A NEAR REAL-TIME, HIGHLY SCALABLE, PARALLEL AND DISTRIBUTED ADAPTIVE OBJECT DETECTION AND RE-TRAINING FRAMEWORK BASED ON THE ADABOOST ALGORITHM

By: Munther Abualkibash

Under the Supervision of Dr. Ausif Mahmood

DISSERTATION

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

AND ENGINEERING

THE SCHOOL OF ENGINEERING

UNIVERSITY OF BRIDGEPORT

CONNECTICUT

April 2015

A NEAR REAL-TIME, HIGHLY SCALABLE, PARALLEL AND DISTRIBUTED ADAPTIVE OBJECT DETECTION AND RE-TRAINING FRAMEWORK BASED ON THE ADABOOST ALGORITHM

Approvals

Committee Members

Name	Signature	Date
Dr. Ausif Mahmood		4-22-2015
Dr. Navarun Gupta		5/6/2015
Dr. Prabir Patra		04/22/15
Dr. Miad Faezipour		04,22,2015
Dr. Saeid Moslehpour		APR 30, 15

Ph.D. Program Coordinator

Name	Signature	Date
Dr. Khaled M. Elleithy		5/6/15

Chairman, Computer Science and Engineering Department

Name	Signature	Date
Dr. Ausif Mahmood		4-22-2015

Dean, School of Engineering

Name	Signature	Date
Dr. Tarek M. Sobh		5/9/2015

A NEAR REAL-TIME, HIGHLY SCALABLE, PARALLEL AND DISTRIBUTED ADAPTIVE OBJECT DETECTION AND RE-TRAINING FRAMEWORK BASED ON THE ADABOOST ALGORITHM

© Copyright 2015 Munther Abualkibash

A NEAR REAL-TIME, HIGHLY SCALABLE, PARALLEL AND DISTRIBUTED ADAPTIVE OBJECT DETECTION AND RE-TRAINING FRAMEWORK BASED ON THE ADABOOST ALGORITHM

ABSTRACT

Object detection, such as face detection using supervised learning, often requires extensive training for the computer, which results in high execution times. If the trained system needs re-training in order to accommodate a missed detection, waiting several hours or days before the system is ready may be unacceptable in practical implementations. This dissertation presents a generalized object detection framework whereby the system can efficiently adapt to misclassified data and be re-trained within a few minutes. Our developed methodology is based on the popular AdaBoost algorithm for object detection. AdaBoost functions by iteratively selecting the best among weak classifiers, and then combining several weak classifiers in order to obtain a stronger classifier. Even though AdaBoost has proven to be very effective, its learning execution time can be high depending upon the application. For example, in face detection, learning can take several days. In our dissertation, we present two techniques that contribute to reducing to the learning execution time within the AdaBoost algorithm. Our first technique utilizes a highly parallel and distributed AdaBoost algorithm that

exploits the multiple cores in a CPU via lightweight threads. In addition, our technique uses multiple machines in a web service similar to a map-reduce architecture in order to achieve a high scalability, which results in a training execution time of a few minutes rather than several days. Our second technique is a methodology to create an optimal training subset to further reduce the training execution time. We obtained this subset through a novel score-keeping of the weight distribution within the AdaBoost algorithm, and then removed the images that had a minimal effect on the overall trained classifier. Finally, we incorporated our parallel and distributed AdaBoost algorithm, along with the optimized training subset, into a generalized object detection framework that efficiently adapts and makes corrections when it encounters misclassified data. We demonstrated the usefulness of our adaptive framework by providing detailed testing on face and car detection, and explained how our framework applies to developing any other object detection task.

ACKNOWLEDGEMENTS

In the name of God, the Most Gracious, the Most Merciful, I wholly devote my thanks to God who has helped me all the way to complete this work successfully. I owe a debt of gratitude to my loving family: my parents, my wife, Sawsan, and my daughters, Sarah and Lena, for all of their time waiting, and for their understanding and encouragement.

I am honored to have had the opportunity to work under the supervision of Dr. Ausif Mahmood. Throughout the course of this dissertation, Dr. Mahmood provided invaluable guidance, and worked tirelessly to help me craft this dissertation through several stages. His encouragement and support was vital to the completion of this dissertation.

I would like to express special thanks to Dr. Khaled Elleithy for his support, advice, and encouragement.

I would also like to express my gratitude to committee members Dr. Prabir Patra, Dr. Navarun Gupta, Dr. Miad Faezipour, and Dr. Saeid Moslehpour for their valuable feedback and discussion regarding this work.

I would like to thank Lydia Iarocci for assistance in proofreading and editing.

Last but not least, I would like to thank all the staff of the School of Engineering for their support, who made my studies at the University of Bridgeport a wonderful and exciting experience.

TABLE OF CONTENTS

ABSTRACT .... **iv**

ACKNOWLEDGEMENTS .... **vi**

LIST OF TABLES .... **ix**

LIST OF FIGURES .... **x**

CHAPTER 1: INTRODUCTION **1**

1.1 Research Problem and Scope 1

1.2 Motivation Behind the Research 4

1.3 Potential Contributions of the Proposed Research 5

CHAPTER 2: LITERATURE SURVEY **7**

2.1 Introduction 7

2.2 The Concept of Boosting 7

2.3 AdaBoost Algorithm used by Viola and Jones 9

2.3.1 Integral Image 9

2.3.2 Extracted Feature Types and Selection 10

2.4 Previous Research on Parallel and Distributed AdaBoost 20

2.5 Previous Research on Training Classifiers on a Small Training Set 26

2.6 Summary 27

CHAPTER 3: RESEARCH PLAN **29**

3.1 Parallel and Distributed Web Services-Based AdaBoost Architecture 29

3.1.1 Parallel Execution 32

3.1.2 Web Services and Parallel Execution on One-level Hierarchy 32

3.1.3 Web Services and Parallel Execution on Two-level Hierarchy 34

3.1.4 Web Services and Parallel Execution on One-level Hierarchy with N Number of Slave Nodes 36
3.2 Extracting Minimum Size Subset that has 100% Detection Rate for Faces and Non-faces 38
3.3 Implementing a Cascade Structure Based on a Small Training Set 41
3.4 A Near Real-time Re-training 47
3.5 Summary 48
CHAPTER 4: RESULTS 49
4.1 Parallel and Distributed AdaBoost Testing Results 49
4.2 Pruning Techniques Testing Results 56
4.2.1 Pruning by Clustering 56
4.2.2 Pruning by Distance and Number of Neighbors 59
4.2.3 Pruning Using Recursive Elimination of Neighbors 60
4.2.4 Results of Pruning Techniques 62
4.2.4.1 Results of Pruning by Clustering 64
4.2.4.2 Results of Pruning by Distance and Number of Neighbors 65
4.2.4.3 Results of Pruning using Recursive Elimination of Neighbors 66
4.3 A Near Real-time Re-training to Fix Results of Missed Objects and False Positives 67
4.3.1 The Proposed Framework Results for Cars 67
4.3.2 The Proposed Framework Results for Faces 74
4.4 Power Analysis 78
4.5 Summary 86
CHAPTER 5: CONCLUSION 87
5.1 Future Directions 88
REFERENCES 89
APPENDIX A: SAMPLES OF IMPLEMENTATION CODE 98

LIST OF TABLES

TABLE 2.1 TOTAL NUMBER OF FEATURES USING 24X24 PIXELS WINDOW SIZE FOR LIENHART AND MAYDT'S FEATURES [52] 13
TABLE 2.2 EXAMPLE OF IMAGES AND LABELS 16
TABLE 2.3 EXAMPLE OF IMAGES, LABELS, AND WEIGHTS 16
TABLE 4.1 COMPARISON OF THE FIRST FOUR APPROACHES USED 50
TABLE 4.2 THE RESULT OF THE PREDICTIVE EQUATION BASED ON THE NUMBER OF NODES ATTACHED TO ONE SUB-MASTER NODE 52
TABLE 4.3 OVERHEAD ON THE ONE-LEVEL NETWORK USING ONE MASTER AND 5 SLAVE NODES 52
TABLE 4.4 OVERHEAD ON THE TWO-LEVEL NETWORK USING ONE MASTER, 5 SUB-MASTER NODES, AND 25 SLAVE NODES 53
TABLE 4.5 COMPARISON OF THE FIFTH APPROACH WITH DIFFERENT NUMBER OF NODES ... 54
TABLE 4.6 EXAMPLE OF THREE IMAGES AFTER PRUNING BASED ON ID 64
TABLE 4.7 EXAMPLE OF THREE IMAGES AFTER PRUNING BASED ON DISTANCE AND NUMBER OF NEIGHBORS 65
TABLE 4.8 EXAMPLE OF THREE IMAGES AFTER PRUNING BASED ON RECURSIVE ELIMINATION OF NEIGHBORS 66
TABLE 4.9 RESULTS OF RE-TRAINING TO DETECT A MISSED CAR 68
TABLE 4.10 RESULTS OF RE-TRAINING TO DELETE FALSE POSITIVES 69
TABLE 4.11 RESULTS OF DETECTING CARS BY THE PROPOSED FRAMEWORK WITHOUT RE-TRAINING 73
TABLE 4.12 RESULTS OF RE-TRAINING TO DETECT A MISSED FACE 74
TABLE 4.13 RESULTS OF RE-TRAINING TO DELETE FALSE POSITIVES 75
TABLE 4.14 RESULTS OF DETECTING FACES BY THE PROPOSED FRAMEWORK WITHOUT RE-TRAINING 78
TABLE 4.15 EFFECT SIZE VALUES 80

