equation $\varrho(\hat{z_0}, \hat{\hat{z_0}})$ references any distance function that can be used to compute the distance between the two specified points. The most common form of distance will be the Euclidean distance, which will not introduce error into the DTW because the recursive nature of this function is designed to eliminate the shifting problems that accompany standalone Euclidean distances. For the WARP process however, we use a squared Euclidean distance that is represented by the function [16]:

$$\varrho(\hat{z_i}, \hat{\hat{z_j}}) = |\hat{z_i} - \hat{\hat{z_j}}|^2 \tag{2.11}$$

Due to the recursive nature of DTW, it can take an extremely long time to calculate the distance measure between two sets of DFT coefficients even after dimensionality reduction. Some simple experimentation indicated that two vectors with a length of 9 took 24.5 seconds to compute the distance, and two vectors of length 8 took 20.2 seconds for computation. This itself is not awful, but when increasing the vector size to 10 the calculations took 8 minutes 32 seconds. Obviously it is possible the randomly generated numbers for the vectors caused this kind of increase in runtime, but the mere possibility such an increase exists indicates that some method is needed to reduce the number of calculations needed to compute DTW distance. Because of this it is very common to limit the DTW calculations so that they can not deviate from the diagonal beyond a certain constraint. One of the most common constraints is the Sakoe-Chiba Band.

The Sakoe-Chiba band is used to limit the warping path from deviating beyond a specified length from the diagonal. The biggest benefit of this is that it reduces computational time from $O(M^2)$ to $O(M\omega)$ where ω is the gradient [44]. The main cause of this speedup is that it prevents distance measures being taken from points that are to far away from each other. Because DTW distance is predicated upon the two shapes being similar, but slightly different in the time series, it

stands to reason that it is pointless to calculate the distance between two points with completely opposite frequencies. If you go bck to the Inverse DFT coefficients and their frequencies, it stands to reason that a coefficient at frequency M-1 is not going to have any relationship with a distance at frequency $\frac{-M}{2}$. The Sakoe-Chiba band eliminates the calculation of this distance during its execution of DTW.

The important step in Sakoe-Chiba band calculations is to determine the value of ω which is known as the gradient [37]. The gradient is calculated through an i by j matrix, where the all i elements correspond to one vector, and all j elements correspond to a second vector. This leads to the gradient being defined as follows [37]:

$$|i(K) - j(K)| \leq r \tag{2.12}$$

By running this equation until K reaches the length of i or j, we can find the max value of r which will constrain the warping path while ensuring that needless calculations are not made. This number may need to be adjusted slightly if the gradient is to steep or shallow, but this can be done through simple testing. For our experiments we found that the ideal constraint for the Sakoe-Chiba band was 2.

The second process used for feature extraction and classification is based upon using some known physical features of the fish images in our database. The E.morio fish has a distinct pattern at the edge of the tail that is much lighter in color than the rest of the fish. This pattern is small in width and runs for most of the tail. We used Gabor filters to highlight this line on the tail, and classify the images based on the existence of the line or not.

The first step was to take a Gabor filter that was oriented horizontally with the theta angle at 0 radians. This filter was run over the entire image to detect any horizontal edge components. The next step was to take a Gabor filter oriented at $\frac{\pi}{2}$ radians and made a pass over the entire original input image. The output images from each Gabor filter pass are then combined in a ratio as shown in Figure (). This image shows a distinct amount of texture related to the differing pixel values after each Gabor filter passes over the image. This makes it possible to identify distinct features that can be used in classification of the species. The next step is to automatically look for lighter parts of the image to locate the appropriate stripe on the back of the tail. If this stripe exists the image is E. morio , and if the stripe does not exist it is considered to be some other species.

2.4 Classification

2.4.1 Restricted Boltzmann Machine

An RBM consists of two layers, where each neuron is fully connected to the opposite layer by symmetrically weighted connections, as well as a bias unit. A neuron is updated by calculating the weighted sum of that neuron's inputs and bias,

$$x_j = \sum_i w_{ij} s_i + b_j \tag{2.13}$$

and then find the activation of the neuron using the logistic sigmoid function, which is the neuron's probability of being "on."

$$p_j = \frac{1}{1 + \exp(-x_j)} \tag{2.14}$$

A uniformly random value is taken from the range $(0, 1)$ and compared with the activation value to determine the state of the neuron. This calculation depends only on the neurons in the opposite layer, so a layer can be updated entirely in parallel.

The network is trained by setting the visible layer to an input pattern, let this value be v, sampling the hidden layer to get h, and then sampling again to calculate $\hat{v}$ and $\hat{h}$. We can then calculate the required weight change

$$\Delta w_{ij} = \langle v_i h_j \rangle - \langle \hat{v}_i \hat{v}_j \rangle \tag{2.15}$$

where $\langle \cdot \rangle$ denotes an average over the training data. Biases are updated similarly.

$$\Delta b_j = \langle s_j - \hat{s}_j \rangle \tag{2.16}$$

When training, data is usually split into smaller minibatches of size 10–100, with the different classes of data evenly represented in each minibatch, if feasible, or else randomly distributed among minibatches. For each minibatch, all samples are presented and then the weights are updated. An epoch is the presentation of all minibatches.

Implementation and training parameters were influenced by the University of Toronto machine learning group's excellent guide to training RBMs [**?**].

2.4.2 DBN

The DBN is implemented as a layered network, where each layer is an RBM. To infer from a sample, the lowest layer's visible units are set to the input, its hidden units are sampled, and then the visible units of the next layer are set to these hidden units. The final associative layer can perform several iterations of Gibbs sampling, and then to generate an output, each RBM in turn gets its hidden values from the layer above and then samples its visible units.

To train the DBN, each RBM is trained in turn, bottom up, and higher layers use the hidden values of lower layers as their input. This yields a network with good generative properties, and can serve as a starting point for further refining the network for discrimination [13].

2.4.3 CNN

The CNN is similar to standard feed forward networks, however it introduces a new layer architecture called a convolution layer, which is made up of several feature maps. Each node in a feature map is connected to all the nodes in a 5x5 region at the same location in the previous layer. This 5x5 weight matrix is shared among all neurons in the feature map. Thus, the forward pass of the convolution layer can be efficiently implemented by a convolution over the input.

$$Y_j = \text{conv}(X_i, k_{ij}) + b_{ij} \tag{2.17}$$

where conv is a convolution over the valid region of the input, X_i is is the input feature map or image, k_{ij} is the weight kernel for that feature map, and b_{ij} is the bias.

The downsampling layers merely reduce the size of each feature map in the convolution layer. Different schemes are possible, including choosing the maximum or average value, but for simplicity downsampling is often used, and that was the method chosen here. The subsampling layer also activates the feature map outputs.

One or several standard fully connected layers are used as the last layers of the network. Standard multilayer perceptron (MLP) equations are used.

$$Y_j = \sum_i x_i w_{ij} + b_j \tag{2.18}$$

The network is trained through standard backpropogation and gradient descent. Errors backpropogate through the fully connected layers as expected.

Backpropogation through the subsampling layers is potentially more complicated. For example, Simard et al. recommend "pushing" rather than "pulling" sensitivity information [41]. However we solve this using another convolution, as described in Jason Bouvries excellent "Notes on Convolutional Neural Networks"[**?**].

$$S_i = \text{conv}(S_j, \text{rot180}(k_{ij})) \tag{2.19}$$

where here we take the full convolution, assuming zero padding outside the image bounds and use a rotated version of the weight matrix.

Calculating convolution layer sensitivites is easily accomplished by merely upsampling the subsampling layer sensitivites.

Chapter 3

Experimentation

3.1 Thresholding

A difficulty of choosing a thresholding method is the lack of a strict "goodness" criterion. Of course, a fast algorithm is desirable but often, the more expensive methods yield more accurate results. The results in Figure 3.1 show the time differences for each algorithm operating on an image 480×640 pixels. The quality of the segmentations may only be qualitatively evaluated from the algorithm output. Some sample results of the algorithms are comparatively shown in Appendix A.

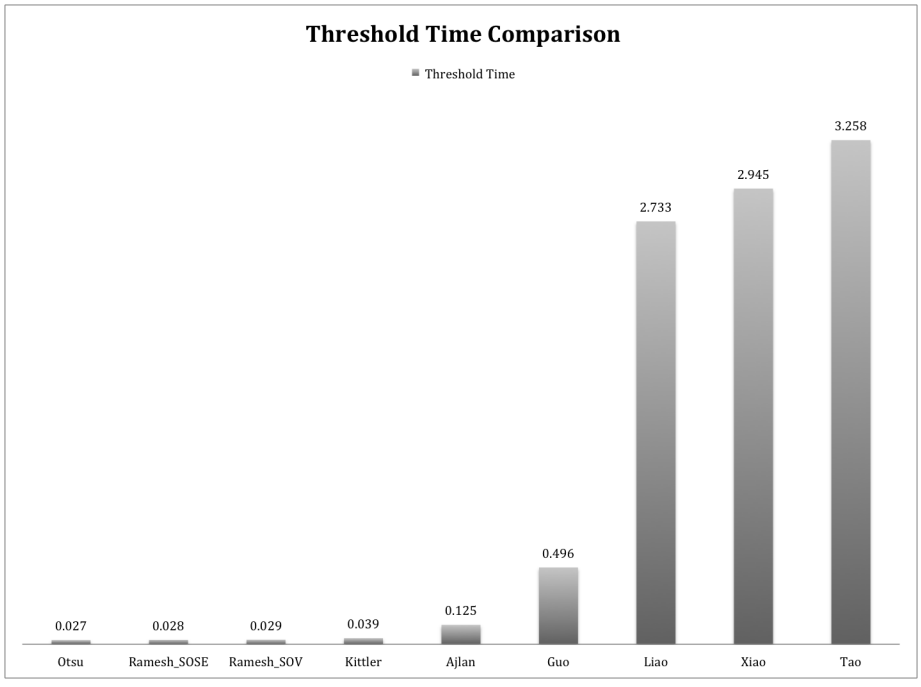


Figure 3.1: Algorithm Execution Time to Threshold a 480 × 640 Pixel Image

The data in Figure 3.1 also demonstrate that statistical methods which focus on the histogram are orders of magnitude faster than methods which involve individual pixel intensity. Although Liao's method had a (comparatively) large computational time, this is because it was actually a multi-level threshold that grouped both of the top two layers into the foreground.

In Appendix is a comparison between thresholding results with and without the edge thinning technique. While not all algorithms benefitted from this additional processing, it improved the accuracy of several algorithms. The algorithm by Guo *et al.* shows significant accuracy improvement in Figure B.3 and Figure B.4. The segmentations in these data adhere much more strongly to the actual fish region with the extra edge-thinning process. The results from Liao *et al.*'s and Kittler *et al.*'s algorithm also show slight (but less significant) improvement from the edge thinning process. In some cases, the edge thinning caused a collapse of the fish region and was largely corrupted. This can be seen in the results from Ramesh *et al.*'s sum of variance method. In other cases such as the results from the graph cut algorithm, a poorly detected region was expanded by the edge thinning to achieve a more accurate result that was larger than before!

Overall, when the data was qualitatively selected by hand to be used in the classification system, the addition of the edge thinning results improved the reliability of the data. This improved data could then be used with WARP and the classification networks to identify the fish in the video.

3.2 WARP

Before running the WARP process on our database of fish images, we first conducted a few simple tests to ensure that WARP performs as advertised on images that are slightly different, but of the exact same object. This was done by creating sets of images of different shapes. We created 10 images each of a simple square, of a basic triangle, and a diamond as seen in Figure 3.2. In addition we created a set of images that consisted of squares with tails oriented in different directions Figure 3.3. These tails were used to test the ability of WARP to classify images that are just slightly different in the time series. The first experiment run involved creating a probe consisting of images of one shape and then a gallery of a completely different shape, and using the average DTW distance to classify. By doing this every image was properly classified. The next experiment involved creating a probe and gallery set that included images from each shape and doing a direct comparison based on the distance between each probe image and each gallery image. In this experiment the recognition rate was 97.3 percent. The final experiment that we ran for testing involved creating a probe and gallery set consisting of square images with tails in different locations, as well as a few other shapes. This test returned excellent results of 98.4 recognition rate, and all images with different tails were properly classified, indicating that the WARP process worked as advertised. With these numbers we were able to continue on and test how effective WARP is with our fish database.

For our experiments involving the WARP process we began with a set of data provided by the University of New Orleans that was obtained through a joint project between the school and the National Oceanic and Atmospheric Administration (NOAA). This data was obtained by placing a video camera underwater inside a waterproof container to capture footage over time. This video was then broken down into greyscale images. These images were organized into several different groups. The first group includes images that were separated by different species. The problem with these images was the small number per species, as well as the fact that the images for each species were recorded in a different time and different place from all the other places. This prevented us from using the background subtraction because only the E. morio indexE. morioset of images had enough to perform proper background subtraction on. The other set of data came from one time

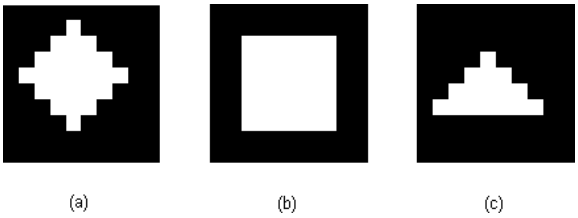


Figure 3.2: 3 different test shapes used

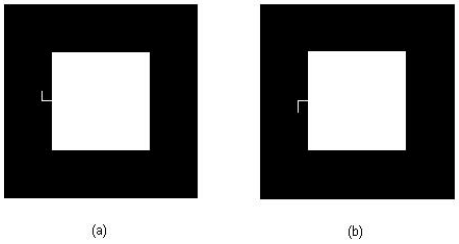


Figure 3.3: 2 examples of same shape with slight alterations to test DTW correctness

period of filming. This set had 1800 images available at a resolution of 1280 by 960 pixels, and we performed background subtraction on each image.

After background subtraction, each image was searched to see if any fish were present in the image through simple blob detection. If the blob was large enough, The pixels inside the bounding box were then segmented out into a separate and smaller image of 281 by 121 pixels. In order to test WARP's ability to consider rotation, images were not rotated to orient the major axis of the fish along the x-axis. After each fish had been segmented from the original image we manually went through the fish images to select our testing set. This manual search was necessary because as much of an improvement we have in background substitution, there are still images that can not be used for classification. After this was done we were left with a complete testing set of 54 images. While this is certainly minuscule compared to the original 1800 images, it is important to know that of the 1800 images, only 129 images had any fish in them. Because the camera was left underwater a large portion of the footage is just of the empty ocean. see 3.4 fool!