LIST OF FIGURES

FIGURE 2.1 THE SUMMATION OF ALL PIXELS ON TOP AND TO THE LEFT OF X,Y IS THE INTEGRAL IMAGE VALUE AT X,Y 9
FIGURE 2.2 CALCULATING THE INTEGRAL IMAGE IN A RECTANGULAR REGION 10
FIGURE 2.3 FIVE RECTANGULAR FEATURES. FIGURE (A) SHOWS TWO RECTANGLE HORIZONTAL AND VERTICAL FEATURES, FIGURE (B) SHOWS THREE RECTANGLE HORIZONTAL AND VERTICAL FEATURES, AND FIGURE (C) SHOWS A FOUR RECTANGLE FEATURE 11
FIGURE 2.4 FEATURES (A, B, C, D, E, F, G, AND H) ARE EXAMPLES OF LINE FEATURES........ 12
FIGURE 2.5 FEATURES (A, B, C, AND D) ARE EXAMPLES OF EDGE FEATURES 13
FIGURE 2.6 FEATURES (A AND B) ARE EXAMPLES OF CENTER-SURROUND FEATURES 13
FIGURE 3.1 ONE-LEVEL HIERARCHY FOR WEB SERVICES AND PARALLEL ADABOOST, BASED ON ONE MASTER AND FIVE SLAVE NODES (TOTAL OF SIX PCS) 30
FIGURE 3.2 TWO-LEVEL HIERARCHY FOR WEB SERVICES AND PARALLEL ADABOOST, BASED ON ONE MASTER, FIVE SUB-MASTER NODES, AND 3 SLAVE NODES FOR EACH SUB-MASTER NODE (TOTAL OF TWENTY ONE PCS) 31
FIGURE 3.3 ONE-LEVEL HIERARCHY FOR WEB SERVICES AND PARALLEL ADABOOST, BASED ON ONE MASTER AND N SLAVE NODES 31
FIGURE 3.4 EXTRACTING AN OPTIMIZED TRAINING SUBSET 39
FIGURE 3.5 EACH FEATURE VALUE IS CALCULATED IN EACH IMAGE FROM THE OPTIMIZED SUBSET OF THE TRAINING SET 40
FIGURE 3.6 FINDING THE MULTIPLIERS FOR THE SUM OF ALPHAS 43
FIGURE 3.7 MULTIPLIER COEFFICIENT FOR EACH STAGE TO ACHIEVE 100% TRUE POSITIVE DETECTION 44
FIGURE 3.8 CLASSIFIER OF TWENTY ONE STAGES 44
FIGURE 3.9 ARCHITECTURE FOR THE RE-TRAINABLE SYSTEM 48
FIGURE 4.1 PARALLEL EXECUTION TIME BASED ON THE TOTAL NUMBER OF SLAVE WORKSTATIONS 50
FIGURE 4.2 REAL AND PREDICTIVE PARALLEL EXECUTION TIME BASED ON TOTAL NUMBER OF SLAVE WORKSTATIONS IN THE LAST LEVEL 51
FIGURE 4.3 PARALLEL EXECUTION TIME BASED ON THE TOTAL NUMBER OF SLAVE WORKSTATIONS USING THE MASTER – N SLAVE NODES APPROACH 54

FIGURE 4.4 REAL AND PREDICTIVE PARALLEL EXECUTION TIME BASED ON THE TOTAL NUMBER OF SLAVE WORKSTATIONS USING THE MASTER – N SLAVE NODES APPROACH 55
FIGURE 4.5 MULTIPLE RECTANGLES ON EACH FACE AND SOME FALSE POSITIVE RECTANGLES 56
FIGURE 4.6 EACH 1-VALUE REPRESENTS A RECTANGLE IN THE REAL IMAGE IN FIGURE 4.5 57
FIGURE 4.7 A UNIQUE ID FOR EACH GROUP OF 1-VALUES 57
FIGURE 4.8 THE 8 NEIGHBORS OF THE CENTER POINT 58
FIGURE 4.9 (A, B, C) EXAMPLE OF THREE IMAGES USED FOR TESTING 63
FIGURE 4.10 (A, B, C) EXAMPLE OF THREE IMAGES AFTER DETECTING THE EXPECTED FACES AND FALSE POSITIVES 63

CHAPTER 1: INTRODUCTION

1.1 Research Problem and Scope

Object detection is an important area of research in computer vision. One of the most popular approaches for object detection is based on combining many weak classifiers together to achieve one strong classifier through a technique called Boosting. A modified version of this technique was developed by Viola and Jones, where a classifier is created by iteratively selecting a best single feature (weak) classifier from a set of a very large number of potential features. If a sequential implementation is used for boosting, it becomes a time consuming process often requiring days of execution time to implement a classifier with a reasonable number of features. Thus, an efficient parallel boosting algorithm is highly desired for object detection [1].

Boosting has been applied to various object detection implementations such as face [2], car [3-5], and airplane detection [6]. Of these, face detection has received considerable attention as it has proven to be a more challenging field. Also, face detection is needed in many practical applications, such as intelligent human-computer interfaces, video conferencing, and face recognition. For face recognition, first, faces need to be detected by finding their locations in the image, then the faces are extracted, before finally performing the recognition [7].

Many approaches have been developed to detect faces in the last couple of decades. Some of these are based on statistical learning methods, e.g., Support Vector Machine [8], Neural network [9], and Bayesian decision rule [10]. Hjelmas and Low [11] and Yang et al. [12] provide surveys of most of the face detection approaches prior to 2002. Based on Yang et al.'s work, face detection approaches can be classified into four major categories: first, the knowledge-based method in which human knowledge is the basis for face determination; second, the feature invariant approach that makes finding the structure of facial features easy, even in difficult conditions such as strong lighting; third is the template matching method, which focuses on the storing of faces for matching purposes; finally, the appearance-based method that trains the computer to detect faces. Of these, the appearance-based approach yields better results but requires more processing. Since the speed of computers is continuously increasing, using appearance-based methods is preferred compared to the other methods. One of the most significant contributions to face detection work in the appearance-based category is from Viola and Jones [2, 13, 14]. The original Viola and Jones algorithm was published in 2001, and since then it has become the most popular approach to face detection because of its high accuracy and real-time detection capabilities.

Another survey in 2010 by C. Zhang and Z. Zhang [15] discusses the new enhancements in face detection since the previously published surveys of [11] and [12] in 2001 and 2002 respectively. Their main focus is on face detection through boosting algorithms. One of the examples of detecting faces through boosting has been done by Viola and Jones [2, 13, 14]. Viola and Jones' algorithm is based on a variation of boosting technique called AdaBoost [16], short for Adaptive Boosting, which employs a

concept called integral image that greatly reduces the number of computations in the detection algorithm. Furthermore, it uses a cascade of classifiers in an effective manner to obtain high real-time face detection accuracy.

Since AdaBoost is resistant to overfitting [17, 18] and Viola and Jones' algorithm [2, 13, 14] has proven itself to be accurate and fast at the same time, the purpose of this work is to focus on improving this approach by developing a framework that can adapt efficiently to false positives and false negatives in object detection in real implementations through a highly parallel and reduced optimized dataset re-training process. Learning from a smaller training dataset has the important advantage of better generalization in addition to reduced re-training execution time. When the system is able to obtain good classification results through a small number of training examples, this itself is strong proof of the system's generalization abilities [19].

In this dissertation, in order to develop an adaptive near real-time and re-trainable object detection framework, a new parallel and distributed implementation of the AdaBoost boosting algorithm has been proposed and developed. The implementation of the proposed parallel AdaBoost not only reduces the execution time, it also requires less memory for the distribution of training samples to processing nodes. The memory requirement is further reduced by requiring fewer training samples. This has a positive effect on the execution time, as the entire training dataset can be easily stored in the main Random Access Memory (RAM), eliminating the disk swapping that usually occurs in a traditional AdaBoost implementation when the standard set of training samples is used, as provided by Viola and Jones. To exploit the parallelism at the multi-core level for this object detection framework, the Task Parallel Library (TPL) has been used. To further

reduce the execution time, the process incorporates distributed processing on a network of workstations through the use of web services. Even though parallel or distributed implementations of AdaBoost have been reported (e.g., [1, 20-37],) none of the existing parallel approaches matches this proposed implementation in terms of its efficiency in its execution time. This is mainly because the parallel approach is multifaceted; that is, it employs distributed processing in terms of web services and its highly parallel multi-core exploitation through TPL, and reduces memory usage through distribution and optimization of the training dataset.

1.2 Motivation Behind the Research

These days, there are many algorithms for object detection; however, none of them have a 100% detection rate. Therefore, better algorithms are needed so that the detection rate can be near 100% and false positives can be reduced. At the same time, it is also challenging to provide a high detection rate in real-time. Usually, training a classifier might take several days to learn to detect objects, and the difficulty in training a base classifier is known to be a computational problem [38, 39].

However, in cases where the system failed to detect a valid object, there should be a methodology where the system can be re-trained instantly so that such false negatives can be avoided. For example, where a face detection system is installed in a site, if an employee has a change in facial appearance because of an injury or changing hair styles, the system might fail to detect their face. In such cases, there is a need for a system that can be re-trained to adapt to changing features in near real-time. As a result, building a

framework able to re-train a classifier to take care of missed examples of an object, as well as to reduce the number of false positive examples, is highly needed.

This research is motivated by the drawbacks and limitations of existing systems, as well as the following reasons:

First, the AdaBoost algorithm [2, 13, 14] is one very famous approach in the object detection field. Second, to make the AdaBoost algorithm more useful, a huge amount of computation power is needed. Third, multi-core processors have become very common and effective in using multiple threads simultaneously, which greatly enhances computational power. Multi-core processors are the best suited to computer vision algorithms because of their high computational power and multi-threading features. Parallel and distributed AdaBoost can be used to speed up the training process in cases where a classifier must be trained in a very short time. For example, supposing there is a need to train a face detector on a client site, the system needs to be customized to the site's specific conditions. In such cases, it is highly desirable that the training phase can be done as quickly as possible.

1.3 Potential Contributions of the Proposed Research

We develop a novel, highly parallel and distributed implementation of the Viola and Jones algorithm that exploits the multiple cores in a CPU via lightweight threads, and also uses multiple machines via web service software architecture, to achieve high scalability. Based on the number of processors available, a nearly linear increase in processing speed is achieved, and the learning of a feature in the AdaBoost algorithm can be accomplished within a few seconds. The training execution time is further reduced by

using a smaller optimized training dataset. The features extracted from this small optimized subset achieve a high detection rate. The significant speedup of the proposed parallel and distributed AdaBoost enables us to extract the optimized subset from the entire set. The optimized subset is yielded by replacing some images with other images based on the weight value assigned by the AdaBoost algorithm, until getting a small subset that can achieve high detection rates. We encapsulate the entire parallel and distributed system into a general object detection framework where the training dataset can be updated as a result of misclassified data, and the system re-trained within a few minutes. Therefore, this general object detection framework is considered an adaptive framework such that a smaller optimized training subset is used to yield high detection rates while further reducing the re-training execution time. We demonstrate the usefulness of our adaptive framework on face and car detection. The framework developed here is applicable to any object detection task.

CHAPTER 2: LITERATURE SURVEY

2.1 Introduction

There have been many attempts to speed up the AdaBoost algorithm. The following are two main approaches to speed up AdaBoost: (1) reducing the number of data points that participate in training the base learners or (2) reducing the search area by using only a small set of features [1]. In order to speed up AdaBoost more quickly, there is a way to use both approaches together: by using less data and fewer features, a significant reduction in computation time can be observed.