Figure 3.4: Different Fish species used Fish 1 (a), Fish 2 (b), Fish 3 (c), Fish 4 (d)

The 54 images we had were broken up into lists based on species. Fish 1 had 6 images, Fish 2

had 2 images, Fish 3 had 16 images, and Fish 4 had 30 images. The fish were broken up based on visual inspection, and Fish 3 was the E. morio fish.

Once the data was organized properly we ran it through the WARP process in several different ways. The first way was to evenly split each species of fish into a probe set and gallery set. All of the probes were gathered together in one list, and the gallery images were placed in a separate list. Each image in the probe was then compared to every single gallery image, and when the DTW distance between the two images was less than 5, the images were considered to be of the same species. If the distance was greater than or equal to 5 the images were considered to be of different species. From this direct comparison, we had a recognition rate of 53.2 percent. This percentage was obtained by looking at the number of appropriate classifications that occurred regardless of if the images were of the same species or not. We attempted to improve the recognition rate by changing the threshold for the DTW distance, but it had not effect on the recognition rate.

The biggest problem we found that caused such a low recognition rate is that a lot of the images have enough of a different orientation that WARP can not compensate for. As shown in Figure 3.5, a fish can be turning into or away from the camera, and by doing this will distort the shape in a way that WARP can not understand when it is comparing to a fish that has an almost perfect side view as shown in Figure (). This issue caused us to use WARP in a slightly different method to help compensate for this deficiency.

Figure 3.5: 2 E. morio fish with different orientation to camera

The second WARP experiment we conducted involved computing an average distance based on the distances between one probe image a set of gallery images. This was done by setting up probe and gallery lists so that each list was made up of a single, but different species of fish. An example would be when we made the probe set consist of only E. morio fish, and the gallery was made up of the Fish 4 images. The DTW distance between an E. morio image and each Fish 4 image was calculated and averaged together. If this average was less than 5, the E. morio image was considered to be of the same species as Fish 4, and if the average was greater or equal to 5, the image was said to be of a different species. By doing this experiment and comparing every species of fish to every other species of fish, only 22 fish were misclassified out of 162 classification attempts for a recognition rate of 86.4 percent. Of interesting note for this recognition rate is that all 22 misclassification's happened when the gallery set was made up of the 6 images from Fish 1. When Fish 2 was compared only 1 of the 2 images were classified appropriately, Fish 3 had 7 of 16 images classified correctly, and Fish 4 had 18 of 30 images appropriately classified. An example of Fish 1 is shown in Figure 3.4. As shown, the background subtraction was not great for this fish, and it is also possible that Fish 1 is actually of the same species as one of the other 3 species, but because of the loss of detail, it is impossible to know. The best way to remedy this is to gather additional data as well as improve the background subtraction to see if the recognition rate involving this fish

will be improved, or if there is an underlying issue.

From these experiments it seems that using WARP with non-ideal images can produce beneficial results, but these results do not come from direct comparisons. Rather it appears that WARP is beneficial when an image can be compared to a database of fish based on a single species. If this image has a close enough average distance to a specific species set it can then be classified to that specific species.

3.3 Gabor Filters

For our work with Gabor filters, we were concerned with identifying features in a fish's look that were unique. In this case we attempted to try to identify a fish as an E. morio if it had the vertical light strip on it's tail. We used the same set of data as in the WARP process, and the background substitution and image segmentation were done in the exact same way to end up with 54 images, 16 of which were E. morio. For this experiment each image was run through both horizontal and vertical Gabor filters. The output image from the horizontal and vertical filters can be seen in Figure 3.6b and Figure 3.6c respectively. After these two output images were obtained, an image consisting of the ratio of the horizontal filter to the vertical filter was calculated. This image was then eroded slightly to significantly reduce the halo surrounding the image which is a byproduct of the background substitution and the two filters. Figure 3.6d shows this ratio image for an E. morio and non-E. morio fish. As shown, there is a distance vertical line on the tail of the image that is extremely visible after the use of the Gabor filters. Taking this output image we then turn the image into a purely black-and-white image by turning any value in the 0-.5 range to a 0 value, and any pixel value above .5 becomes a 1. This turns most of the image black, but makes the tail line white. The problem with this thresholding is that due to lighting other areas of the fish may appear white as well, as seen in Figure. 3.7. To combat this we then search the image for white space with a width between 5 and 15 pixels, and with a height of at least 15 pixels. This ensures that only vertical line segments are used in classifying the fish. If this whitespace exists, the fish is then classified as an E. morio, and the lack of this whitespace indicates the fish is not an E. morio.

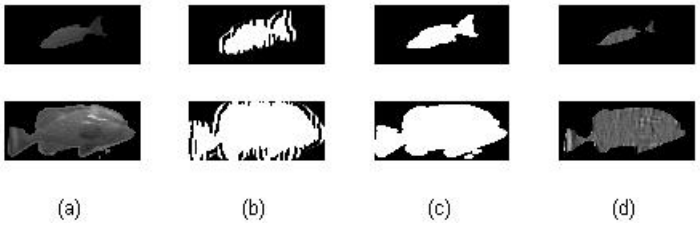


Figure 3.6: 2 Different fish images (a) with horizontal Gabor filter (b), vertical Gabor filter (c) and ratio of Gabor filters (d)

For experimentation we tested all 54 images, and had 5 misclassification's for a recognition rate of 90.7 percent. 2 of those misclassification's involved a non-E. morio being classified as E.morio, and the other 3 were a result of saying an E. morio fish was not an E.morio fish. The two non-E.morio fish were incorrectly classified because a bright light was shining on a small portion of the fish as shown in Figure 3.8. This bright light caused a strip of pixels to appear brighter than they

actually were, and led to the misclassification. For the three E. morio fish, all three fish were at extreme angles that prevented the camera from either capturing the tail fin at all, or the tail fin was so small the line did not have enough width or height to meet the whitespace criteria Figure 3.8.

Figure 3.7: non E. morio (a) and E. morio (b) ratio images after thresholding

Figure 3.8: 2 missclassed images (a) non-E. morio and (b) E. morio

3.4 Classification

3.4.1 Transformed Data

The RBM was tested on a set of simple geometric shapes, to show that it was robust to small translations and rotations. Simple 13x13 square, circle, and triangle shapes were used, and a set of transformed images was created with random shifts of 0–6 pixels and −15–15 degrees.

It was trained on the original shapes, and tested on the transformed shapes. The RBM was able to reproduce the original shape in most cases, and the noise was reduced when the input was presented three times and majority voting was used.

Similar tests were performed with the CNN. The same shapes and transformations were used to create a set of 300 transformed images. The CNN perfectly classified the testing set of 150 image, even when only trained on the original patterns.

Another, more difficult, test was performed using patterns of vertical bars. Different numbers of bars were used, but the bars appeared in the same overlapping region of the images. The same random transformations were applies to create a new set of 150 training images and 150 testing images. When trained on only the original data, the netowk was only able to classify 84 (56%) of the images. However, when trained on 150 images of the transformed data, and tested on the other half, the network was able to perfectly classify the data.

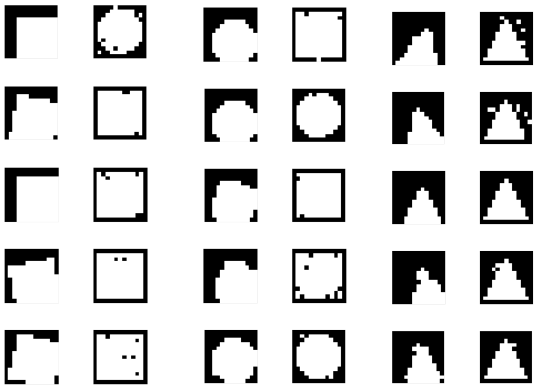


Figure 3.9: RBM results for transformed data

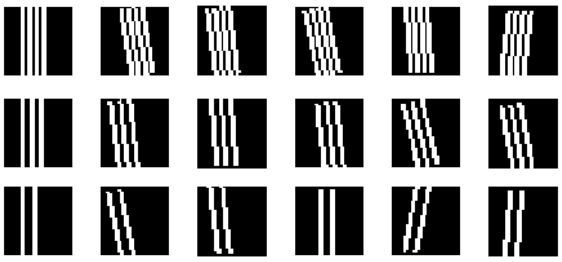


Figure 3.10: Examples of the vertical bars used to test the CNN

3.4.2 Partial Data

The RBM was also tested on a set of simple shapes intended to determine its ability to recover missing data. The network was trained on the original shapes and tested on partial versions of those same shapes. It was able to reproduce the original shapes in nearly all cases.

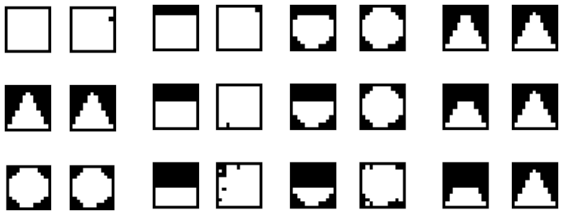


Figure 3.11: RBM results for partial data

3.4.3 Real Data

Tests were also performed on real world data. Segmented images from video streams were automatically normalized for rotation and size. Suitable images were then manually selected and divided into testing and training sets, where the training set was selected to have only high quality images. The testing set had images of more variable quality, and included images of fish not included in the training set.

The training data was 34 images, consisting of 17 extracted binary fish images and these same images flipped horizontally. Testing data was 84 images from 42 unique images.

To improve efficiency, a method of reducing inputs to the DBN was employed. Pixel locations which were off or on in 90% of the training images were assumed to have a fixed value and excluded from presentation to the DBN. These pixels were then set to their assumed values for presentation to the CNN. This significantly sped up training, and might be practical to use in a deployed system if a large enough sample is used to determine 'uninteresting' pixel locations.

7.03% of pixels (1547) were "interesting", and the DBN contained two RBMs (three layers of neurons) with 1547 neurons in each layer. Each layer was trained for 750 epochs with a learning rate of 0.1, and a weight decay of 0.0001. Momentum was initially 0.5, but after 25 epochs was increased to 0.9.

The CNN architecture used is shown in Figure 1.6. It consists of two convolution layers (with two subsampling layers) with 5 and 25 feature maps respectively, and one output layer of two neurons. The convolution layers were trained with a learning rate of 0.001 and the output layer was trained with a rate of 0.01.

In the first test, the CNN is trained and tested directly on the image data. The CNN quickly converged on the training data and achieved a high level of accuracy on the testing data. The CNN was trained for 75 epochs; training for 50 and 100 resulted in under- and overfitting respectively. overfitting, and the CNN classified with 93% accuracy (missclassifying 6 of 84).

In the second test the DBN is trained on the interesting pixels, learning a good model of the input data. The CNN is then trained and tested on the output of the DBN for a given sample. The CNN was trained for 300 epochs; training for longer or shorter didn't have much apparent difference. Results were even better than results on the raw images, with 98% accuracy (2 missclassifications).

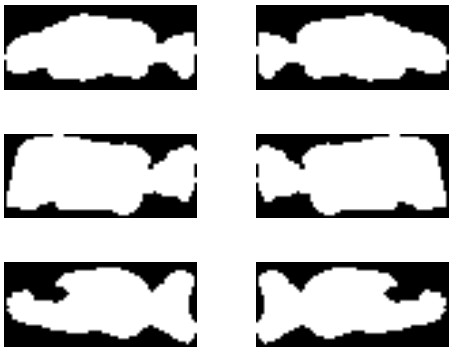


Figure 3.12: The 6 inputs missclassified by the CNN

Figure 3.13: The 2 inputs missclassified when the DBN was used as preprocessing before the CNN. On the left are the input images, and on the right are the DBN outputs.

Finally, the CNN was tested on the image data after being processed to set uninteresting pixels to their assumed value. We wanted to be sure the increased performance was due to the DBN processing, and not simply the extra normalization as a result of keeping a subset of pixels constant. Results are far worse for this test, achieving an accuracy of only 83% (14 missclassifications). Adjusting training time yielded no improvement in results.

Chapter 4

Future Work

For some member of the research team, there is the opportunity to continue the study of this problem at the respective affiliate institution. We would be able to continue collaboration with the researches at the University of New Orleans and NOAA. In this chapter we will outline what further steps could be taken to advance the overall project goals.

4.1 Thresholding

Since the results of threshold comparison did not demonstrate that a single method was optimal or always gave clearer results, it would be beneficial to implement a system to select the best threshold method. This could perhaps be investigated and applied by using statistics of the difference image between the background and original images. For instance, if an image is detected to contain a higher level of noise, certain methods may be known to perform better and could be selected to threshold that particular data. In other cases, several algorithms can be run and one of the results chosen. Ideally, a novel algorithm tailored to this application would be developed to achieve high accuracy and data conservation in the results. This, however, would require more time and study than practical at present.

Another beneficial component of the system would be to have an evaluation method of the results of different thresholding algorithms. Without ground truth data, however, this would be very difficult to obtain and objectively evaluate. Methods have been used that take user input on each image before thresholding and segmentation; this method allows the user to clarify certain areas of an object that different algorithms may not detect. This would be done, for example, by a user being able to click points along the edge of an object and also points definitely outside of an object. By this scheme, uncertain areas could be included and regions (such as shadow) could be deliberately avoided. Of course, with the breadth of our data, this could be an impractical approach. However, a method could be devised where the thresholding and segmentation process has a certainty measure and when the certainty is below a particular value, the data is passed to the user-assisted part of the system.

4.2 Features

Now that both the WARP process and classification via Gabor filters has shown early success, the next step is to run more experiments with new data. By running both experiments up against larger fish databases, it will give us a chance to see just how effective both processes are with images that are not perfect. This will require getting additional data from NOAA that can be used. Ideally a sizable set of each species of fish we are interested in classifying could be provided for use in WARP and to further testing the Gabor filter method of identifying E. morio fish.

The second step to continue work on the features is to identify unique features of different fish species similar to the E. morio tail stripe. If certain features like that can be located, it would be possible to have a modified Gabor filter experiment to look for these features, or maybe some other filter or feature extraction method could be used.

4.3 Classification

Now that both network architectures have been validated, the next step is to further fine-tune the training parameters. A larger data set for testing will also increase confidence in these results, and will enable testing this method on other fish species as well.

Additionally, other sorts of inputs should be considered. These features will be necessary to distinguish fish with similar shapes and might include the original pixel values rather than the binary version of the image, filtered images, and other extracted features. For these tests, other network architectures should be investigated as well, including MLPs, SVMs, and decision trees.

Appendix A

Thresholding Results

This appendix shows some sample fish data with a superimposed outline of where objects were detected by the various threshold methods. These are presented two algorithms at a time - side by side - over the same set of fish data.

Figure A.1:
Left: Ajlan - Right: Kittler

Figure A.2:
Left: Ajlan - Right: Kittler

Figure A.3:
Right: Guo - Left: Liao

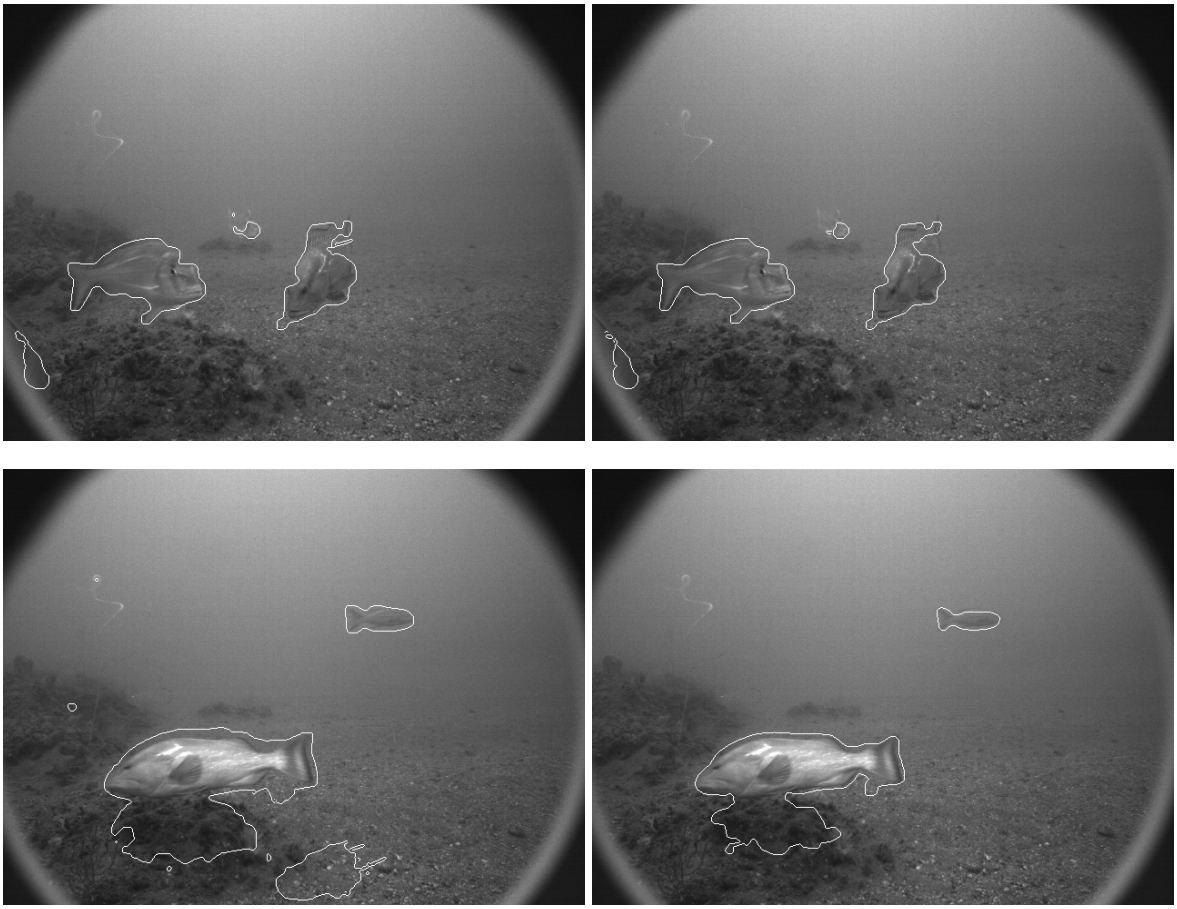


Figure A.4:
Right: Guo - Left: Liao

Figure A.5:
Left: Ramesh (sum of square error) - Right: Ramesh (sum of variance)

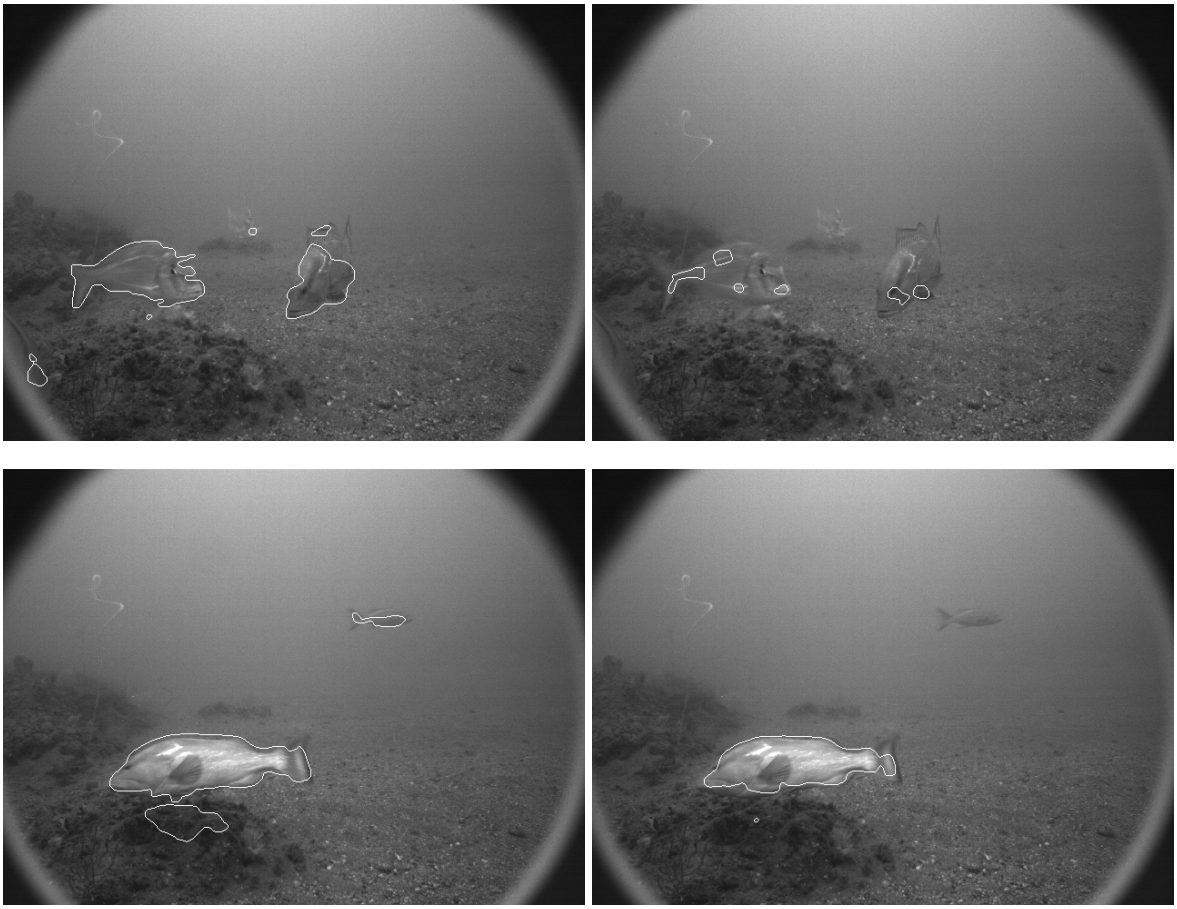


Figure A.6:
Left: Ramesh (sum of square error) - Right: Ramesh (sum of variance)

Figure A.7:
Left: Xiao - Right: Tao

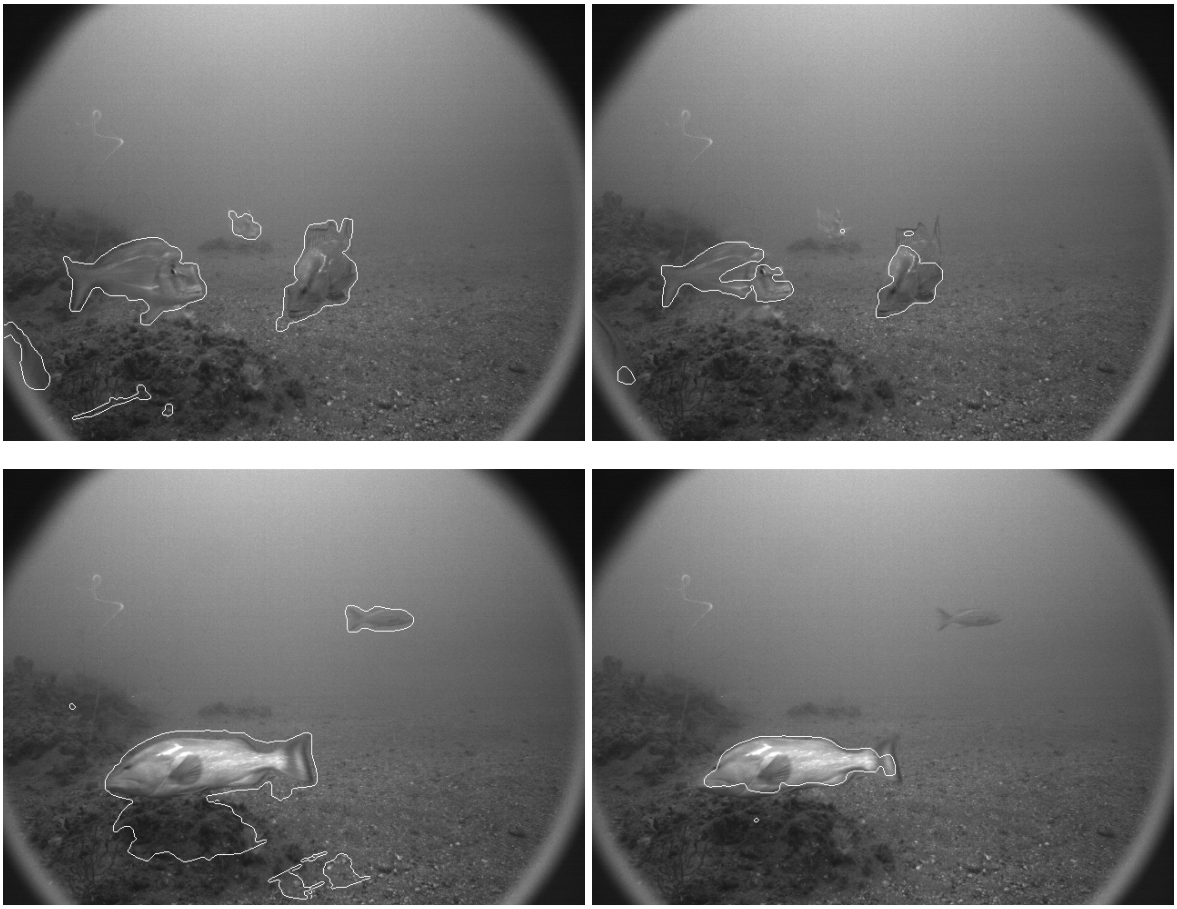


Figure A.8:
Left: Xiao - Right: Tao

Appendix B

Edge Thinning Results

A comparison is presented of how thresholding algorithms performed with and without the edge thinning operations.