2.2 The Concept of Boosting

Boosting is a well-known ensemble learning method based on the concept that a strong classifier will be produced if repetitive enhancements are applied to any weak base-learner. The concept of boosting evolved from the theoretical question of whether one strong classifier can emerge from any weak learning classification tool. This concept was proven through the development of the first boosting algorithms by Schapire [40] and Freund [41].

Where a binary classification exists, the rate of accurate classification from a weak learner is somewhat higher than the rate from a random guess, while a strong

learner should obtain an almost completely accurate classification. In theory, the process of obtaining a strong learner is difficult in comparison to obtaining a weak learner [42].

Boosting algorithms learn by obtaining an accurate classification through iteratively merging weak learners. The concept of boosting was mentioned by Schapire and Freund in [43], where the classification of each base-learner achieves a somewhat better result than random guessing, because each base-learner has some useful information about the problem structure. However, there would be no difference in the weak learner's performance by only repeating it several times on the same training set [44]. In boosting, the base-learner performance is improved by not only handling the base-learner itself, but by iterative re-weighting of the training set [43]. Therefore, the base-learner will obtain a different result from the information in every loop.

The success of the base-learner in one round will play a main role in objects' weighting in the next round. As a result, misclassified objects will receive more concentration by being assigned higher weights.

The precision in boosting is yielded by incrementally raising the importance of the objects that are hard to classify. For each round, the weights are calculated based on the successful classification results of prior rounds. During each round, all misclassified objects from previous rounds will receive more attention. At the last stage, one accurate prediction is created by putting together all the outputs of the prior base-learners [44].

Another well-known ensemble learning method is called bagging [45]. The name bagging was derived from "bootstrap aggregation". It was one of the easiest approaches to improving classification by using multiple samples from one data set.

At the beginning, bagging was used for decision tree models; however, it can also be applied for classification or regression. In the bagging approach, many different training set versions are used, where each set is applied to the training of a different model. In a regression situation, the results of the models are combined by averaging. However, in a classification situation, the results are combined by voting to generate one result. When a small modification occurs in the training set, then a serious change can happen in the model. As a result, Bagging is more successful in unstable nonlinear models [44, 46].

2.3 AdaBoost Algorithm used by Viola and Jones

One of the main contributions of Viola and Jones is the integral image [2, 13, 14]. The benefit of using the integral image is to speed up the computation of rectangular features used in AdaBoost. This section reviews the calculations in the integral image and describes the AdaBoost algorithm.

2.3.1 Integral Image

To get the integral image position values x, y, the summation of the all pixel values located above and to the left of x, y is taken. Figure 2.1 explains this concept.

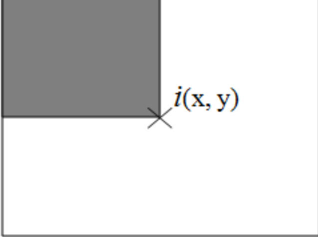


Figure 2.1 The summation of all pixels on top and to the left of x,y is the integral image value at x,y

The following equation explains the computations for the integral image:

$$i(x,y) = \sum_{x' \leq x, y' \leq y} o(x', y')$$

where $i(x,y)$ $i(x,y)$represents the integral image value, and $o(x,y)$ represents the original image value.

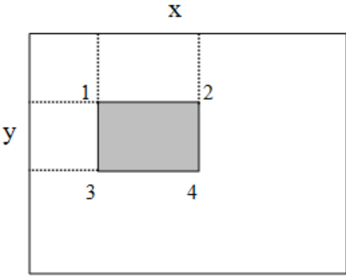


Figure 2.2 Calculating the integral image in a rectangular region

For obtaining the integral image value for the dark rectangle in Figure 2.2, integral image values of points 1, 2, 3, and 4 are calculated for the following equation:

$$i(\text{dark rectangle}) = 4 + 1 - (2 + 3)$$

2.3.2 Extracted Feature Types and Selection

The features extracted in Viola and Jones' algorithm are based on Haar features [2, 13, 14, 47]. By applying integral images, calculation of Haar features can happen in a constant time [48].

A further advantage of Haar features is the power of the features in dealing with lighting variance and noise. Subtracting the difference between the dark and light sides of the rectangle prevents the impact of noise and lighting variance, which would affect all

the pixel values of the entire image equally [49]. Another advantage of Haar features is the easiness of altering the scale of the features [50].

The five types of Haar features used in the original algorithm are shown in Figure 2.3.

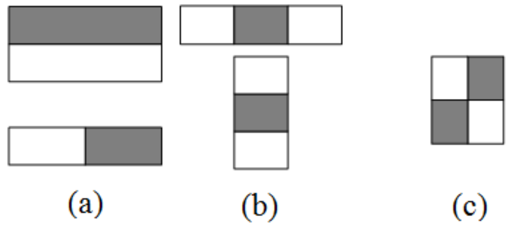


Figure 2.3 Five rectangular features. Figure (a) shows two rectangle horizontal and vertical features, figure (b) shows three rectangle horizontal and vertical features, and figure (c) shows a four rectangle feature

To calculate the value of an image based on a particular feature (feature value), the sum of the integral image values for the pixels located in the light side of the rectangle are subtracted from the dark side [2, 13, 14]. The window size used for training and detecting purposes in face detection is 24x24 pixels. For scaling, the starting point is the smallest size of a rectangular feature, e.g., in a three rectangle feature type, it is 3x1 pixels; in a two rectangle feature type, it is 2x1 pixels. Each rectangular feature is scaled up until reaching a total window size of 24x24. As a result, the total number of features for each type is:

- For a three rectangle feature type, 27,600 features
- For a two rectangle feature type, 43,200 features
- For a four rectangle feature type, 20,736 features

The total number for all features combined is: three rectangles horizontal + three rectangles vertical + two rectangles horizontal + two rectangles vertical + four rectangles = 27,600 + 27,600 + 43,200 + 43,200 + 20,736 = 162,336 features. During the learning

phase, all of these features will be computed for all faces in the training set. The set of faces used for training purposes in this study is the same one that has been used by Viola and Jones for face detection [2, 13, 14]. The size of each image is 24x24. There are 4,916 faces and 7,960 non-faces in the set [51]. Thus, the total number of all possible features in all training images is 2,090,238,336 (i.e. number of training images multiplied by features per image).

Lienhart and Maydt [52] added a set of 45° rotated Haar features, which improve the basic set of Haar features that was used by Viola and Jones, and which can also be computed in an efficient way in constant time at all scales. In addition, the extended set adds extra domain-knowledge to the training phase, which is tough to learn. Lienhart and Maydt mentioned that by employing these extra rotated Haar features, the false alarm rate decreased by about 10% at a specific hit rate.

14 feature prototypes were added by Lienhart and Maydt, which are divided into three different groups: eight line features, four edge features, and two center-surround features, as shown in Figure 2.4, Figure 2.5, and Figure 2.6.

- Eight line feature prototypes

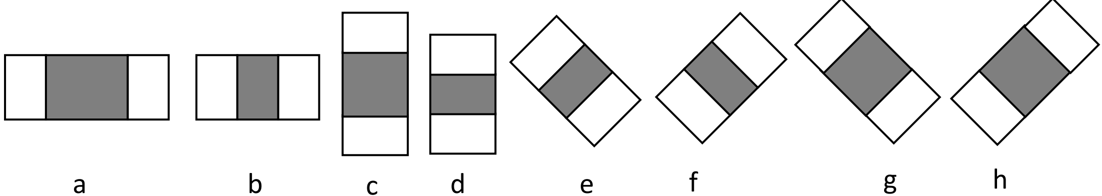


Figure 2.4 Features (a, b, c, d, e, f, g, and h) are examples of line features

- Four edge feature prototypes

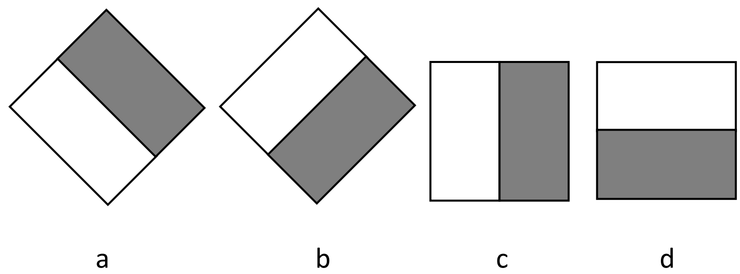


Figure 2.5 Features (a, b, c, and d) are examples of edge features

- Two center-surround feature prototypes

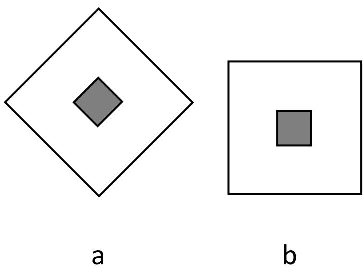


Figure 2.6 Features (a and b) are examples of center-surround features

After applying only the features that were not used by Viola and Jones on window size 24x24, the total number of features obtained after scaling is shown in Table 2.1.

Table 2.1 Total number of features using 24x24 pixels window size for Lienhart and Maydt's features [52]

Line features	**Number of features**
a	20,736
c	20,736
e	4,356

f	4,356
g	3,600
h	3,600
Edge features	**Number of features**
a	8,464
b	8,464
Center-surround features	**Number of features**
a	1,521
b	8,464
Total	**84,297**

Lienhart and Maydt's features were not implemented in this dissertation. However, adding the features that were used by Lienhart and Maydt (a total of 84,297 features) to the features that were used by Viola and Jones (a total of 162,336) would result in using a total of 246,633 features, which would increase the training time to find the best feature among total features.

Viola and Jones have used AdaBoost to select features and combine weak classifiers into one strong classifier [53]. Peter Harrington [54] has mentioned that "AdaBoost is considered by some to be the best supervised learning algorithm."

The conventional AdaBoost algorithm used by Viola and Jones works by assigning a weight to each image in the training set, where all images have an equal weight at the beginning. Training is started on the training set to obtain a first weak classifier and calculate the error of this weak classifier. Next, the weights of all images in the training set are adjusted based on the error, where falsely classified images are assigned a relatively higher weight. Then, the second weak classifier is trained based on the new updated weights [2, 13, 14, 54]. The training will be repeated to extract more weak classifiers, and the weights will be adjusted on every iteration until reaching the maximum number of iterations that is assigned by a user.