Figure B.1: Ajlan
Left: Standard
Right: With Edge Thinning

Figure B.2: Ajlan
Left: Standard
Right: With Edge Thinning

Figure B.3: Guo
Left: Standard
Right: With Edge Thinning

Figure B.4: Guo
Left: Standard
Right: With Edge Thinning

Figure B.5: Kittler
Left: Standard
Right: With Edge Thinning

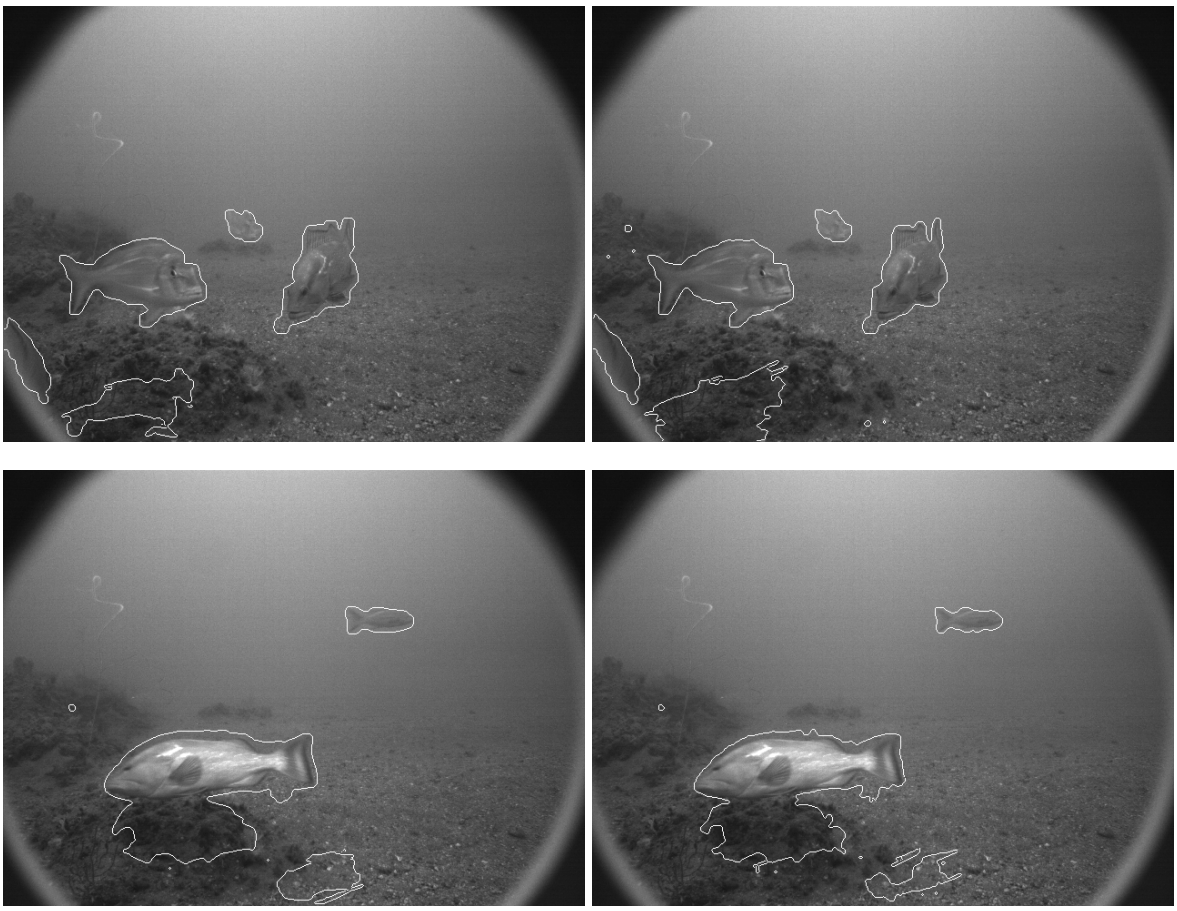


Figure B.6: Kittler
Left: Standard
Right: With Edge Thinning

Figure B.7: Liao
Left: Standard
Right: With Edge Thinning

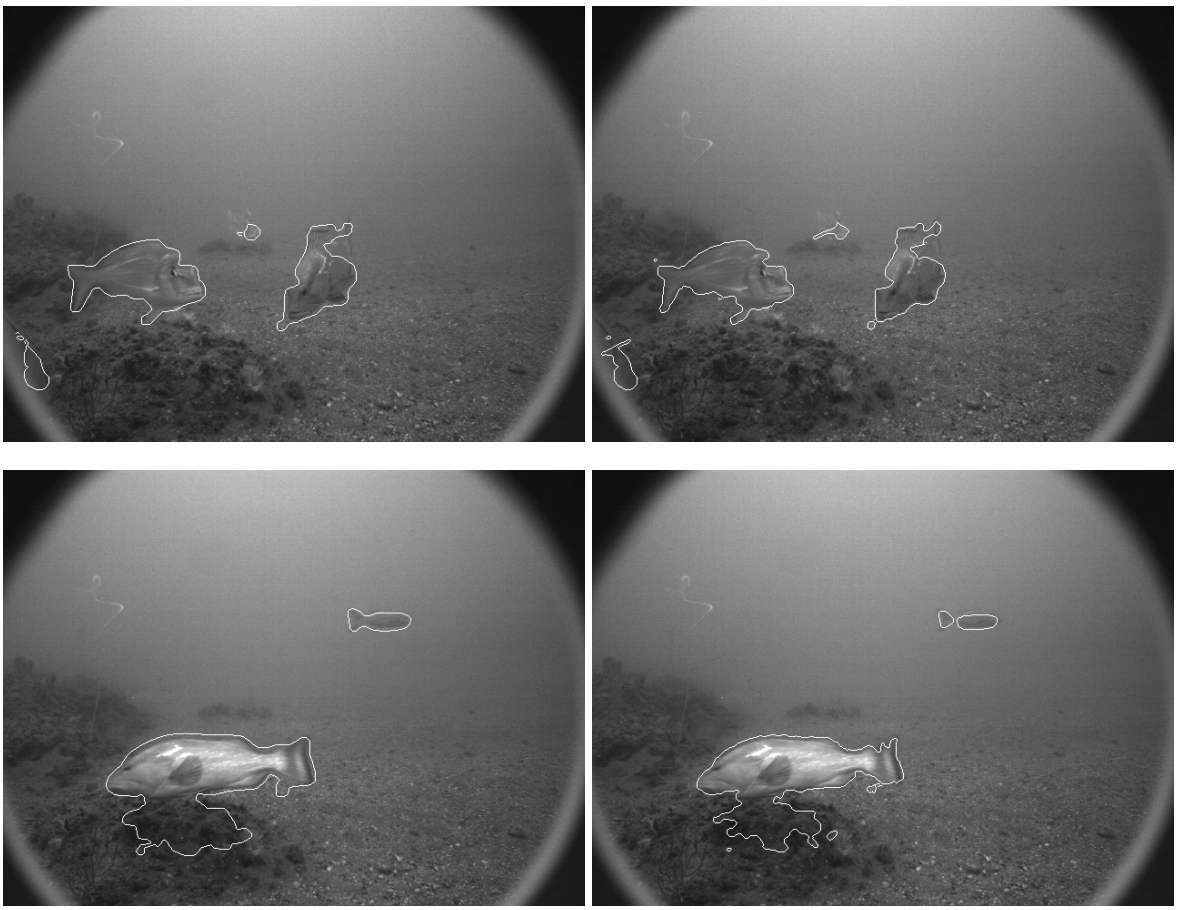


Figure B.8: Liao
Left: Standard
Right: With Edge Thinning

Figure B.9: Otsu
Left: Standard
Right: With Edge Thinning

Figure B.10: Otsu
Left: Standard
Right: With Edge Thinning

Figure B.11: Ramesh (Sum of Square Error)
Left: Standard
Right: With Edge Thinning

Figure B.12: Ramesh (Sum of Square Error)
Left: Standard
Right: With Edge Thinning

Figure B.13: Ramesh (Sum of Variance)
Left: Standard
Right: With Edge Thinning

Figure B.14: Ramesh (Sum of Variance)
Left: Standard
Right: With Edge Thinning

Figure B.15: Tao
Left: Standard
Right: With Edge Thinning

Figure B.16: Tao
Left: Standard
Right: With Edge Thinning

Figure B.17: Xiao
Left: Standard
Right: With Edge Thinning

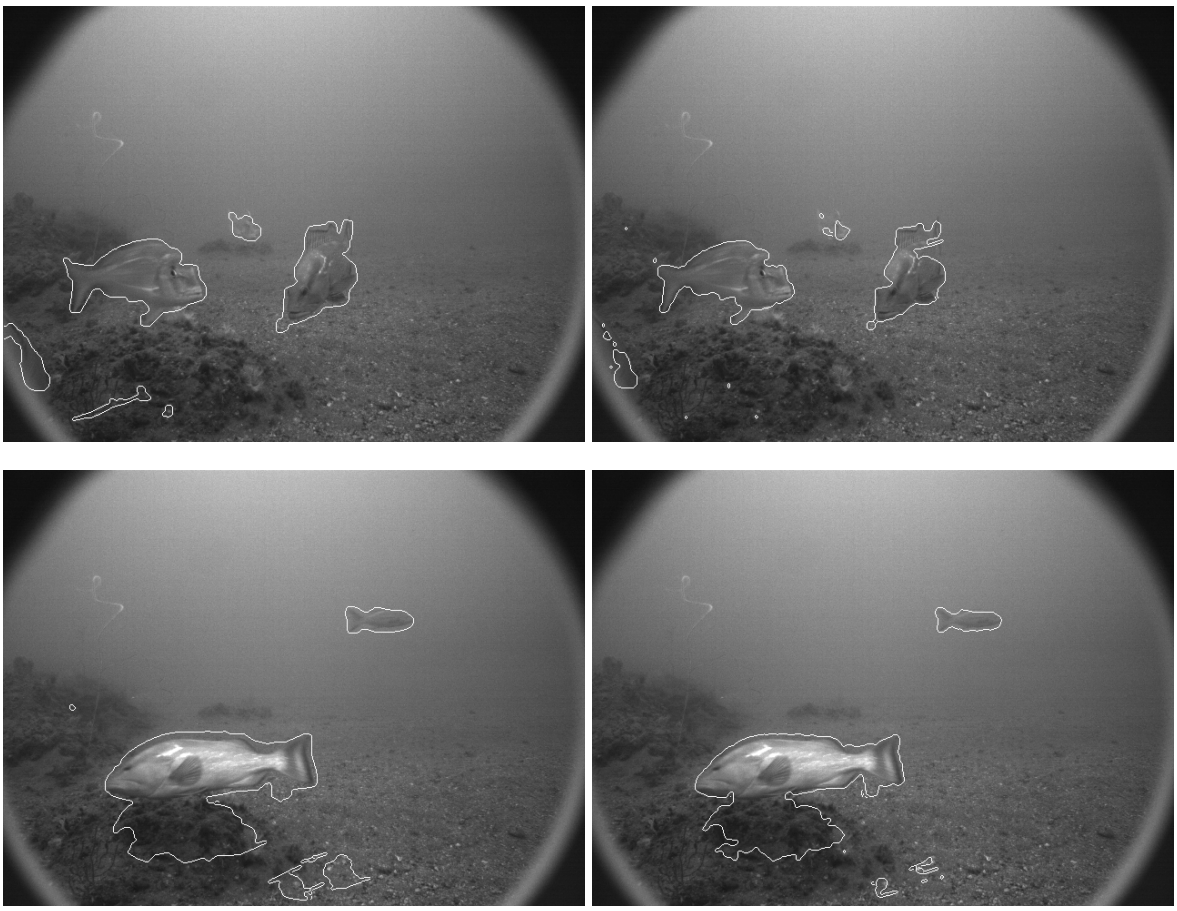


Figure B.18: Xiao
Left: Standard
Right: With Edge Thinning

Appendix C

Threshold Code

C.1 Graph Cut: Tao *et al*'s Method [42]

```
1  function [newImg] = Tao(img)
2  %   This function will use a graph cut method to determine
3  %   an optimal threshold for a grayscale image.
4  %
5  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
6  %       @version 2011-07-12
7  %
8  %   param img A matrix of unsigned integers from 0 to 255
9  %
10 %   return newImg A binary image (0 or 255) thresholded by Tao's graph cut
11 %   scheme
12 %
13
14     % This will be a matrix where M(i,j) is the cut between pixels of level i and j
15     global M1;
16
17     % A copy of the parameter image
18     myImage = img;
19
20     % The neighborhood size to consider when calculating a node's weight
21     r = 4;
22
23     % A shape parameter to control the strictness of intensity difference of two nodes
24     dI = 625;
25
26     % A shape parameter to control the strictness of location difference between two nodes
27     dX = 8;
28
29 %   Calculate the distances and weight components that can be used later
30     d = repmat(1:2*r+1,[2*r+1,1]);
31
```

```
32      % A mask that will hold distance values for any pixel in the diameter r neighborhood
33      distLookup = sqrt( (d-(r+1)).^2 + (d'-(r+1)).^2 );
34
35       % A matrix that will hold the distance component of a node's weight
36      weightDistLookup = exp( -1*( (distLookup.^2)/dX ));
37
38      % Pad the image with -1 to avoid edge collision
39      padImg = padarray(int16(myImage),[r r],-1);
40
41      % The locations in the padded image that correspond to the original.
42      [x y] = find(padImg>=0);
43
44      % Allocate space for the matrix of cuts
45      M = zeros(256,256);
46
47  %   Calculate the cuts between class i and j pixels using appropriate
48  %   lookup tables.  Note that the intensity component of the weight is
49  %   not multiplied through until after this process.
50      for i=min(x):max(x)
51          for j=min(y):max(y)
52              center = padImg(i,j);
53              for k=-r:r
54                  for l=-r:r
55                      testPixel = padImg(i+k,j+l);
56                      if testPixel >= center && weightDistLookup(k+r+1,l+r+1) < r
57                          M(center+1,testPixel+1) = M(center+1,testPixel+1) ...
58                          + weightDistLookup(k+r+1,l+r+1);
59                      end
60                  end
61              end
62          end
63      end
64
65  %   Now we will multiply the cut matrix by the intensity differences.  This
66  %   can be obtained simply from the i and j location in the matrix.
67      m = repmat(1:256,[256 1]);
68      exM = exp(-1*( (abs(m-m').^2)/dI ));
69      M = M .* exM;
70      M1 = M;
71
72      NCuts = zeros(1,256); % The matrix to hold the NCut values
73
74  %   Calculate the NCuts
75      for i=0:255
76          NCuts(i+1) = Ncut(i);
77      end
78
79  %   Find the minimum NCut as per Tao's method and use it as the threshold
80      ncut_min = min(min(NCuts(NCuts>0)));
81      t = find(NCuts == ncut_min);
```

```
82      t = t(1)-1; % Because the image is from 0 to 255 and the search if from 1 to 256.
83
84  %   Adapt the provided image with the calculated threshold and return it as
85  %   a grayscale binary image (0 or 255)
86      newImage = zeros(size(myImage));
87      newImage( myImage > t ) = 255;
88      newImg = newImage;
89      return;
90  end
91
92  %-------------------------------------------------------------------------
93
94  function [nc] = Ncut(t)
95  %   This function calculates the NCut as derived by Tao.
96  %
97  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
98  %       @version 2011-07-12
99  %
100 %   param t The threshold candidate
101 %
102 %   return nc the ncut using threshold t
103 %
104
105     % We will need to access a previously computed matrix of
106     % class cuts between class i and j valued nodes
107     global M1;
108
109     % The asso(A,B) value obtained from the M matrix
110     assocAA = sum(sum(triu( M1(1:t+1,1:t+1) )));
111
112 %   if t is 255, then asso(A,A) will be the entire M matrix. Otherwise,
113 %   calculate as per Tao's method
114     if t < 255
115         assocBB = sum(sum(triu( M1(t+2:256,t+2:256) )));
116         cAB = sum(sum(M1(1:t+1,t+2:256)));
117     else
118         assocBB = 0;
119         cAB = 0;
120     end
121
122      % Add the small constance to prevent divide by 0 runtimeerror.
123     nc = ( cAB / (assocAA + cAB + 0.000000000000001) ) ...
124     + ( cAB / ( assocBB + cAB + 0.000000000000001) );
125
126     return;
127 end
```

C.2 Histogram-Based: Otsu's Method [29]

```
1  function [newImg] = Otsu(img)
```

```
%       Function Otsu
%
%       This function calculates an optimal threshold based on Otsu's class
%       variance method and returns the thresholded image.  This assumes that
%       the image uses 256 gray levels (0-255)
%
%       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
%       @version 2011-07-12
%
%       @param img The image to be thresholded
%       @return newImg The thresholded image
%
        % Store a copy of the parameter image.
    myImage = img;

    % Create a histogram
    H = imhist(myImage);
    H = H';

    % Create a normalized histogram
    P = H / sum(H);

    % Calculate the cumulative mean
    cmean = sum(P .* [1:256]);

        % Cacluate the class variance for each possible threshold
    sigArray = zeros(1,256);
    for i = 1:256
       sigArray(i) = SigmaB(cmean, i, P);
    end

        % Find the threshold that maximizes the class variance.
    t = find( sigArray == max(sigArray( sigArray > 0 )) ) - 1;

        % Binarize the parameter image according to the calculated threshold.
    newImage = myImage;
    newImage( newImage <= t ) = 0;
    newImage( newImage > t ) = 255;

    newImg = newImage;
    return;
end

%-------------------------------------------------------------------------

function [sig] = SigmaB( cmean, k, P )
%       Function SigmaB
%
%       This function calculates the between-class variance with respect
%       to a particular threshold k
```

```
%
%       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
%       @version 2011-07-12
%
%       @param cmean The cumulative mean of the image being thresholded
%       @param k The threshold
%       @param P The normalized histogram of the image
%
%       @return sig The between-class variance of the image with respect to k
%
        % Calculate the zeroth and first order moments
    moment0 = sum( P(1:k) );
    moment1 = sum( P(1:k) .* [1:k] );

    % only accept a variance >= 0
    if moment0*(1-moment0) > 0
        sig = ( ( cmean * moment0 - moment1 ) ^ 2 ) / ( moment0*(1-moment0) );
    else
        sig = -1;
    end
end
```