As a result, each α value for every weak classifier is determined based on the error value (ε) of that weak classifier, where each error value is calculated by dividing the number of misclassified images over the total number of images [54]. Therefore, α value is calculated as seen in the following equation:

$$\alpha = \log \frac{1}{\varepsilon / (1 - \varepsilon)}$$

The main goal of a weak classifier is to get the optimal threshold among positive and negative examples for any rectangular feature. This technique has been known as the decision stump, which is considered a simple decision tree [55]. The pseudocode of implementing the decision stump, using the indexes of the sorted values of the features to accelerate the process, is explained later in this chapter. The selected threshold minimizes

the number of misclassified examples. The decision of a weak classifier is 1 or 0, i.e. positive or negative, as shown in the following equation:

$$h_j(x) = \begin{cases} 1, & if\ p_j f_j(x) < p_j \theta \\ 0, & \text{otherwise} \end{cases}$$

where p is either 1 or -1, θ is the threshold, and f is the feature.

Pseudocode of the AdaBoost algorithm:

- Suppose there are N number of images in a training set. Each image is labelled as either 0, for negative images, or 1, for positive images, as shown in Table 2.2

Table 2.2 Example of images and labels

Images	x_1	x_2	x_3	....	x_N
Label	1	0	0	....	1

- Initializing the weight for each image in the first round is shown in the following table:

Table 2.3 Example of images, labels, and weights

Images	w_{1,x_1}	w_{1,x_2}	w_{1,x_3}	....	w_{1,x_n}
Label	1	0	0	....	1
Weight	$\frac{1}{2l}$	$\frac{1}{2m}$	$\frac{1}{2m}$	....	$\frac{1}{2l}$

where *l* is the total number of positive images (i.e. faces) and *m* is the total number of negative images (i.e. non-faces) in the training set

- For t = 1 to T:

1. Normalize the weight of each image in each round as follows:

$w_{t,x_i} = \frac{w_{1,x_i}}{w_{sum}}$, where w_{sum} is the sum total of the weights of all images in the same round.

2. Calculate the error of all features, until finding the feature that has the minimum error. The selected feature is the best weak classifier in t^{th} round

$$\epsilon_t = \min_{f,p,\theta} \sum_{x_i} w_{x_i} \left| h(x_i, f, p, \theta) - y_{x_i} \right|$$

3. Based on the minimum error (ϵ_t), which is determined by f_t, p_t, and θ_t, find $h_t(x)$, where:

$$h_t(x) = h(x, f_t, p_t, \theta_t)$$

4. As preparation for the next round, the weight should be updated:

$$w_{t+1,x_i} = w_{t,x_i} \beta^{1-e^{x_i}}$$

where $e_{x_i} = 1$ if x_i is misclassified, $e_{x_i} = 0$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$

- At the end, after going through all rounds, the strong classifier is determined as follows:

$$C(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & otherwise \end{cases}$$

where $\alpha_t = \log \frac{1}{\beta_t}$

Pseudocode of the decision stump

1. Set minimum_error = +∞
2. Set threshold = 0
3. Set sign = 0
4. Set current_error = 0
5. Set e_p = 0
6. Set e_n = 0
7. Set epsilon = 0
8. Set index = 0
9. Set h_classification = 0
10. Set total_error = summation of the weight of all images
11. Set negative_weights_sum = summation of the weight of all negative images only
12. Set positive_weights_sum = summation of the weight of all positive images only
13. Set negative_weights_sum_below = negative_weights_sum
14. Set positive_weights_sum_below = positive_weights_sum
15. For 1... Total number of images

    Set index = the index of the image in the images list as ordered by the feature value

    If the label of the current image in the loop returned by index presents a positive label

        Subtract the weight of that current image from positive_weights_sum_below

    Else

```
            Subtract the weight of that current image from
            negative_weights_sum_below
        Set e_p = positive_weights_sum_below + (negative_weights_sum -
            negative_weights_sum_below)
        Set e_n = negative_weights_sum_below + (positive_weights_sum -
        positive_weights_sum_below)

        If e_p < e_n
            Set current_error = e_p
                If minimum_error > current_error
                    Set minimum_error = current_error
                    Set threshold = the feature value of the current image
                    Set sign = -1
        Else
            Set current_error = e_n
              If minimum_error > current_error
                    Set minimum_error = current_error
                    Set threshold = the feature value of the current image
                    Set sign = +1
16. For 1... Total number of images
        If the feature value of the current image in the loop > threshold
            Set h_classification = sign
        Else
```

Set h_classification = sign * -1

If h_classification value does not match the label of the current image in the loop

Set epsilon = epsilon + the weight value of the current image in the loop

2.4 Previous Research on Parallel and Distributed AdaBoost

Some AdaBoost algorithms that were mentioned in the literature paid more attention to enhancing the generalization error or the degree of toughness when it dealt with noisy data [23, 56, 57]. Additionally, different attempts have been made to parallelize AdaBoost to speed up learning time. K. Zeng et al. [25] parallelized AdaBoost through the hybrid usage of MPI, OpenMP, and transactional memory. By sharing data through transactional memory, OpenMP has been used for low level parallelization and MPI for high level parallelization. K. Zeng et al. [25] built a heterogeneous PC cluster system of sixteen PCs using two processor types. PCs with the same processor type were grouped. In one group, there are 8 PCs with Core2 Quad 2.8G processors and in the other group there are 8 PCs with Core2 Dual 2.8G processors. Each group has the PCs connected to each other through 100Mb Ethernet. The total number of images in the set includes 64,328 faces and 43,712 non-faces. The size of each image is 20x20 pixels, and the total number of Haar features is 299,298. Their AdaBoost algorithm ran for 1,000 rounds. K. Zeng et al. [25] reported a speedup of 31.2 over a sequential implementation with 48 cores. In our implementation where we used the Intel Xeon Quad-Core Processor E5-1620 3.6GHz and a RAM size of 16 GB, the speedup obtained was 44.5 when using 48 cores.

Another attempt was made by Lazarevic and Obradovic [26] to solve the problem of learning an enormous homogeneous database for training classifiers by distributing the database over many places. The database size was too large to fit in one place, preventing it from being located in the computer memory. Also Lazarevic and Obradovic [27] parallelized AdaBoost using a database that was distributed because of the enormous size of the datasets. Their approach is based on dividing the data into separate small sets to use each set later in training base classifiers. Outputs of these classifiers are then combined based on their accuracy. Their parallel boosting algorithm is capable of working on a distributed database; however, it is not suitable to work in environments such as distributed cloud computing since their architecture relies on the shared memory systems technique. In addition, Merler et al. [28] used weight dynamics to parallelize AdaBoost and named it P-Adaboost. It is noticed that the outputs of these three previous parallel AdaBoost algorithms [26-28] do not match the sequential AdaBoost outputs because the authors made some changes to the training process of the original version of AdaBoost. However, K. Zeng et al. [25] found the same output for both parallel and sequential AdaBoost. Idealy, there should be no differences in the output of parallelized AdaBoost and sequential AdaBoost except to reduce the training time.

Galtier et al. [29] used JavaSpace and MPJ to parallelize AdaBoost where the output of parallel AdaBoost is same as in the sequential one. Because of hierarchical hardware architecture, K. Zeng et al. used the hybrid programming model. However, Galtier et al. [29] used the message passing model. In addition, Galtier et al. [29] parallelized AdaBoost by distributing balanced load tasks to multiple workers. Their

work is based on heterogeneous processors in a geographically distributed environment. However, this kind of setup produces latency on the network. The authors mentioned that using virtual shared memory in large distributed locations did not prevent them from speeding up the process in many cases.

Galtier et al. [29] used 134,736 Haar features with a training dataset of 8,500 24x24-pixel images. The speedup obtained with 64 workers on a homogenous cluster was 53.9 as compared to their sequential implementation. In our algorithm, the total number of Haar features is much higher (i.e. 162,336) and the training dataset size is also larger (12,876 images of 24x24 pixels). Despite the large dataset and feature sizes, using 64 cores, we obtain a speedup of 55.7 and much less execution time per feature.

Palit and Reddy [1] used parallel AdaBoost to get a boosted ensemble classifier by using the computation process in many nodes working at the same time. This resulted in a better performance as compared to using the serial version of AdaBoost, and they were able to speed up computation time. One of the reasons for this is that each one of the nodes used for computation purposes works individually, where there is no data shared between nodes. As a result, there is no communication occurring between nodes. The basic structure of their parallel algorithm does not follow the basic structure of the serial version of AdaBoost. Palit and Reddy implemented and tested their parallel algorithm using the MapReduce framework. Their approach makes the computation process of each node independent from the other nodes in order to reduce the communication cost between nodes.

Another approach for speeding up AdaBoost, called LazyBoost, was done by Escudero et al. [30], which takes advantage of using different feature selection and ranking methods by training the base learner in each boosting iteration on a random small set of features of a selected fixed size. Another fast boosting algorithm has been developed by Busa-Fekete and Ke´gl [31] that uses multiple-armed bandits (MAB). In MAB, there are many small groups of the base classifier sets that are considered “arms”. Therefore, in each boosting iteration, there is only one small set among all the small sets that will be chosen to be used by the boosting algorithm, instead of training the base classifier using all the sets. However, the work to speed up AdaBoost mentioned in [30, 31] was done on a single machine, and a parallel and distributed architecture was not applied; therefore, their achievement was bounded by the resources of the single machine used.

There is another approach in boosting parallelization, where the parallelization occurs in the weak learner instead of parallelizing the ensemble itself. Wu et al. [32] used MapReduce to create an ensemble of C4.5 classifiers, and they called it MReC4.5. To enable the classifiers to be used in different environments, they have to be established in the cloud or on cluster computers, and any operations on serial form should be sequenced at the model level.

Another framework built using MapReduce on a large data set for regression trees and learning classification purposes was done by Panda [33], and is called PLANET. However, the previous approaches [32, 33] cannot be generalized to be used in boosting ensemble methods, since they are mostly limited to use for weak learners.

In another parallelizied boosting algorithm, Fan et al. [34] implemented a scalable and distributed learning type of boosting algorithm. Among the entire training set, only a small part is used to train each different classifier. Two different ways have been used to train the classifiers: either using random samples (r-sample), or the entire training set is disjointed into separate small sets (d-sample) so that on every round, the weak learner will receive a different d-sample. The main focus of the approach in Fan et al. [34] was parallelization in space, not in time. By distributing massive data on multiple nodes, the space problem was handled. However, gaining faster speed in processing time had not yet been reached.