C.3 Histogram-Based: Liao's Method [23]

```
function [newImg] = Liao(img)
%       This function will implement Otsu's method using lookup tables to
%       optimze space and speed. This uses 3 layers.  At the end of the function, for the
%       sake of experimentation, the top 2 layers are combined into a single foreground.
%
%       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
%       @version 2011-07-12
%
%       param img The image to be thresholded.
%
%       return newImg The thresholded image.
%
    myImage = img;

    % The number of possible gray intensities
    greys = 256;

     % Set up a histogram of the image.
    H = imhist(myImage);
    H = H';

    % Create a normalized histogram.
    H = H / sum(H);

    % Create lookup table matrices for faster computation
    % according to the paper by Liao et al. 2001
```

```
27      [P,S] = GenerateLookupTables(H);
28
29          % The number of classes in which to threshold the image.
30      layers = 3;
31
32          % The number of variations that will be calculated.
33      Sigma = zeros( ones(1, layers-1)*( greys ) );
34
35      % Calculate the variance for each class according to otsu's method
36      for i=1:numel(Sigma)-1
37          [x,y] = ind2sub(size(Sigma), i); % The row and column of a candidate threshold.
38          ind = [x,y];
39          if issorted(ind) && x ~= y && y < greys
40              Sigma(i) = ( S(1, ind(1))^2 ) / P(1, ind(1));
41              for j=2:length(ind)
42                  Sigma(i) = Sigma(i) + (S(ind(j-1)+1, ind(j))^2)/P(ind(j-1)+1, ind(j));
43              end
44              Sigma(i) = Sigma(i) + (S(ind(end)+1, greys)^2)/P(ind(end)+1, greys);
45          end
46      end
47
48      % Fix all the invalid math results
49      Sigma( isnan(Sigma) ) = 0;
50      maxSig = max(Sigma);
51
52      % Find the single largest value in the Sigma matrix.
53      while numel(maxSig) > 1
54          maxSig = max(maxSig);
55      end
56      [t1,t2] = ind2sub(size(Sigma), find(Sigma == maxSig));
57      t = [t1, t2] - 1;
58
59      % This will threshold an image to only two levels.
60      newImage = zeros(size(myImage));
61          newImage(newImage > t(1)) = 255;
62
63      newImg = newImage;
64      return;
65  end
66
67  %-------------------------------------------------------------------------
68
69  function [PMatrix,SMatrix] = GenerateLookupTables(H)
70  %
71  %       This function will compute lookup tables for the
72  %       u-v interval zeroth and first order moments of a
73  %       cass with gray levels from u to v.
74  %
75  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
76  %       @version 2011-07-12
```

```
77 %
78 %       param H The normalized histogram. Should be a row matrix
79 %
80 %       return PMatrix The lookup table of zeroth-order moments of the
81 %       histogram
82 %
83 %       return SMatrix The lookup table of first-order moments of the
84 %       histogram
85 %
86      levels = length(H);
87      P = zeros(levels, levels);
88      S = zeros(levels, levels);
89      for row = 1:levels
90          for col = row:levels
91              if row == 1
92                  if col == 1
93                      P(row, col) = H(col);
94                      S(row, col) = col*H(col);
95                  else
96                      P(row, col) = P(row, col-1) + H(col);
97                      S(row, col) = S(row, col-1) + col*H(col);
98                  end
99              else
100                 P(row, col) = P(1, col) - P(1, row-1);
101                 S(row, col) = S(1, col) - S(1, row-1);
102             end
103         end
104     end
105     PMatrix = P; % The matrix of zeroth-order moments
106     SMatrix = S; % The matrix of first-order moments
107 end
```

C.4 Histogram-Based: Ramesh [35]

C.4.1 Sum of Square Error

```
1  function [ newImg ] = Ramesh_SOSE( img )
2  % This function calculates an optimal threshold using a sum of square errors.
3  %
4  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
5  %       @version 2011-07-12
6  %
7  % param img The image to be thresholded.
8  %
9  % return newImg The image after the threshold has been applied.
10 %
11     myImage = img;
12
13     H = imhist(myImage);
14     H = H';
```

```
15      % H now holds the histogram values of the grayscale version of our image.
16
17      % Calculate the errors:
18      ErrorArray1 = zeros(1, 256);
19      for i = 1:256
20          ErrorArray1(i) = Error1(i-1, H);
21      end
22
23      e1Min = min( ErrorArray1( ErrorArray1 > 0   ) );
24      t1 = find( ErrorArray1 == e1Min )-1;
25      t1=t1(1);
26
27      NewImg1 = myImage;
28      NewImg1( NewImg1 <= t1  ) = 0;
29      NewImg1( NewImg1 > t1  ) = 255;
30
31      newImg = NewImg1;
32  end
33
34  %-------------------------------------------------------------------------
35
36  function [error] = Error1(t, H)
37  %       Function Error1
38  %
39  %       This function calculates the error between two classes
40  %
41  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
42  %       @version 2011-07-12
43  %
44  %       @param t The threshold which separates the two classes
45  %       @param H The image histogram
46  %
47  %       @return error The error between classes
48  %
49          L = length(H);
50          if length(t) > 1
51                  error = -1;
52                  return;
53          end
54          if t == length(H)
55             error = -1;
56             return;
57          end
58          if t < 1
59             error = -1;
60             return;
61      end
62
63          H2 = H .* [1:length(H)];
64
```

```
65          if sum( H(1:t+1) ) > 0
66                  m1 = sum( H2(1:t+1) ) / sum( H(1:t+1) );
67          else
68                  error = -1;
69                  return
70          end
71
72          if sum( H(t+2:length(H)) ) > 0
73                  m2 = sum( H2(t+2:length(H2)) ) / sum( H(t+2:length(H)) );
74          else
75                  error = -1;
76                  return;
77          end
78
79          temp1 = H(1:t+1) .* ( ([0:t] - m1) .^ 2 );
80
81          temp2 = H(t+2:end) .* ( ([t+1:L-1] - m2) .^ 2 );
82
83          % calculate the total error
84          error = sum(temp1) + sum(temp2);
85          return;
86  end
```

C.4.2 Sum of Variances

```
1   function [newImg] = Ramesh_SOV  (img)
2   %       This function uses the sum of variances to calculate an optimal image threshold
3   %
4   %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
5   %       @version 2011-07-12
6   %
7   %       @param img The image to be thresholded.
8   %
9   %       @return newImg The thresholded image.
10  %
11      myImage = img;
12
13      H = imhist(myImage);
14      H = H';
15      % H now holds the histogram values of the grayscale version of our
16      % image.
17
18      % Calculate the errors:
19      ErrorArray2 = zeros(1, 256);
20      for i = 1:256
21          ErrorArray2(i) = Error2(i-1, H);
22      end
23
24      e2Min = min( ErrorArray2( ErrorArray2 > 0 ) );
25      t2 = find( ErrorArray2 == e2Min )-1;
26      t2=t2(1);
```

```
27
28      NewImg2 = myImage;
29      NewImg2( NewImg2 <= t2 ) = 0;
30      NewImg2( NewImg2 > t2 ) = 255;
31
32      newImg = NewImg2;
33  end
34
35  %-------------------------------------------------------------------------
36
37  function [error] = Error2(t, H)
38  %       Function Error1
39  %
40  %       This function calculates the variance between two classes
41  %
42  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
43  %       @version 2011-07-12
44  %
45  %       @param t The threshold which separates the two classes
46  %       @param H The image histogram
47  %
48  %       @return error The error between classes
49  %
50          L = length(H);
51          if length(t) > 1
52                  error = -1;
53                  return;
54          end
55          if t == length(H)
56             error = -1;
57             return;
58          end
59          if t < 1
60             error = -1;
61             return;
62          end
63
64          H2 = H(1:length(H)) .* [1:length(H)];
65
66          if sum( H(1:t+1) ) > 0
67                  m1 = sum( H2(1:t+1) ) / sum( H(1:t+1) );
68          else
69                  error = -1;
70                  return
71          end
72          if sum( H(t+2:length(H)) ) > 0
73                  m2 = sum( H2(t+2:length(H2)) ) / sum( H(t+2:length(H)) );
74          else
75                  error = -1;
76                  return;
```

```
77          end
78
79          temp1 = H(1:t+1) .* ( ([0:t] - m1) .^ 2 );
80
81          temp2 = H(t+2:end) .* ( ([t+1:L-1] - m2) .^ 2 );
82
83          N1 = sum(H(1:t+1));
84          N2 = sum(H(t+2:end));
85          error = (1/(N1 - 1))*sum(temp1) + (1/(N2-1))*sum(temp2);
86
87  end
```

C.5 Entropy: Xiao [47]

```
1  function [newImg] = Xiao(img)
2  %Xiao A Bilevel threshold algorithm
3  %     I = Xiao(img) will use Xiao's gray-level spacial correlation
4  %     algorithm.
5  %
6  %         @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
7  %         @version 2011-07-12
8  %
9  %     @param img Any image matrix to be thresholded.
10 %
11 %     @return newImg An image thresholded into 2 classes.  One will be of value
12 %         0 and the other of value 255.
13 %
14      myImage = img;
15
16      % the size of the image
17      im_size = size(myImage);
18
19      % the number of possible gray levels
20      g = 256;
21
22      % the radius of a neighborhood
23      N = 3;
24
25      % the similarity threshold
26      zeta = 4;
27
28      % the 2-dimensional probability function
29      P = zeros(N*N,g);
30
31      % the matrix of similar pixel counts at location G(i,j)
32      G = zeros(g,g);
33
34      % This for loop will generate G(x,y) a matrix of the same dimensions as
35      % the argument image. G(x,y) is a count of how many neighbors of pixel
36      % (x,y) in a NxN neighborhood are within zeta gray levels.
```

```
37      for i=N:im_size(1)-N+1
38          for j=N:im_size(2)-N+1
39              nbd = myImage(i-((N-1)/2):i+((N-1)/2), j-((N-1)/2):j+((N-1)/2));
40              c = myImage(i,j);
41              s = sum( abs(nbd - c) < zeta );
42              while numel(s) > 1
43                  s = sum(s);
44              end
45              G(i,j) = s;
46          end
47      end
48
49      % P is the two-dimensional histogram that represents the probability of
50      % a pixel being at a particular gray level and having m similar
51      % neighbors.
52      for i=1:N*N
53          ind = find( G==i );
54          P(i,:) = hist(double(myImage(ind)),[0:255])/length(myImage(:)); %#ok<NBRAK,FNDSB>
55      end
56
57      % This calls a function to calculate the entropy as a result of a
58      % particular threshold (i).
59      Ent = zeros(1,g);
60      for i=1:g
61          Ent(i) = Shannon(P,i,N);
62      end
63
64      EntMax = max(max(Ent));
65
66      t = find(Ent == EntMax);
67      t = t(1) - 1; % The threshold is adjusted because Matlab indexes from 1 to 256
68
69      newImage = myImage;
70      newImage(newImage <= t) = 0;
71      newImage(newImage > t ) = 255;
72      newImg = newImage;
73      return;
74  end
75
76  %-------------------------------------------------------------------------
77
78  function [ent] = Shannon(P, tArray, N)
79  % Shannon Calculates Shannon entropy
80  %  Note that this function uses a weight muliplied to each component of the
81  %  entropy sum.
82  %
83  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
84  %       @version 2011-07-12
85  %
86  %       @param P The normalized histogram
```

```
87 %       @param tArray The array of thresholds to be used.
88 %       @param N The number of pixels to be considered for similarity in a neightborhood.
89     HArray = zeros(1, length(tArray)+1); % This will hold each summation of the entropy.
90     sz = size(P);
91
92     % This calculates the weight as defined by Xiao in his paper on thresholding
93     % with gray-level correlation.
94     w = (1+exp((-9*(1:N*N))/(N*N))./(1-exp((-9*(1:N*N))/(N*N)));
95
96     % This allows the following loop to simply calculate the entropy sum from a(i-i) to a(i)
97     a = [0 tArray sz(2)];
98
99     % The following loop will calculate the shannon entropy of each class
100    % as dictated by Xiao in his paper.  It is essentially Shannon entropy
101    % with a weight on each term of the sum.
102    for i = 1:length(a)-1
103        PA = sum(sum(P(:,a(i)+1:a(i+1)))); % The sum of all the probabilities in the current c
104        HArray(i) = 0;
105        if PA ~= 0
106            P1=P(:,a(i)+1:a(i+1))+0.0000000000001; % So that we will not end up dividing by ze
107            E = (P1/PA).*log10(P1/PA); % This is the basic shannon entropy.
108            dim = size(E);
109
110            % Set up a matrix of weights. The weight is determined by the size of the
111            % neighborhood and the number of similar neighbors.
112            WM = repmat(w',1,dim(2));
113            E = E .* WM; % Weight the various entropys.
114            HArray(i) = sum(sum( E )); % Sum the array of entropy
115        end
116    end
117
118    % Factor in a negative one because the log will result in a negative  value when the
119    % ratio P1/PA is less than 10
120    ent = (-1)*sum(HArray);
121 end
```

C.6 Entropy: Guo [11]

```
1  function [newImg] = Guo(img)
2  %       This function implements Guo's method of Bilevel thresholding using Shannon
3  %       entropy.
4  %
5  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
6  %       @version 2011-07-12
7  %
8  %       @param img The image to be thresholded
9  %
10 %       @return newImg The new (binarized) version of the parameter images
11 %
12     myImage = img;
```

```
13
14      % Create a histogram
15      H = imhist(myImage);
16      H = H';
17
18      %Create a normalized histogram
19      P = H / sum(H);
20
21      entArray = zeros(1, 256);
22      for i=1:256
23          entArray(i) = shannon(P, i);
24      end
25
26      t = find( entArray == max( entArray( entArray > 0 ) ) ) - 1;
27
28      newImage = myImage;
29      newImage( newImage <= t ) = 0;
30      newImage( newImage > t ) = 255;
31      newImg = newImage;
32      return;
33  end
34
35  %-------------------------------------------------------------------------
36
37  function [ent] = shannon( P, t )
38  % This function calculates shannon entropy for a bilevel threshold.
39  %
40  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
41  %       @version 2011-07-12
42  %
43  %       @param P The (normalized) histogram to be analyzed.
44  %       @param t The threshold candidate.
45  %
46  %       @return ent The entropy using the provided threshold candidate.
47  %
48      PA1 = sum( P(1:t) );
49      PA2 = sum( P(t+1:end) );
50
51      if PA1 == 0 || PA2 == 0
52          ent = -1;
53          return;
54      end
55
56      A1 = P(1:t) / PA1;
57      A2 = P(t+1:end) / PA2;
58
59      HA1Array = A1 .* log10( A1 );
60      HA1 = (-1) * sum( HA1Array( ~isnan(HA1Array) ) );
61
62      HA2Array = A2 .* log10( A2 );
```

```
63      HA2 = (-1) * sum( HA2Array( ~isnan(HA2Array) ) );
64
65      ent = HA1 + HA2;
66  end
```

C.7 Entropy: Ajlan [2]

```
1   function [newImg] = Ajlan(img)
2   %       This function uses a cross-entropy funciton to calculate the optimum threshold
3   %       for a grayscale image.
4   %
5   %           @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
6   %           @version 2011-07-12
7   %
8   %       @param img The image to be thresholded.
9   %
10  %       @return newImg The original thresholded into two (2) classes.
11  %
12          % Store the argument image in a new variable.
13      myImage = img;
14
15      % The image histogram
16      H = imhist(myImage);
17      H = H';
18
19      % use Ajlan's method to calculate the threshold
20      t = threshold(H);
21
22      % Create a new variable that will hold the image and be modified according to the threshol
23      newImage = myImage;
24      newImage(newImage <= t) = 0;
25      newImage(newImage > t) = 255;
26
27      % Return the thresholded image.
28      newImg = newImage;
29      return;
30  end
31
32  %-------------------------------------------------------------------------
33
34  function [t] = threshold(H)
35  %   This function will use the cross-entropy to calculate an optimal
36  %   threshold for the provided (normalized) histogram.
37  %
38  %       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
39  %       @version 2011-07-12
40  %
41  %   @param P The normalized histogram (presumably of some grey-level image)
42  %
43  %   @return t The optimal treshold obtained by the cross-entropy method
```

```
%
        % This is a dummy number.  The entropy should be much lower than this for grayscale im
    min = 1000;

    % Begin the search at zero.
    thresh = 0;

    % This is the shape parameter of the gamma function.
    N = 10;

    % This variable will be used to calculate the mean.
    q = gamma(N+0.5)/(gamma(N)*sqrt(N));

    % Search all possible gray values for the optimum threshold.
    for i=1:length(H)
        % The probability distribution array for the object region.
        PO = H(1:i)/sum(H(1:i));

        %  The probability distriubtion for the background region.
        PB = H(i+1:end)/sum(H(i+1:end));

        %  The mean of the object region.
        meanO = q*sqrt(sum(H(1:i).*((1:i).^2))/sum(H(1:i)));

        % The mean of the background region.
        meanB = q*sqrt(sum(H(i+1:end).*((i+1:length(H)).^2))/sum(H(i+1:end)));

        % These must both be numbers because they will serve as indices.
        if ~isnan(meanO) && ~isnan(meanB)
                % The poisson distribution of the object region.
            QO = poisspdf([1:i], meanO); %#ok<NBRAK>

            % The poisson distribution of the background region.
            QB = poisspdf([i+1:length(H)],meanB); %#ok<NBRAK>

            % so that we will never divide by 0;
            POtmp = PO+0.0000000000000000001;
            QOtmp = QO+0.0000000000000000001;
            PBtmp = PB+0.0000000000000000001;
            QBtmp = QB+0.0000000000000000001;

            % The meat of the cross-entropy function.  See Aljan, 2008
            D = sum((POtmp-QOtmp) .* log(POtmp./QOtmp))+ sum((PBtmp-QBtmp) .* log(PBtmp./QBtmp

            %  If we have a new minimum, keep it.
            if min > D && D ~= -Inf
                min = D;
                thresh = i;
            end
        end
```

```
    end

    % Return the threshold normalized to be between 0 and 255
    t = thresh-1;
end
```

C.8 Clustering: Kittler [20]

```
function [newImg] = Kittler(img)
%       Function Kittler
%
%       This function uses Kittler et al's method of finding the optimal threshold
%       of a grayscale image.
%
%       @author Joseph T. Anderson <jtanderson@ratiocaeli.com>
%       @version 2011-07-12
%
%       @param img The image to be thresholded
%
%       @return newImg The parameter image thresholed by the calculated threshold
    myImage = img;

    % the number of possible gray levels
    g = 256;

    % the histogram of the image
    H = imhist(myImage);
        H = H';

        % what will later be the criterion function
    J = zeros(1,g);

        % calculate the components of the criterion function with respect to different thresho
    for i=1:g-1
        P1 = sum(H(1:i));
        P2 = sum(H(i+1:end));
                u2 = zeros(1,g-i);

                u1 = sum( H(1:i) .* [1:i] )/P1;
                u2 = sum( H(i+1:g) .* [i+1:g] )/P2;

                sigmaS1 = sum( (([1:i] - u1) .^ 2) .* H(1:i) )/P1;
                sigma1 = sqrt(sigmaS1);

                sigmaS2 = sum( (([i+1:g] - u2) .^ 2) .* H(i+1:g) )/P2;
        sigma2 = sqrt(sigmaS2);

        J(i) = 1 + 2*(P1*log(sigma1)+P2*log(sigma2)) - 2*(P1*log(P1) + P2*log(P2));
        J( isnan(J) ) = 0;
        J( J == -Inf ) = 0;
```

```
43      end
44
45      t = find(J==min(min(J)))-1;
46      tt = find( J(J<0)==min(min(J)));
47
48      newImage = myImage;
49      newImage(newImage <= t) = 0;
50      newImage(newImage > t) = 255;
51
52      newImg = newImage;
53      return;
54  end
```

Appendix D

Classification Code

D.1 Architectures

D.1.1 CNN

```
1  classdef CNN < handle
2      %CNN convolutional neural network class
3      %
4      % Usage:
5      %   net=CNN(inputDim, [layer1N layer2N], [layer3N], [10^-3 10^-3 10^-2])
6      %   %for each epoch
7      %    net.rates=net.rates*k   %decrease rates according to schedule
8      %    %for each input
9      %     net.input=input        %set the input
10     %     net.forward()
11     %     net.backward(target)   %backprop based on target output
12     %   net.input=(testInput)  %get output for test input
13     %
14     % I believe this code is fairly optimized.  The main bottlenecks are the
15     % convolutions and matrix multiplications required, and the function calls
16     % to f, fp, rot180, and upsamp.  Performance could probably be increased by
17     % moving these inline, at the cost of maintainability.
18     %
19     % implementation was based in parts on Jake Bouvries notes, available at
20     % http://www.math.duke.edu/%7Ejvb/papers/cnn_tutorial.pdf
21
22     properties
23         inputDim     %vector of the input dimensions
24         layers={}    %cell array of the layers
25         convN        %number of convolutional layers (rest fully connected)
26         convSize=5   %kernel size for the convolutions
27         weights={}   %cell array of weights
28         biases={}    %cell array of biases
29         rates        %array of layer learning weights
```

```
30          %f=@(x)1.7159*tanh(x)     %activation function (scaling lets us use -1 and
31                                    %  and +1 as targets efficiently)
32          %fp=@(y)1.7159*(1-y.^2)   %its derivative (takes the OUTPUT of the node)
33          f=@(x)tanh(x);            %better performance was found using unscaled
34          fp=@(y)1-y.^2;            %  activation functions, at least for now
35          input       %network input
36          output      %network output
37          sens={}     %cell array of sensitivities calculated during backprop
38          grad={}     %cell array of gradients
39
40      end
41
42      methods
43          function net=CNN(inputDim, convLayers, fullLayers, rates)
44              %creates the network
45              %  inputDim: the dimensions of the input image
46              %  convLayers: array of the number of maps in each conv layer
47              %    i.e. [layer1-num layer2-num ...]
48              %  fullLayers: size of the fully connected layers
49              %    i.e. [... layer(N-1)-num layerN-num]
50              %  rates: array of rates for all layers
51              %
52              %  the convolutional layers are automatically split into a
53              %  convolutional layer and a subsampling layer (as are the
54              %  rates associated with them)
55
56              %create cell arrays to hold the layers and weights.
57              net.convN=2*length(convLayers);
58              net.layers=cell(1, net.convN + length(fullLayers));
59              net.weights=cell(1,net.convN + length(fullLayers));
60
61              %initialize convolution and subsampling layers
62              for i=1:net.convN
63                  %if its odd, its a convolution layer
64                  if mod(i,2)==1
65                      num=convLayers((i+1)/2);  %get the number of feature maps
66                      if i==1     %if its the first layer, previous layer is input
67                          prevDim=inputDim;
68                          prevMaps=1;
69                      else        %otherwise get the actual previous layer
70                          prevDim=size(net.layers{i-1});
71                          prevMaps=size(net.layers{i-1},3);
72                      end
73                      %initialize states, weights, and biases
74                      net.layers{i}=zeros(prevDim(1)-4, prevDim(2)-4, num);
75                      net.weights{i}=random('normal', 0, 0.001, ...
76                          net.convSize, net.convSize, prevMaps, num);
77                      net.biases{i}=random('normal', 0, 0.0001, num, 1);
78                  %otherwise its a subsampling layer
79                  else
```

```
80                  prevDim=size(net.layers{i-1});
81                  prevMaps=size(net.layers{i-1},3);
82                  net.layers{i}=zeros([(prevDim(1)+1)/2 (prevDim(2)+1)/2 prevMaps]);
83              end
84          end
85
86          %now initialize the fully connected layers
87          for i=1:length(fullLayers)
88              num=fullLayers(i);
89              net.layers{net.convN+i}=zeros(num, 1);
90              inputs=length(net.layers{net.convN+i-1}(:));
91              net.weights{net.convN+i}=random('normal', 0, 0.005, num, inputs);
92              net.biases{net.convN+i}=random('normal', 0, 0.0001, num, 1);
93          end
94
95          %set the rate (convolution weights get repeated since they apply to
96          %subsampling layers as well)
97          net.rates=[kron(rates(1:net.convN/2), [1 1]) rates(net.convN/2+1:end)];
98
99      end
100
101     function output=forward(net)
102         %runs the forward pass of the network
103         %
104         %for the convolutional layers: for each feature map, we sum the
105         %convolutions over all input maps
106         %
107         %for the subsampling layers: for each of the input feature maps we
108         %simply subsample every other node
109         %
110         %for the fully connected layers: we do a standard MLP forward pass
111
112        for l=1:net.convN  %for the convolutional and subsampling layers
113            %get the right input
114            if l==1
115                linput=net.input+0; %make sure that if we have logical input
116                                    %we convert it into a form for conv2
117            else
118                linput=net.layers{l-1};
119            end
120
121            %if its an odd layer its a convolutional layer
122            if mod(l,2)==1
123                s=size(linput);
124                Y=zeros(s(1)-4, s(2)-4, size(net.weights{l}, 4));
125                for m=1:size(net.weights{l},4)  %for every feature map
126                    %convolve, accumulating each input map in Y
127                    for im=1:size(linput,3)  %for every input feature map
128                        %get the kernel for that input map
129                        %and perform the convolution
```

```
130                             k=net.weights{l}(:,:,im,m);
131                             c=conv2(linput(:,:,im), k, 'valid')+net.biases{l}(m);
132                             y=net.f(c);
133                             Y(:,:,m)=Y(:,:,m)+y;
134                         end
135                     end
136 
137                 %otherwise its a subsampling layer
138                 else
139                     %sample every other point
140                     %more complicated downsampling schemes are possible, for
141                     %example taking the maximum value of a region is sometimes
142                     %used, as is summing over a subregion or averaging
143                     Y=linput(1:2:end, 1:2:end, :);
144                 end
145 
146                 net.layers{l}=Y;
147             end
148 
149             %fully connected layers
150             %we take the vector form of the final subsampling layer as our
151             %first input
152             for l=net.convN+1:size(net.layers,2)  %for every fully conected layer
153                 %standard forward pass
154                 y=net.weights{l}*net.layers{l-1}(:)+net.biases{l};
155                 Y=net.f(y);
156                 net.layers{l}=Y;
157             end
158 
159             net.output=net.layers{end};
160             output=net.output;
161         end %forward
162 
163         function error=backward(net, target)
164             %backpropogation step
165             %
166             %for fully connected layers: we use standard backprop
167             %
168             %for subsampling layers: we need to get the sensitivities in the
169             %next layer that had that pixel in their input patch.  this is hard
170             %but luckily theres a simple convolution to do it for us
171             %
172             %for convolutional layers: we only need to upsample the
173             %sensitivities from the subsampling layer
174 
175             %network error
176             error = net.output-target;
177 
178             %initialize our sens and grad cell arrays
179             net.sens=cell(size(net.layers));
```

```
180             net.grad=cell(size(net.layers));
181
182             %output layer sensitivity and gradient
183             net.sens{end}=net.fp(net.layers{end}(:)).*error(:);
184             net.grad{end}=net.sens{end}*net.layers{end-1}(:).';
185
186
187             %all fully connected layers
188             %we simply use standard backprop
189             for l=length(net.layers)-1:-1:net.convN+1
190                 net.backpropFull(l);
191             end
192
193             %for the last subsampling layer, we can also use normal bprop
194             l=net.convN;
195             sen=net.weights{l+1}.'*net.sens{l+1};
196             %but now we reshape it, since thats what the convolutional layer
197             %expects
198             sl=size(net.layers{l});
199             net.sens{l}=reshape(sen, sl(1), sl(2), sl(3));
200
201             %all other convolutional and subsampling layers
202             for l=net.convN-1:-1:1
203                 if mod(l,2)==1  %if its a convolutional layer
204                     net.backpropConvolution(l);
205                 else %otherwise its a subsampling layer
206                     net.backpropSubSampling(l)
207                 end
208             end
209
210             %now we update the weights and biases
211             net.updateWeights();
212
213
214         end  %backward
215
216         function backpropConvolution(net, l)
217             %performs backprop on layer l, a convolutional layer. we simply need
218             %to upsample the sensitivites from the subsampling layer that
219             %follows it and then calculate the gradient
220
221             %sensitivity calculation
222             ss=upsamp(net.sens{l+1});
223             sl=size(net.layers{l});
224             %and make sure its the right size
225             %(crop the last row and col if necessary)
226             net.sens{l}=net.fp(net.layers{l}).*ss(1:sl(1), 1:sl(2), :);
227
228             %now we find our gradients
229             %get the right input
```

```
230             if l==1
231                 linput=net.input;
232             else
233                 linput=net.layers{l-1};
234             end
235 
236             net.grad{l}=zeros(5,5,size(linput,3),size(net.layers{l},3));
237 
238             for m=1:size(net.layers{l}, 3)  %for each feature map
239                 for im=1:size(linput,3)  %for each input feature map
240                     %use a convolution to calculate gradient
241                     c=rot180(conv2(linput(:,:,im), rot180(net.sens{l}(:,:,m)), 'valid'));
242                     if im==1
243                         net.grad{l}(:,:,im,m)=c;
244                     else
245                         net.grad{l}(:,:,im,m)=net.grad{l}(:,:,im,m)+c;
246                     end
247                 end
248             end
249         end
250 
251         function backpropSubSampling(net, l)
252             %performs backprop on layer l, a subsampling layer
253             %we use a fancy convolution to make getting the sensitivities easy
254             %we dont need to calculate any gradients since this simple
255             %implementation has no trainable parameters yet for this layer
256 
257             for m=1:size(net.layers{l},3)  %for each feature map
258                 %for each feature map in the next convolutional layer
259                 for om=1:size(net.layers{l+1},3)
260                     %we get the weighted sensitivities using a convolution
261                     cs=net.sens{l+1}(:,:,om);
262                     w=net.weights{l+1}(:,:,m,om);
263                     ws=conv2(cs, rot180(w), 'full');
264                     if om==1
265                         ms=ws;
266                     else
267                         ms=ms+ws;
268                     end
269                 end
270                 net.sens{l}(:,:,m)=ms;
271             end
272         end
273 
274         function backpropFull(net, l)
275             %performs backprop on layer l, a fully connected layer
276             %this is standard backprop
277 
278             net.sens{l}=net.fp(net.layers{l}(:)).*(net.weights{l+1}.'*net.sens{l+1});
279             net.grad{l}=net.sens{l}*net.layers{l-1}(:).';
```

```
280         end
281
282         function updateWeights(net)
283             %updates all weights and biases using the calculated gradients
284
285             %convolutional and subsampling layers
286             for l=1:net.convN
287                 if mod(l,2)==1  %if its a convolution layer
288                     net.weights{l}=net.weights{l}-net.rates(l)*net.grad{l};
289                     gs=sum(sum(net.sens{l}));
290                     net.biases{l}=net.biases{l}-net.rates(l)*gs(:);
291                 else %if its a subsampling layer
292                     %dont do anything, no parameters yet
293                 end
294             end
295             %fully connected layers
296             for l=net.convN+1:length(net.grad)
297                 net.weights{l}=net.weights{l}-net.rates(l)*net.grad{l};
298                 net.biases{l}=net.biases{l}-net.rates(l)*net.sens{l};
299             end
300         end
301
302     end  %methods
303
304 end  %class
305
306 function Y=rot180(X)
307     %rotates X by 180 deg
308
309     Y=X(end:-1:1, end:-1:1);
310 end
311
312 function Y=upsamp(X)
313     %up-samples X to be twice as large
314     %
315     %this is the simple implementation:
316     %
317     %     s=size(X);
318     %     Y=zeros(s(1)*2, s(2)*2, s(3));
319     %     for i=1:s(3)
320     %         Y(:,:,i)=kron(X(:,:,i), ones(2));
321     %     end
322     %
323     %but its much more efficient to reshape our 3d matrix into a 2d one,
324     %upsample that, then reshape it back to 3d
325
326     s=size(X);
327     X=reshape(X, s(1), s(2)*s(3));
328     Y=kron(X, ones(2));
329     Y=reshape(Y, s(1)*2, s(2)*2, s(3));
```

```
330 end
```