Another algorithm called the MultBoost algorithm was proposed by Gambs et al. [35]. It was basically created to accomplish computation privacy. The design of the MultBoost algorithm enables it to work in a parallel setting. However, it can be parallelized in both space and time if each node has independent data, and if the computation process of each node is totally isolated, and does not know or rely on other nodes' data. In the MultBoost algorithm, two or more nodes are able to start a boosting classifier in a privacy-preserving setting.

Huang and Shi [24] implemented a parallel and distributed AdaBoost. All the features that were used in AdaBoost training were divided into multiple feature blocks, where the structure of the feature block is based on three components: block number, basic data, and optimal feature of that block. The distributed system is based on client computers and a server pool, where all the computing nodes are in the server pool. In Adaboost, there are sequential and parallel parts; the client is responsible for the

sequential part, and all the parallel parts will be sent to the computation nodes. Once each node completes the assigned task, the output will be sent back to the client to combine it and produce the final result. Huang and Shi used 5,646 face and 13,030 non-face images collected from the Internet as a training set, and the size of each image was 20x20 pixels. They tested their parallel algorithm on 5 PCs where the CPU speed was 1.8GHz on each PC. The maximum obtained speed based on extracting 64 features was 2.66 times faster compared to their implementation of the sequential AdaBoost algorithm. However, using N nodes did not allow them to reach the ideal value of N times because the weights were updated in each iteration, which could not be parallelized. Also, they were not able to save that huge number of features in the memory simultaneously, which led to longer training.

Lozano and Rangel [36] worked to speed up boosting algorithm running time by changing the training process of the base learners from sequential to parallel. However, they accomplished boosting parallelization on different algorithms equivalent to AdaBoost, where several individual distributions of the data were used to train many base learners concurrently before combining them. Their algorithm is executed in batches, where the first batch works like bagging [37]. Probability simplex was used to obtain random individual distributions to train base classifiers. Different approaches were then implemented in the subsequent batches, which depended on the information generated from the execution of the base learners in the prior batches, in order to get a useful distribution. One main processor rapidly generated all these distributions and then sent them to individual processors to obtain a new trained batch of base learners. Lozano and

Rangel focused on methods more than performance and implementation problems. However, Galtier et al. [29] paid more attention to performance and implementation problems. Lozano and Rangel's [36] approach is different as compared to Lazarevic and Obradovic's [27] approach in that the whole data set is used to train the base learner of every processor [36].

2.5 Previous Research on Training Classifiers on a Small Training Set

Freund and Schapire [16] mentioned that it is possible to train a base learner using only a small set of randomly selected data instead of using the entire weighted data, by using the weight vector as a discrete probability distribution.

FilterBoost [58] is an algorithm for using a small set to train a base learner. It is based on the modification [59] of AdaBoost aimed at reducing the logistic loss. In every boosting iteration while using the FilterBoost algorithm, many labeled samples are generated. However, the base learner has the choice to accept or reject the generated sample point, and among the accepted points, only a small set is used in the base learner training process.

C. Shen et al. [60] proposed an approach to training classifiers to detect faces based on a small training set by using covariance features [61] instead of Haar features, and Fisher Discriminant Analysis (FDA) instead of decision stump.

After Viola and Jones [2] developed real-time face detection based on AdaBoost, much has been done to improve their work either by enhancing the boosting method or by speeding up the training process; for example, S. Z. Li and Z. Zhang [62] presented an enhanced FloatBoost method that enhances detection accuracy by using backward feature

selection in AdaBoost training. J. Wu et al. [63] accelerated the training process by using forward feature selection. P. Minh-Tri and C. Tat-Jen [64] used approximation to decrease the training time when using the decision stump.

X. Li et al. [65] combined active learning with AdaBoost to increase the classification performance of AdaBoost while using a small sample set for training. AdaBoost is applied to the current labeled sample set to get an optimal hyperplane, and selecting any unlabelled sample that is closest to the optimal hyperplane. One of the drawbacks of using active learning is the risk of uninformative examples, which overwhelm the active learning algorithm.

K. Levi and Y. Weiss [19] worked on training classifiers to detect objects based on small training sets. However, their features were presented using local edge orientation histograms (EOH) instead of Haar features.

2.6 Summary

Using both a smaller amount of data and features by reducing the number of data points that participate in training the base learners, and reducing the search area by using only a small set of features will result in a significant reduction in computation time, when compared to using only one of these methods.

There is another way of training classifiers by using online learning [66] instead of offline learning, which the majority of training processes do. N. C. Oza [66] proposed an online version of bagging and boosting. The way online learning algorithms work is by dealing with every single example of a training set as it arrives. Therefore, no example of the training set will be saved or reused. Regarding the speed of such an online

algorithm, N. C. Oza has mentioned that “such algorithms run faster than typical batch algorithms in situations where data arrive continuously. They are also faster with large training sets for which the multiple passes through the training set required by most batch algorithms are prohibitively expensive.” [66]

The critical part in online learning is the computation process of sample weights, because there is no previous knowledge about how difficult the sample is. As a result, an estimation of the importance of the sample is necessary while this sample goes through every base classifier [67].

Since our proposed framework is based on parallel AdaBoost and web services, and uses an optimized training set, memory is not a problem as in other cases. For now, there is no need to use online learning because of parallel and distributed offline learning.

CHAPTER 3: RESEARCH PLAN

3.1 Parallel and Distributed Web Services-Based AdaBoost Architecture

This chapter presents the implementation of the proposed framework in parallel and distributed AdaBoost. The original AdaBoost determines the best weak classifier in each round based on the minimum classification error. The AdaBoost algorithm must go through all features to determine which feature yields the minimum error. Since there is a large number of features, the execution time during the learning phase is high. Our parallel approach [68] speeds up the execution time by efficiently parallelizing the AdaBoost algorithm. We implement a four-way approach in order to get results in the shortest possible time. We run the main computational part of AdaBoost in parallel, using Task Parallel Library (TPL). Task Parallel Library is a library built into the Microsoft .NET framework. The advantage of using TPL is apparent in multi-core CPUs where the declared parallel workload is automatically distributed between the different CPU cores by creating lightweight threads called tasks [69].

To further improve the execution time of AdaBoost, we use web services to run parallel Adaboost on multiple workstations in a distributed manner. Our first-level architecture does workload division based on feature type. Since there are five feature

types, we use five workstations at Level 1. As shown in Figure 3.1, a total of six machines are used, one master and five slave nodes. To achieve further scalability, computation of each of the feature types is further expanded to a second level using web services. Figure 3.2 illustrates twenty-one PCs being used in a two-level hierarchy: master, sub-masters, and slave nodes. Figure 3.3 shows N number of PCs being used in a one-level hierarchy: master and slave nodes.

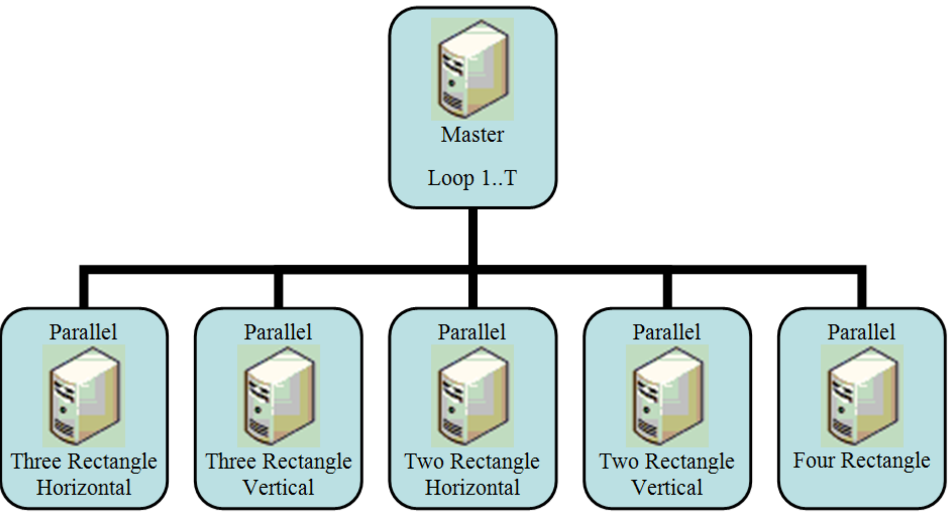


Figure 3.1 One-level hierarchy for Web Services and Parallel AdaBoost, based on one master and five slave nodes (total of six PCs)

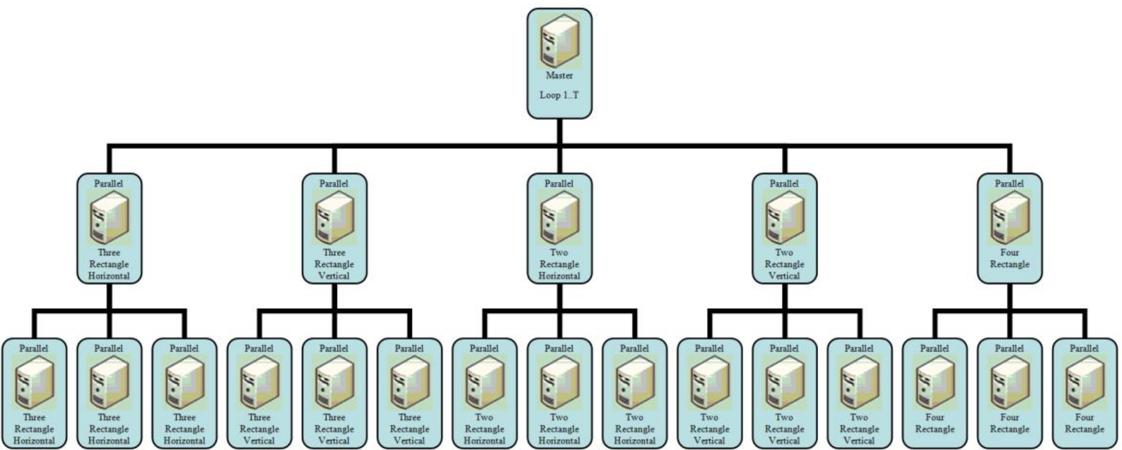


Figure 3.2 Two-level hierarchy for Web Services and Parallel AdaBoost, based on one master, five sub-master nodes, and 3 slave nodes for each sub-master node (total of twenty one PCs)