D.1.2 RBM

```
1  classdef RBM < handle
2      %RBM restricted boltzmann machine class
3      %
4      %Usage:
5      % net=RBM(vdim, hdim) %create a network of the right dimensions
6      % %break data up into mini-batches
7      % RBMtrainer(rbm, minibatches, epochs, rate)
8      %
9      % For more information on RBM best practices, see A Practical Guide to
10     % Training Restricted Boltzmann Machines by G. Hinton. Some recommendations
11     % from version 1.0 of that document were used in this implementation.
12
13     properties
14         vdim                    %number of visible units
15         hdim                    %number of hidden units
16         weights                 %weights
17         biasH                   %hidden layer biases
18         biasV                   %visible layer biases
19         hidden                  %hidden layer states
20         visible                 %visible layer states
21         deltaw                  %weight update values
22         momentum=0.5            %momentum coefficient
23                                 %  Hinton recommends starting with .5, then
24                                 %  increasing to .9 as training progresses
25         weightCost=0.0001       %weight decay coefficient
26                                 %  .0001 is the value Hinton recommends trying
27     end
28
29     methods
30         function net=RBM(vdim, hdim)
31             %RBM constructor
32             %   initializes weights to small random values, biases and deltaw 0
33             %   takes dimensions as input
34
35             net.vdim=vdim;
36             net.hdim=hdim;
37
38             net.weights=random('normal', 0, 0.01, vdim, hdim);
39             net.biasV=zeros(1, vdim);
40             net.biasH=zeros(1, hdim);
41             net.deltaw=zeros(size(net.weights));
42         end
43
44         function p=sampleHidden(net)
45             %sampleHidden
46             %  updates the hidden layer based on the visible layer
47             %  returns the probabilities that were calculated for sampling
```

```
48
49              raw= net.visible * net.weights + net.biasH;
50              p=1./(1+exp(-raw));
51              samples=rand(size(raw));
52              sampledOutput=(p>samples)+0; %logical types tend to break things
53              net.hidden=sampledOutput;
54          end
55
56          function p=sampleVisible(net)
57              %samplesVisible
58              %  updates the visible layer based on the hidden layer
59              %  returns the probabilities that were calculated for sampling
60              raw= net.hidden * net.weights.' + net.biasV;
61              p=1./(1+exp(-raw));
62              samples=rand(size(raw));
63              sampledOutput=(p>samples)+0;
64              net.visible=sampledOutput;
65          end
66
67          function avgError=train(net, inputs, rate)
68              %train
69              %  trains the network on one minibatch
70              %  takes as input a cell array of an  arry of inputs, and the
71              %  learning rate returns the average mean squared reconstruction
72              %  error (note: this "error" is not as useful as error outputs when
73              %  training MLPs. See the Hinton document for more info.)
74
75              %initialize everything to zero
76              avgError=0;
77              vi_hj=zeros(size(net.weights));
78              vihat_hjhat=zeros(size(net.weights));
79              vi=zeros(size(net.biasV));
80              hj=zeros(size(net.biasH));
81              vihat=zeros(size(net.biasV));
82              hjhat=zeros(size(net.biasH));
83
84              %We do one iteration of contrastive divergence training (CD1) for
85              %each input. In the positive phase we clamp the visible and then
86              %sample the hidden layer.  In the negative phase we sample the
87              %visible layer, and then sample the hidden layer.  Then we use the
88              %CD update equations to update weights based on averages over the
89              %minibatch.
90              for input=inputs.'
91                  %positive phase
92                  %clamp visible, sample hidden
93                  net.visible=input.'+0; %make sure its scalar not logical
94                  hiddenProb=net.sampleHidden();
95                  %we'll average over these in the weight update
96                  vi=vi+net.visible;
97                  hj=hj+net.hidden;
```

```
98              vi_hj=vi_hj+net.visible.'*hiddenProb;
99
100             err=net.visible; %first part of the error calc
101
102             %negative phase
103             %sample visible, sample hidden
104             visibleProb=net.sampleVisible();
105             hiddenProb=net.sampleHidden();
106             %we'll average these in the weight update step
107             vihat=vihat+net.visible;
108             hjhat=hjhat+net.hidden;
109             vihat_hjhat=vihat_hjhat+net.visible.'*hiddenProb;
110
111             %now cacluate the reconstruction error and add it to our
112             %running total
113             err=mse(err-net.visible);
114             avgError=avgError+err;
115         end
116
117         %now adjust the weights using the CD1 update equations.  we use the
118         %probabilities in some places rather than the binary states to
119         %decrease noise and speed training.  these equations were actually
120         %based on an old matlab implementation (by Hinton), and might not
121         %correspond exactly to the current recommendations of when to use
122         %the states vs the probabilities.
123         n=length(inputs);
124         net.deltaw = net.momentum*net.deltaw + ...
125             rate*(vi_hj - vihat_hjhat)/n - net.weightCost*net.weights;
126         net.weights=net.weights+net.deltaw;
127         net.biasH=net.biasH + rate* (hj/n-hjhat/n);
128         net.biasV=net.biasV + rate* (vi/n-vihat/n);
129
130         %average the reconstruction error
131         avgError=avgError/n;
132
133     end
134
135     function output=present(net, input, iters)
136         %present
137         %  samples the input for iters Gibbs sampling iterations
138         %  and returns the final visible states (the output)
139         net.visible=input(:)';
140         for i=1:iters
141             net.sampleHidden();
142             net.sampleVisible();
143         end
144         output=net.visible;
145     end
146
147     function output=presentDeterministic(net, input, iters)
```

```
148             %presentDeterministic
149             %  samples the input for iters Gibbs sampling iterations
150             %  deterministically and returns the final probabilities
151             net.visible=input(:)';
152             for i=1:iters
153                 net.sampleHidden();
154                 p=net.sampleVisible();
155             end
156             output=p;
157         end
158     end
159 end
```

D.1.3 DBN

```
1  classdef DBN < handle
2      %DBN deep belief network class
3      %
4      % Usage:
5      %   dbn=DBN([inputDim], [layer1Dim layer2Dim ...]);
6      %   %split data into minibatches
7      %   dbnError=DBNtrainer(dbn, minibatches, dbnEpochs, dbnRate);
8      %   testingStochasticOut=dbn.present(testingInput, iters)
9      %   testingDeterministicOut=dbn.present(testingInput, iters)
10     %   %iters 0 simply infers and then generates, with no gibbs sampling
11
12     properties
13         vdim
14         hdims=[]
15         layers={}
16     end
17
18     methods
19         function dbn=DBN(vdim, hdims)
20             %DBN constructor
21             %creates the specified RBMs
22             dbn.vdim=vdim;
23             dbn.hdims=hdims;
24
25             for l=1:length(hdims)
26                 if l>1 %if not first layer, vdim is hdim of previous
27                     vdim=hdims(l-1);
28                 end
29                 dbn.layers{l}=RBM(vdim, hdims(l));
30             end
31
32         end %DBN
33
34         function infer(dbn, input)
35             %performs an upwards pass, sampling all hidden layers
36             for l=1:length(dbn.layers)
```

```
37              rbm=dbn.layers{l};
38              if l==1
39                  rbm.visible=input;
40              else
41                  rbm.visible=dbn.layers{l-1}.hidden;
42              end
43              rbm.sampleHidden();
44          end
45      end %infer
46
47      function inferDeterministic(dbn, input)
48          %infers based on input, but uses probabilities rather than
49          %stochastic states
50          for l=1:length(dbn.layers)
51              rbm=dbn.layers{l};
52              if l==1
53                  rbm.visible=input;
54              else
55                  rbm.visible=dbn.layers{l-1}.hidden;
56              end
57              rbm.hidden=rbm.sampleHidden();
58          end
59      end
60
61      function out=generate(dbn)
62          %downward pass, sampling all visible layers
63          for l=length(dbn.layers):-1:1
64              rbm=dbn.layers{l};
65              if l<length(dbn.layers)
66                  rbm.hidden=dbn.layers{l+1}.visible;
67              end
68              rbm.sampleVisible();
69          end
70
71          out=dbn.layers{1}.visible;
72
73      end %generate
74
75      function out=generateDeterministic(dbn)
76          %generates, but uses probabilities instead of stocahstic states
77          for l=length(dbn.layers):-1:1
78              rbm=dbn.layers{l};
79              if l<length(dbn.layers)
80                  rbm.hidden=dbn.layers{l+1}.visible;
81              end
82              rbm.visible=rbm.sampleVisible();
83          end
84
85          out=dbn.layers{1}.visible;
86
```

```
87          end %generate
88
89          function out=present(dbn, input, iters)
90              %presents the network with an input pattern, does an upwards
91              %inference pass, performs iters iterations of gibbs sampling in
92              %last layer, then performs a downwards generating pass
93              dbn.infer(input);
94              rbm=dbn.layers{end};
95              for ii=1:iters
96                  rbm.sampleVisible();
97                  rbm.sampleHidden();
98              end
99              out=dbn.generate();
100
101         end
102
103         function out=presentDeterministic(dbn, input, iters)
104             %presents a pattern to the network, but all inference and generation
105             %is deterministic
106             dbn.inferDeterministic(input);
107             rbm=dbn.layers{end};
108             for ii=1:iters
109                 rbm.visible=rbm.sampleVisible();
110                 rbm.hidden=rbm.sampleHidden();
111             end
112             out=dbn.generateDeterministic();
113         end
114
115     end %methods
116 end %class
```

D.2 Trainers

D.2.1 RBM

```
1  function error=RBMtrainer(rbm, minibatches, epochs, rate)
2      %trains the RBM
3      % prints statistics every 1% towards completion
4      % returns the error plot
5      % after 25 epochs, increases momentum to 0.9
6      %
7      %inputs:
8      %  rbm - the network to train
9      %  minibatches - a cell array of the minibatch matrices
10     %  epochs - the number of epochs to train for
11     %  rate - the learning rate
12
13     numbatches = length(minibatches);  %the number of batches
14
15     %keep track of the first 100 weights and the reconstruction error
```

```
16      %to plot at the end of training
17      %TODO: could make these plots update as training progresses
18      weights=zeros(epochs*numbatches, 100);
19      error=zeros(epochs*numbatches,1);
20
21      for e=1:epochs
22          if e==25
23              rbm.momentum=0.9;
24              fprintf('increasing momentum to 0.9\n')
25          end
26          if mod(e, epochs/100)<1
27              fprintf('%2d%% complete.\n', round(e/epochs*100));
28          end
29          for b=randperm(numbatches)  %we randomize the order of the batches
30              %train the network
31              err=rbm.train(minibatches{b}, rate);
32
33              %keep the statistics
34              dataindex= (e-1)*numbatches+b;
35              weights(dataindex,:)=rbm.weights(1:100).';
36              error(dataindex)=err;
37
38              %print statistics every 1% towards completion
39              if mod(e, epochs/100)<1
40                  fprintf('\t batch %d reconstruction error: %f\n', ...
41                      b, err)
42              end
43          end
44      end
45
46      %figure(1),plot(weights), title('weights')
47      %figure(2),plot(error), title('reconstruction error')
48  end
```