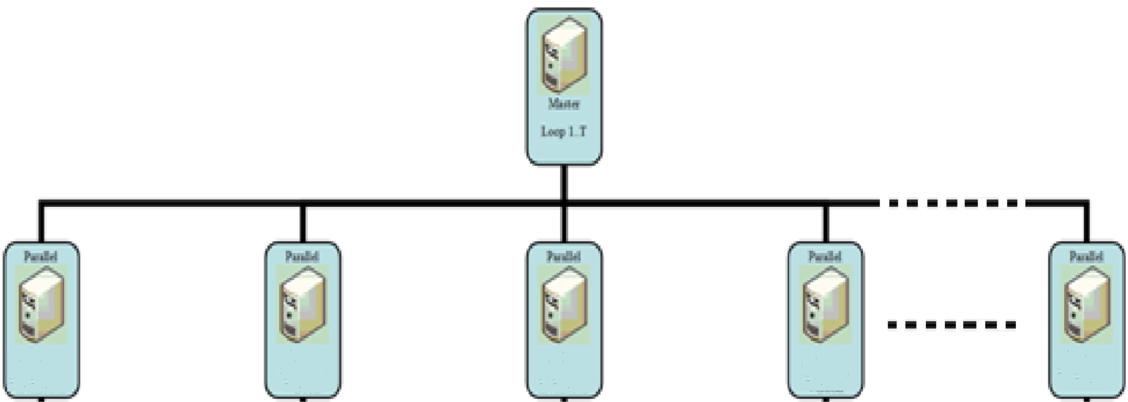


Figure 3.3 One-level hierarchy for Web Services and Parallel AdaBoost, based on one master and N slave nodes

The following list consists of four approaches for speeding up execution times that are implemented in the AdaBoost algorithm:

- Parallel execution
- Web Services and Parallel execution on one-level hierarchy
- Web Services and Parallel execution on two-level hierarchy

- Web Services and Parallel execution on one-level hierarchy with N number of slave nodes

3.1.1 Parallel Execution

All features are grouped based on type, such as three rectangle horizontal, three rectangle vertical, two rectangle horizontal, two rectangle vertical, and four rectangle. Each group is uploaded to the system memory in parallel. Once all of these have been loaded, the AdaBoost rounds from 1 to T begin. For each round, the result is to locate five features that have minimal error, in parallel, from five different groups. Among the five selected features, the feature that possesses the least minimum error is chosen. From that selected feature, the weights of all images are updated in preparation for the next round of the AdaBoost algorithm. One advantage of the previous mentioned approach lies within its execution time. Since selecting a minimum error feature runs simultaneously, the execution time is reduced by approximately a factor of five.

3.1.2 Web Services and Parallel Execution on One-level Hierarchy

Each group of features is distributed to a separate PC. Since five groups exist, five PCs are used for feature calculations, and the master coordinates the five PCs as shown in Figure 3.1. The parallel and distributed pseudocode for this approach is described below.

Pseudocode of one-level hierarchy (master and five slave workstations) Parallel and Distributed AdaBoost:

1) Example images $(x_1, y_1), \dots, (x_n, y_n)$ where $y_i = 0,1$ for negative and positive examples, respectively, are given
2) Prepare one master workstation and five slave workstations
3) Each slave workstation is assigned to one particular feature type
   - (slave workstation 1, Three rectangles Horizontal)
   - (slave workstation 2, Three rectangles Vertical)
   - (slave workstation 3, Two rectangles Horizontal)
   - (slave workstation 4, Two rectangles Vertical)
   - (slave workstation 5, Four rectangles)
4) On slave workstations: Initialize all images on each slave workstation
5) On master workstation:
   - Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0,1$ respectively, where m and l are the number of negatives and positives, respectively
   - For $t = 1, \dots, T$:
     1. Normalize the weights, $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j-1}^{n} w_{t,i}}$ so that w_t is a probability distribution
     2. Send the weights to all slave workstations
     3. On each slave workstation:
        a. For each feature j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \, |h_j(x_i) - y_i|$
        b. Send the classifier h_t with the lowest error ϵ_t to master workstation
     4. On master workstation:

a. Among the received classifiers from each slave workstation, choose the classifier h_t with the lowest error ϵ_t

b. Update the weights: $w_{t+1,i}=w_{t,i}\beta_t^{1-e_i}$ where $e_i = 0$ if example x_i is classified correctly, otherwise $e_i = 1$, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$

6) The final strong classifier is:

$$h(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases}$$

where $\alpha_t = \log \frac{1}{\beta_t}$

3.1.3 Web Services and Parallel Execution on Two-level Hierarchy

The previous technique divided the work based on feature type. Now, we further distribute the calculations in a feature type to another set of machines in the next hierarchical level as shown in Figure 3.2.

Pseudocode of two-level hierarchy (master, five sub-master workstations, and 25 slave workstations) Parallel and Distributed AdaBoost:

1) Example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0,1$ for negative and positive examples, respectively, are given
2) Prepare one master workstation, five sub-master workstations, and twenty five slave workstations
3) Each sub-master and its slave workstations are assigned to one particular feature type
   - (sub-master workstation 1 and 5 slave workstations, three rectangles Horizontal)

- (sub-master workstation 2 and 5 slave workstations, three rectangles Vertical)
- (sub-master workstation 3 and 5 slave workstations, two rectangles Horizontal)
- (sub-master workstation 4 and 5 slave workstations, two rectangles Vertical)
- (sub-master workstation 5 and 5 slave workstations, four rectangles)

4) On all slave workstations: Initialize all images on each slave workstation
5) On master workstation:
   - Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0,1$ respectively, where m and l are the number of negatives and positives respectively
   - For $t = 1, \ldots, T$:
     1. Normalize the weights, $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j-1}^{n} w_{t,i}}$ so that w_t is a probability distribution
     2. Send the weights to all sub-masters, and each sub-master sends the weights to its slave workstations
     3. Sub-masters divide the features between their slave workstations, where each slave workstation is responsible for an equal portion
     4. On each slave workstation:
        a. For each feature j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \, |h_j(x_i) - y_i|$
        b. Send the classifier h_t with the lowest error ϵ_t to the assigned sub-master workstation
        c. Each sub-master chooses the classifier h_t with the lowest error ϵ_t amongst their slave workstations and sends it to the master workstation
     5. On master workstation:

a. Among the received classifiers from each sub-master, choose the classifier h_t with the lowest error ϵ_t

b. Update the weights: $w_{t+1,i}=w_{t,i}\beta_t^{1-e_i}$ where $e_i = 0$ if example x_i is classified correctly, otherwise $e_i = 1$, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$

6) The final strong classifier is:

$$h(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases}$$

where $\alpha_t = \log \frac{1}{\beta_t}$

3.1.4 Web Services and Parallel Execution on One-level Hierarchy with *N* Number of Slave Nodes

The previous techniques divided the work based on feature types. Here, we distribute the work to all slave nodes, in balance, where each node will receive almost the same amount of data. Figure 3.3 shows the hierarchal master-slave configuration.

Pseudocode of one-level hierarchy (master and N slave workstations) Parallel and Distributed AdaBoost:

1) Example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0,1$ for negative and positive examples, respectively, are given
2) Prepare one master workstation and N slave workstations
3) Each slave workstation is assigned to an equal number of features
4) On master workstation:

- Distribute all images to each slave workstation in parallel in order to initialize and sort the feature values of the images
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0,1$ respectively, where m and l are the number of negatives and positives, respectively
- For $t = 1, \dots, T$:

5. Normalize the weights, $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j-1}^{n} w_{t,i}}$ so that w_t is a probability distribution
6. Distribute the weights to all slave workstations in parallel
7. On each slave workstation, where each slave workstation will work with other slave workstations:
   a. For each feature j, train in parallel a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to $w_t, \epsilon_j = \sum_i w_i \, |h_j(x_i) - y_i|$
   b. Send the classifier, h_t with the lowest error ϵ_t to master workstation
8. On master workstation:
   a. Among the received classifiers from each slave workstation, choose the classifier h_t with the lowest error ϵ_t
   b. Update the weights: $w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$ where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$

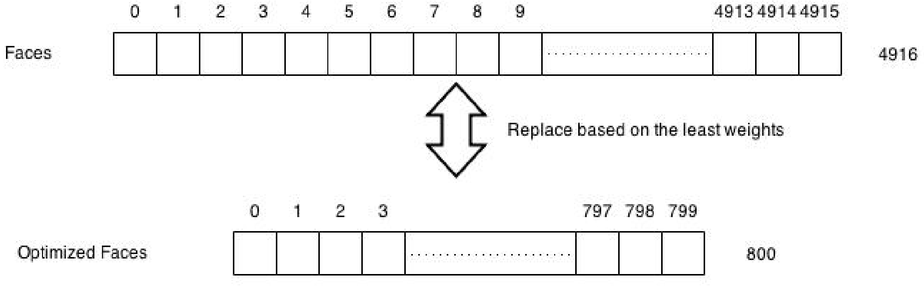
5) The final strong classifier is:

$$h(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases}$$

where $\alpha_t = \log \frac{1}{\beta_t}$

3.2 Extracting Minimum Size Subset that has 100% Detection Rate for Faces and Non-faces

Viola and Jones used 4,916 faces and their mirror images, plus a large number of non-faces, to build their attentional cascade. Using that big number of non-faces has reduced the percentage of false positives. However, since we are building a framework able to re-train in a near real-time fashion, we have tried to find the optimized subset for both faces and non-faces that is capable of achieving a 100% detection rate once testing the extracted features of that subset on the entire set. The training set we have used has 4,916 faces and 7,960 non-faces [51]. We started with a small set of faces and non-faces and continuously replaced some of the images that have lowest weights based on the AdaBoost algorithm. After many tries and replacements, we have found that extracting 200 features from a subset of 800 faces and 800 non-faces was able to correctly classify all the images of the entire set, which means from a total of 1,600 faces and non-faces we were able to correctly classify the entire set of size 12,876. Based on that, 12% of the training set can be considered the optimized set that is capable of obtaining a high face detection rate and reducing the number of false positives.