D.2.2 DBN

```
1   function errors=DBNtrainer(dbn, minibatches, epochs, rate)
2       %trains the dbn
3       %
4       %dbn training is done by greedily training each layer as an RBM, bottom up,
5       %using the hidden state of lower layers as the visible state of the layer
6       %being traind.  each rbm is trained for the specified number of epochs using
7       %the RBMtrainer.
8
9       numbatches=length(minibatches);
10
11      %we train layer by layer, bottom up
12      for l=1:length(dbn.layers)
13          fprintf('training layer %d.\n', l);
14
15          error=zeros(epochs,1);
```

```
16
17          rbm=dbn.layers{l};
18
19          %train the rbm
20          %if we're somehwere in the middle of the dbn, we clamp our visible
21          %units to an inferred value, not the actual input.  let's
22          %precompute what those are going to be (this is probably not the best
23          %way to do this, but it is the easiest)
24          if l==1
25              %if its the first layer, nothing to do
26              inputbatches=minibatches;
27          else
28              %otherwise, run the lower layers on all the inputs
29              for ii=1:length(minibatches) %for each minibatch
30                  minibatch=minibatches{ii};
31                  inputbatch=zeros(size(minibatch));
32                  for jj=1:size(minibatch, 1) %for each input
33                      %infer the value, then grab the layer right below
34                      dbn.infer(minibatch(jj,:));
35                      inputbatch(jj,:)=dbn.layers{l-1}.hidden;
36                  end
37                  inputbatches{ii}=inputbatch;
38              end
39
40              %also initialize our weights to be the same as those in the lower
41              %layer if the size is the same (gives a slight speed up to training)
42              try
43                  dbn.layers{l}.weights=dbn.layers{l-1}.weights;
44                  fprintf('initializing weights to be the same\n');
45              catch err
46                  fprintf('different dimensions, leaving weights random.\n')
47              end
48          end
49          %now we have our inputs, lets just run the RBMtrainer for this layer
50          error=RBMtrainer(rbm, inputbatches, epochs, rate);
51
52          errors{l}=error;
53      end
54  end
```

D.3 Test

```
1  function info=dbncnn(training, testing, trainingTargets, testingTargets, ...
2          usedbn, traincnnwithdbn)
3      %dbncnn(training, testing, trainingTargets, testingTargets, usedbn, trainwithdbn)
4      %  trains the dbn, then cnn, then tests.  returns info about training and
5      %  performance
6      %  inputs:
7      %    training          array of training inputs
8      %    testing           array of testing inputs
```

```
%     trainingTargets  array of training targets
%     testingTargets   array of testing targets
%     usedbn           boolean for whether or not to use dbn
%     trainwithdbn     boolean for whether or not to train with the dbn
%   returns:
%     info={pr dbn cnn dbnError cnnError {orig miss}};

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Settings
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%misc settings
imgScaleFactor=1;            % factor to downsample by

%pixelReducer settings
prTolerance=0.1;             % what percent of pixels can be different at the
                             % location, but the location is uninteresting

%dbn training settings
dbnEpochs=750;               % how many epochs to train for
dbnLayers=2;                 % how many layers in the dbn
dbnRate=0.1;                 % learning rate
dbnBatchSize=100;             % size of the minibatches

%cnn training settings
cnnEpochs=100;
cnnConvLayers=[5 25];        % feature maps in each convolution layer
cnnFullLayers=[2];           % neurons in each fully connected layer
cnnRates=[10^-3 10^-3 10^-2];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trainingN=size(training, 3);
testingN=size(testing, 3);
fprintf('beginning run with %d training and %d testing images.\n',...
    trainingN, testingN);

%down sample our images
fprintf('downsampling images by a factor of %d...', imgScaleFactor);
training=training(1:imgScaleFactor:end, 1:imgScaleFactor:end, :);
testing=testing(1:imgScaleFactor:end, 1:imgScaleFactor:end, :);

dim=size(training(:,:,1));
fprintf('images are %dx%d.\n', dim(2), dim(1));

%initialize pixel reducer
fprintf('analyzing location of significant pixels...\n');
pr=pixelReducer(dim);
```

```
    pr.analyze(training, prTolerance*testingN, (1-prTolerance)*testingN)

    %reduce inputs
    for ii=1:trainingN
        dbnInput(ii,:)=pr.preprocess(training(:,:,ii));
    end
    dbnN=size(dbnInput,2);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % DBN Training
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if (usedbn)
        %create the dbn
        fprintf('creating a dbn with %d layers, each with %d nodes.\n',...
            dbnLayers, dbnN);

        dbnDims=repmat([dbnN], [1 dbnLayers]);

        dbn=DBN(dbnN, dbnDims);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %fprintf('loading previously trained dbn.\n')
        %load initial
        %dbn=usingAndTraining{2};
        %for l=1:length(dbn.layers)
        %    dbn.layers{l}.momentum=0.9;
        %end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %split data into minibatches
        minibatches={};
        batches=ceil(trainingN/dbnBatchSize);
        for ii=1:trainingN
            batch=mod(ii,batches)+1;
            index=ceil(ii/batches); %ceil since matlab indexing starts at 1
            minibatches{batch}(index,:)=dbnInput(ii,:);
        end
        fprintf('split data into %d mini-batches of up to %d cases each.\n',...
            batches, dbnBatchSize);

        %train the dbn
        fprintf('training the %d dbn layers for %d epochs each.\n', ...
            dbnLayers, dbnEpochs)
        dbnError=DBNtrainer(dbn, minibatches, dbnEpochs, dbnRate);
        fprintf('dbn training completed.\n')

    else %if
        dbn=0;
        dbnError=0;
```

```
    end


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % CNN Training
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %use the right inputs
    if (~usedbn | ~traincnnwithdbn)
        %use original as inputs
        %for ii=1:trainingN
        %     dbnout=dbnInput(ii,:);
        %     cnnTraining(:,:,ii)=pr.postprocess(dbnout);
        %end
        %fprintf('using original inputs as cnn training inputs (after pr).\n');
        cnnTraining=training;
        fprintf('using original inputs as cnn training inputs.\n');
    else
        %use dbn outputs as inputs
        for ii=1:trainingN
            dbn.inferDeterministic(dbnInput(ii,:));
            dbnout=dbn.generateDeterministic();
            cnnTraining(:,:,ii)=pr.postprocess(dbnout);
        end
        fprintf('using dbn outputs as cnn training inputs.\n');
    end

    %create cnn
    cnn=CNN(dim, cnnConvLayers, cnnFullLayers, cnnRates);

    %train the cnn
    fprintf('training the cnn for %d epochs.\n', cnnEpochs);
    for e=1:cnnEpochs
        err=0;
        for ii=randperm(trainingN)
            cnn.input=cnnTraining(:,:,ii);
            target=trainingTargets(ii,:);

            cnn.forward();
            cnn.backward(target');

            err=err+mse(cnn.output-target');
        end
        cnnError(e)=err/trainingN;
        if mod(e, cnnEpochs/100)<1
            fprintf('%2d%% done. \t MSE: %f\n', 100*e/cnnEpochs, cnnError(e));
        end
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Testing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fprintf('beginning testing')
if usedbn
    fprintf(' using dbn output as cnn input.\n')
    %preprocess testing images
    for ii=1:testingN
        dbnTesting(ii,:)=pr.preprocess(testing(:,:,ii));
    end
    %use dbn outputs as inputs
    for ii=1:testingN
        dbn.inferDeterministic(dbnTesting(ii,:));
        dbnout=dbn.generateDeterministic();
        cnnTesting(:,:,ii)=pr.postprocess(dbnout);
    end
else
    %fprintf(' using raw input as cnn input (after pr).\n')
    %for ii=1:testingN
    %    dbnTesting=pr.preprocess(testing(:,:,ii));
    %    cnnTesting(:,:,ii)=pr.postprocess(dbnTesting);
    %end
    fprintf(' using raw input as cnn input.\n')
    cnnTesting=testing;
end

%test
missclass=0;
err=0;
orig={};
miss={};
for ii=1:testingN
    cnn.input=cnnTesting(:,:,ii);
    target=testingTargets(ii,:);

    cnn.forward();

    err=err+mse(cnn.output-target');
    [dummy, index]=max(cnn.output);

    if target(index)~=1
        missclass=missclass+1;
        orig{missclass}=testing(:,:,ii);
        miss{missclass}=cnn.input;
    end

    %figure(2); imshow(cnn.input);
    %pause;
end
```

```
209     fprintf('for unseen data: \n avg mse: %f, missclassifications: %d of %d\n', ...
210         err/testingN, missclass, testingN);
211
212     %return all the info
213     info={pr dbn cnn dbnError cnnError {orig miss}};
214
215 end
```

Bibliography

[1] David Ackley, Geoffrey Hinton, and Terrence Sejnowski. A learning algorithm for boltzmann machines. pages 635–649, 1988.

[2] Amani Al-Ajlan and Ali El-Zaart. Image segmentation using minimum cross-entropy thresholding. In *Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics*, San Antonio, TX, USA, October 2009. IEEE.

[3] D. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *AAAI94 workshop on knowledge discovery in databases*, pages 359–370, 1994.

[4] Dimitrios Charalampidis. Image processing applications - edge detection.

[5] Dimitrios Charalampidis. Introduction to image processing.

[6] HD Cheng, YH Chen, and XH Jiang. Thresholding using two-dimensional histogram and fuzzy entropy principle. *Image Processing, IEEE Transactions on*, 9(4):732–735, 2000.

[7] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati. Detecting moving objects, ghosts, and shadows in video streams. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1337–1342, 2003.

[8] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2006.

[9] Marco Antonio Garcia de Carvalho and Roberto marcondes Cesar Junior. Image segmentation using component tree and normalized cut. In *SIBGRAPI - Conference on Graphics, Patters, and Images*, number 23, 2010.

[10] Paul Fieguth. *Statistical Image Processing and Multidemensional Modeling*. Information Science and Statistics. Springer, 2011.

[11] Chonghui Guo and Hong Li. Mulitlevel thresholding method for image segmentation based on adaptive particle swarm optimization algortihm. *AI*, pages 654–658, 2007.

[12] Geoffrey Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[13] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[14] T. Huang, G. Yang, and G. Tang. A fast two-dimensional median filtering algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 27(1):13–18, 1979.

[15] M.H. Hung, J.S. Pan, and C.H. Hsieh. Speed up temporal median filter for background subtraction. In *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conference on*, pages 297–300. IEEE, 2010.

[16] Paolo Ciaccia Ilaria Bartolini. Warp: Accurate retrieval of shapes using phase of fourier descriptors and time warping distance. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 27(1), January 2005.

[17] A. Kaufmann and DL Swanson. *Introduction to the theory of fuzzy subsets*, volume 1. Academic Press New York, 1975.

[18] Hannu Kauppinen, Tapio Seppanen, and Matti Pietikainen. An experimental comparison of autoregressive and fourier-based descriptors in 2d shape classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(2):201–207, February 1995.

[19] E. Keogh and C.A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[20] J. Kittler and J. Illingworth. Minimum error thresholding. *Pattern Recognition*, 19(1):41–47, 1986.

[21] S Lawrence, CL Giles, Ah Tsoi, and AD Back. Face recognition: a convolutional neural-network approach. *Neural Networks, IEEE Transactions on*, 8(1):98–113, 2002.

[22] Y Lecun, L Bottou, Y Bengio, and P Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Ping-Sung Liao, Tse-Sheng Chen, and Pau-Choo Chung. A fast algorithm for multilevel thresholding. *Journal of Information Science and Engineering*, (17):713–727, 2001.

[24] BPL Lo and SA Velastin. Automatic congestion detection system for underground platforms. In *Intelligent Multimedia, Video and Speech Processing, 2001. Proceedings of 2001 International Symposium on*, pages 158–161. IEEE, 2001.

[25] Rolf Lakamper Ulrich Eckhardt Longin Jan Latecki. Shape descriptors for non-rigid shapes with a single closed contour. In *IEEE Conference on Computer Vision and Pattern Recognition, 2000. Procedings.*, volume 1, pages 424–429. IEEE, June 2000.

[26] N.V. Lopes, PA Mogadouro do Couto, H. Bustince, and P. Melo-Pinto. Automatic histogram threshold using fuzzy measures. *Image Processing, IEEE Transactions on*, 19(1):199–204, 2010.

[27] C.A. Murthy and S.K. Pal. Fuzzy thresholding: mathematical framework, bound functions and weighted moving average technique. *Pattern Recognition Letters*, 11(3):197–206, 1990.

[28] Y. Nakagawa and A. Rosenfeld. Some experiments on variable thresholding. *Pattern Recognition*, 11(3):191–204, 1979.

[29] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions On Systems, Man, and Cybernetics*, SMC-9(1):62–66, January 1979.

[30] N.R. Pal. On minimum cross-entropy thresholding. *Pattern Recognition*, 29(4):575–580, 1996.

[31] N. Papamarkos and B. Gatos. A new approach for multilevel threshold selection. *Graphical models and Image Processing*, 56(5):357–370, September 1994.

[32] Massimo Piccardi. Background subtraction techniques: a review. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 4, pages 3099–3104. IEEE, 2004.

[33] D. Rafiei and A.O. Mendelzon. Efficient retrieval of similar shapes. *The VLDB journal*, 11(1):17–27, 2002.

[34] K. Ramar, S. Arumugam, SN Sivanandam, L. Ganesan, and D. Manimegalai. Quantitative fuzzy measures for threshold selection. *Pattern Recognition Letters*, 21(1):1–7, 2000.

[35] N. Ramesh, J.-H Yoo, and I. K. Sethi. Thresholding based on histogram approximation. *IEE Proc.-Vis Image Signal Processing*, 142(5), October 1995.

[36] R. Rosipal, M. Girolami, L.J. Trejo, and A. Cichocki. Kernel pca for feature extraction and de-noising in nonlinear regression. *Neural Computing & Applications*, 10(3):231–243, 2001.

[37] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 26(1):43–49, 1978.

[38] Mehmet Sezgin and Bulent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, 13(1):146–165, January 2004.

[39] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[40] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 22(8), August 2000.

[41] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. *Document Analysis and Recognition, International Conference on*, 2:958, 2003.

[42] Wenbing Tao, Hai Jin, Yimin Zhang, Liman Liu, and Desheng Wang. Image thresholding using graph cuts. *IEEE Transactions On Systems, Man, and Cybernetics - Part A: Systems and Humans*, 38(5), September 2008.

[43] X. Wang, X. Ding, and C. Liu. Gabor filters-based feature extraction for character recognition. *Pattern recognition*, 38(3):369–379, 2005.

[44] Zhihua Wang. *Time Series Matching: a Multi-filter Approach*. PhD thesis, New York University, January 2006.

[45] C.R. Wren, A. Azarbayejani, T. Darrell, and A.P. Pentland. Pfinder: Real-time tracking of the human body. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(7):780–785, 1997.

[46] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, pages 1101–1113, 1993.

[47] Yang Xiao, Zhiguo Cao, and Tianxu Zhang. Entropic thresholding based on gray-level spatial correlation histogram. *IEEE*, 2008.

[48] S. D. Yanowitz and A. M. Bruckstein. A new method for image segmentation. *Computer Vision, Graphics, and Image Processing*, (46):82–95, 1989.

[49] J.Y. Yeh, H.R. Ke, W.P. Yang, I. Meng, et al. Text summarization using a trainable summarizer and latent semantic analysis* 1. *Information processing & management*, 41(1):75–95, 2005.

[50] Dengsheng Zhang and Guojun Lu. Shape retrieval using fourier descriptors. In *In Proceedings of 2nd IEEE Pacific Rim Conference on Multimedia*, pages 1–9. Springer, 2001.

Index