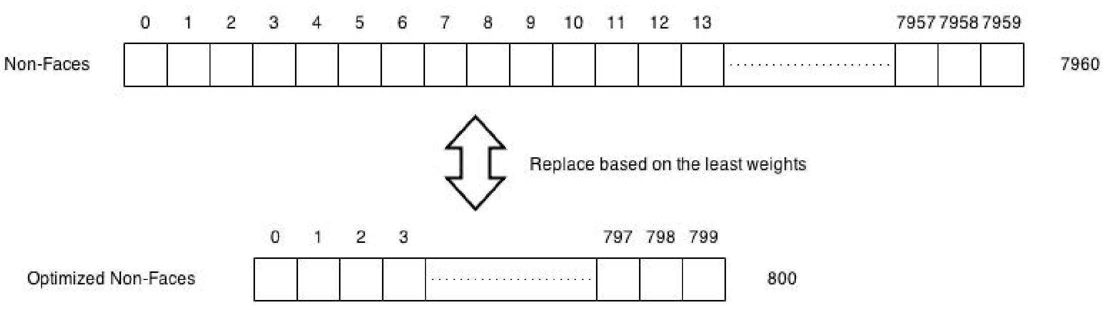


Figure 3.4 Extracting an optimized training subset

Based on this trial, when we started to build the classification cascade, we set the minimum size of the subset to 800 faces and 800 non-faces. Therefore, in each stage, the classifier is based on a small part of the training set. However, the extracted features are tested on the entire set, and each time many images of faces and non-faces that have the lowest weights are replaced until a 100% detection rate is achieved. Then the process jumps to the next stage and replaces images until a specific condition is satisfied. Figure 3.4 shows the optimized subset extracted from the entire set and Figure 3.5 shows the value of each feature extracted from an optimized subset of the training set.

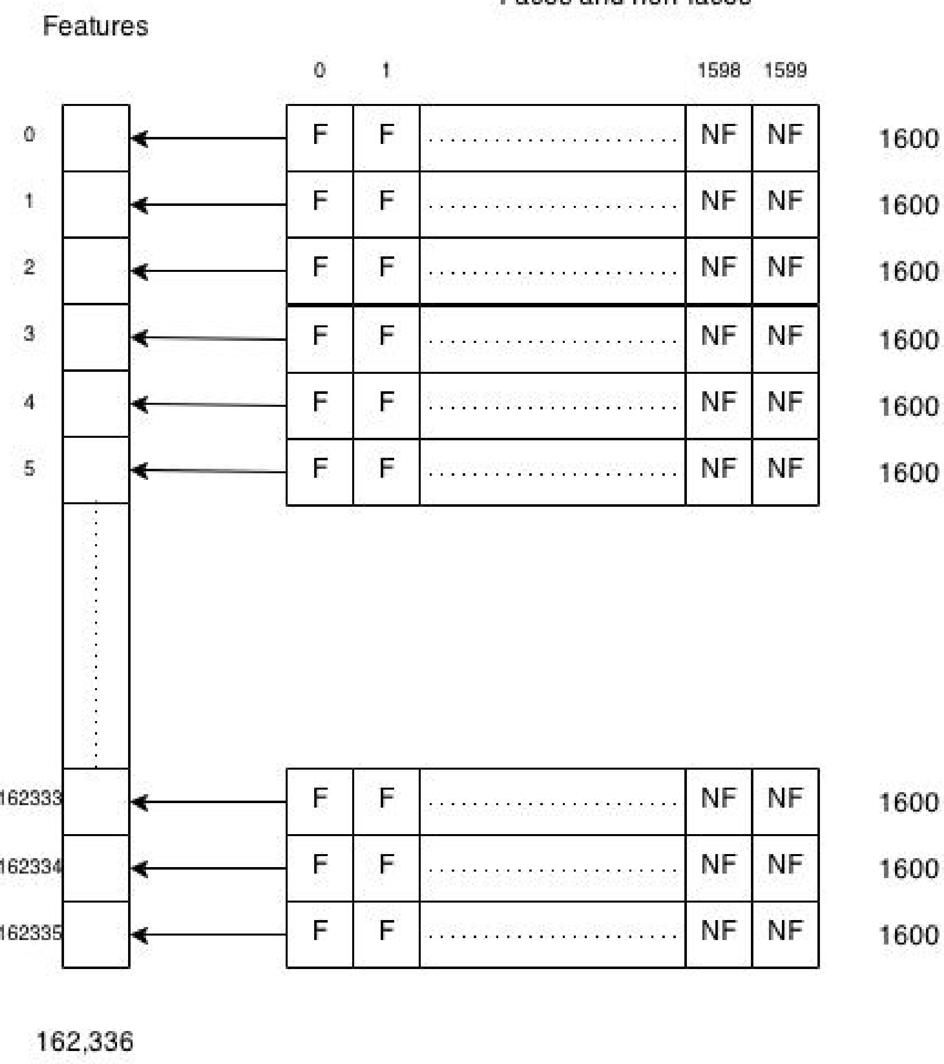


Figure 3.5 Each feature value is calculated in each image from the optimized subset of the training set

The optimization process may be explained mathematically: Given $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ where $x_i \in S, y_i \in Y = \{1,0\}$, to find the smallest subset of images $X \subseteq S$ that maximizes the detection rate $D(S)$, the optimization problem may be formulated as:

$$\arg\max_{x_i \in X} \{D(S) | X \subseteq S, \forall x_i \in S\}$$

- where S, X, and D denote the entire training set, the smallest optimized subset, and the detection rate of the entire set obtained using X, respectively
- a solution $X \subseteq S$ is optimal if the final strong classifier $H(X)$ correctly classifies the positive images $x_i \in S$ where $y_i = 1$, and minimizes the misclassified negative images $x_i \in S$ where $y_i = 0$ as the following:

$$D(S) = 100 - E(H(X))$$

- where $E(H(X))$ denotes the error of the final strong classifier $H(X)$, formulated as:

$$E(H(X)) = \begin{cases} \text{error} = 0, & \text{if } y_i = 1 \\ \arg\min_{x_i \in S} \text{error}, & \text{if } y_i = 0 \end{cases}$$

- where

$$\text{error} = \sum_{i=1}^{m} [y_i - h(x_i)]$$

3.3 Implementing a Cascade Structure Based on a Small Training Set

Viola and Jones built their attentional cascade structure based on the AdaBoost algorithm in order to speed up the detection process by rejecting the majority of the scanned sub-windows. Their cascade has many stages, where in each stage there is a specific number of weak classifiers, and earlier stages have fewer weak classifiers than later stages. This will reduce the computation time by rejecting many of the false positive examples at the beginning. A sub-window will only reach the stages with more weak classifiers if it can successfully pass the earlier stages. This will result in a high face detection rate and reduce the false positive rate. However, to build an efficient cascade

able to detect almost all faces and reject a high percentage of non-faces, many different non-face images to train classifiers on each stage are required.

Since the goal was to build a framework that is able to re-train in near real-time by using a parallel and distributed structure, the cascade was built in a different way, where classifiers on each stage are based on a subset of the training set, instead of the entire set. However, that subset of the training set is still able to obtain a 100% detection rate for each stage in testing the classifiers of each stage on the whole training set. Using the parallel and distributed framework, testing discovered that a strong classifier having 200 weak classifiers can be extracted from a specific set of 800 faces and 800 non-faces to reach a 100% detection rate for both faces and non-faces when it is tested on the entire set. This subset was obtained by replacing the images that have the lowest weights among the subset until achieving an optimized subset that is able to reach 100% detection rates for faces in testing the entire set. From this point, a cascade was built that has 21 stages, where the first stage is based on 5 weak classifiers, the second one has 10 weak classifiers, the third one has 20 weak classifiers, and the number of weak classifiers increases by 10 on each stage until stage 21, which will have 200 weak classifiers. In Viola and Jones' cascade structure, the final stages have 200 weak classifiers, and here the experiment followed Viola and Jones' structure by not having more than 200 weak classifiers for any stage. This was desired because more weak classifiers means more computation time.

In the AdaBoost algorithm that has been used to build classifiers on each stage of the cascade, any sub window will be considered a face when the sum of alphas of all

weak classifiers in the sub window is greater than or equal to 0.5 multiplied by the sum of alphas that were obtained from training classifiers. This result is the threshold that will determine faces from non-faces, as seen in the following equation.

$$h(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq 0.5 \sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases}$$

However, when using a few weak classifiers that were created on the optimized smaller training set, a 100% detection rate for faces will not be achieved unless we reduce the multiplier coefficient for the sum of alphas to less than 0.5. Even though this will result in false positives, we will handle the false positives in an additional stage to be explained later in this chapter. An optimum value for the multiplier coefficient that achieves a 100% true positive rate needs to be determined, as described in Figure 3.6.

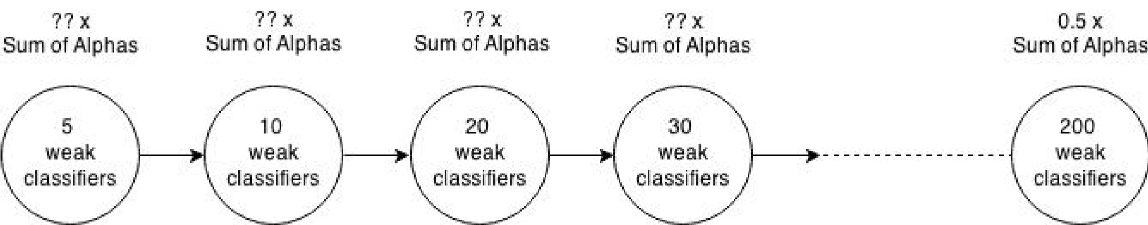


Figure 3.6 Finding the multipliers for the sum of Alphas

To determine the optimum multiplier coefficient for the the sum of alphas, we started with a value of 0.5, and reduced it by .01 every time the detection rate on the entire training set was less than 99.9. Figure 3.7 shows that in the first stage, which has 5 weak classifiers, the coefficient value for multiplying the sum of Alphas is 0.18. In the second stage, which has 10 weak classifiers, the coefficient for multiplying the sum of Alphas is 0.29.

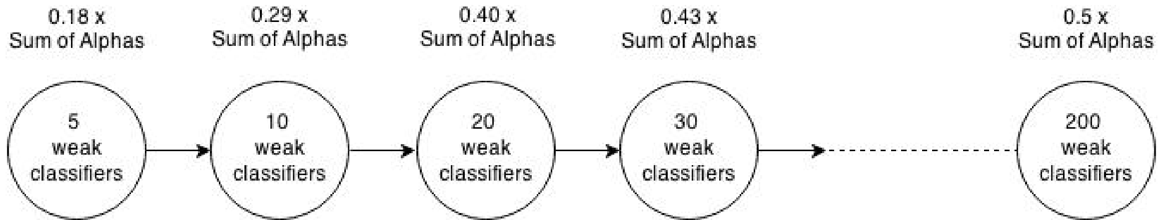


Figure 3.7 Multiplier coefficient for each stage to achieve 100% true positive detection

The final results after finishing our cascade structure are:

1. 21 stages are built where the first stage has 5 weak classifiers, the second stage has 10 weak classifiers, and the third stage has 20 weak classifiers. We then add 10 weak classifiers on every stage as shown in Figure 3.8

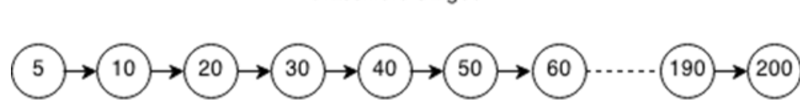


Figure 3.8 Classifier of twenty one stages

2. For each stage, a new threshold was extracted based on multiplying the sum of alphas by the value that will result in a 100% positive detection rate of faces.
3. All weak classifiers were extracted from each stage based on the subset of the training set, while the detection rate was tested on the entire set. 800 faces and 800 non-faces were used as a subset of the training set, where this set is considered an optimized set that has all the images that are able to train classifiers to obtain a 100% detection rate for faces when testing the entire set.

Pseudocode of implementing the cascade structure:

1. Set LargePositiveObjectDataSetImagesSize = 5000
2. Set LargeNegativeObjectDataSetImagesSize = 8000
3. Set OptimizedPositiveObjectDataSetImagesSize = 800
4. Set OptimizedNegativeObjectDataSetImagesSize = 800
5. Set ObjectRateCounter100P = 0, GlobalRounds = 1500
6. Set MaximumFeatures = 200, Threshold = .5, x = 5
7. Set Counter = 0
8. For 1... GlobalRounds

   a) Run parallel and distributed AdaBoost of N slave nodes to obtain x classifiers from an optimized dataset

   b) Obtain the detection rate according to the large dataset

      i. If detection rate of positive objects = 100%

         Increment ObjectRateCounter100P

      ii. If detection rate of positive objects = 100% and ObjectRateCounter100P ≥ (Counter +1)

         Increment Counter

         x = Counter*10

         If x > MaximumFeatures

            Break For loop

         Set Threshold = .05

         Set ObjectRateCounter100P = 0

If total of misclassified negative objects < OptimizedNegativeObjectDataSetImagesSize

Replace the optimized negative objects dataset with objects from the large negative dataset

Else

Replace the optimized negative objects dataset with objects from the misclassified negative objects

Repeat from line 8

iii. If detection rate of positive objects < 99.9

Threshold = Threshold - .01

Else

Threshold = Threshold + .01

iv. If Threshold < .05

Set Threshold=.05

v. According to the misclassified positive objects detection rate, select a percentage of positive objects that have the lowest weight in the optimized positive dataset and replace them with the misclassified positive objects

vi. According to the misclassified negative objects detection rate, select a percentage of negative objects that have the lowest weight in the optimized negative dataset and replace them with the misclassified negative objects

3.4 A Near Real-time Re-training

After building a cascade of 21 stages where each one has a specific threshold to be used for detection purposes, it is time to enable the framework to be re-trained in close to real-time using parallel and distributed AdaBoost. In order to do this, another stage is added, which is copied from the last stage. Now the total number of stages is 22. The reason of creating the new stage is for re-training and for handling false positives. If some objects are not detected using the previous stages or if some false positives are encountered, the last stage only is re-trained based on the missed object or false positives.

The implemented framework (Figure 3.9) was tested on faces and cars. The faces training set is available at [51]. The UIUC Image Database for Car Detection was used as the cars training set [4, 5].

The strategy used in re-training the new stage to detect missed faces and erase false positives is to add any missed faces or false positives of each tested image to the new stage and give them a high weight to have more attention given to them. However, to make the missed faces stronger and the false positives weaker, rectangles that overlap the missed faces but were rejected by pruning, as well as false positives, are added to increase the chance of detecting the missed faces and to erase false positives after completing the re-training process.

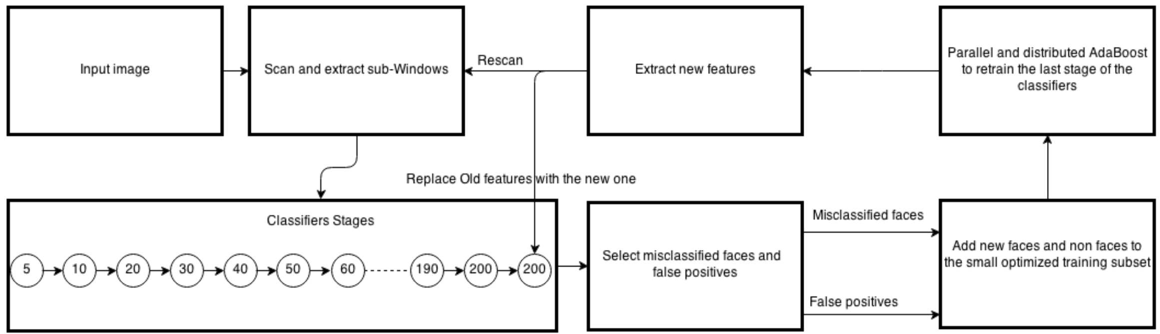


Figure 3.9 Architecture for the Re-trainable System

3.5 Summary

In this work, a parallel and distributed framework converted the AdaBoost algorithm from a sequential version to a parallel and distributed version based on the Microsoft Task Parallel Library (TPL) and web services. In addition, by using the proposed framework, the ability to extract an optimized subset from the entire set was achieved, where the extracted features from the subset were able to produce a 100% detection rate on faces and non-faces after testing these features on the entire set. Also, using the proposed framework to re-train difficult cases, where an object was missed or a false positive existed, was accomplished in near real-time.

CHAPTER 4: RESULTS

4.1 Parallel and Distributed AdaBoost Testing Results

We developed five variations of the AdaBoost algorithm, as follows:

1. Sequential algorithm
2. Parallel on one machine that only uses TPL
3. Web services and parallel execution on one-level hierarchy
4. Web services and parallel execution on two-level hierarchy
5. Web services and parallel execution on one-level hierarchy and N slave nodes

Table 4.1 shows a comparison of the first four approaches that were implemented. These results show a significant improvement in speedup. A speedup of 95.1 times was obtained based on approach number four. Figure 4.1 shows the parallel execution time of the implementation as the number of slave nodes is increased. With 31 machines, an execution time per feature of 4.8 seconds is achieved.

Table 4.1 Comparison of the first four approaches used

	Uploading time to memory – done one time only (in seconds)	Average execution time for each round (in seconds)	Speedup (with respect to sequential execution)
Sequential algorithm on one PC	1780.6	456.5	----
Parallel algorithm on one PC	330.7	116.1	3.9
Parallel and distributed one-level architecture on 6 PCs	92.7	24.6	18.6
Parallel and distributed two-level architecture on 21 PCs	30.3	6.4	71.3
Parallel and distributed two-level architecture on 26 PCs	35.4	5.2	87.8
Parallel and distributed two-level architecture on 31 PCs	31.7	4.8	95.1

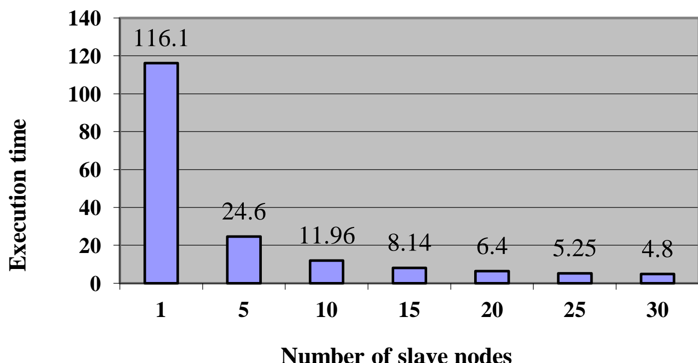


Figure 4.1 Parallel execution time based on the total number of slave workstations

To be able to predict the speedup for any number of machines available, the following predictive equation was developed for calculating parallel execution time, based on the number of nodes in the last level attached to one sub-master node in the middle level.

$$\text{Parallel execution} = (0.2 * \mathrm{n}) + \left(\frac{0.5}{1000}\right) * (\frac{\mathrm{m}}{\mathrm{n}})$$

where n is the number of nodes attached to one sub-master node, and m is the maximum number of features allocated to one sub-master node.

It is noticed that increasing the number of nodes in the last level beyond 7 per feature type will not help further in speeding up execution, since communication overhead in the network is going to be limiting. Table 4.2, Table 4.3, and Table 4.4, and Figure 4.2 demonstrate this.

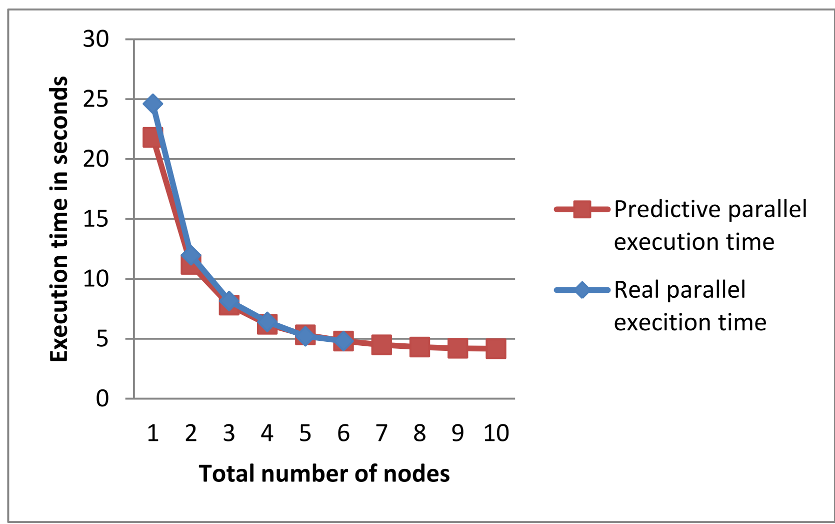


Figure 4.2 Real and predictive parallel execution time based on total number of slave workstations in the last level

Table 4.2 The result of the predictive equation based on the number of nodes attached to one sub-master node

Number of nodes	**Execution time per round (in seconds)**
1	21.8
2	11.2
3	7.8
4	6.2
5	5.3
6	4.8
7	4.5
8	4.3
9	4.2
10	4.1

Table 4.3 Overhead on the one-level network using one master and 5 slave nodes

	Average overhead on network per round (milliseconds)
4 rectangle node	251.04
3 rectangle vertical node	257.8
3 rectangle horizontal node	384.8
2 rectangle vertical node	253.3
2 rectangle horizontal node	356.61

Table 4.4 Overhead on the two-level network using one master, 5 sub-master nodes, and 25 slave nodes

	Average overhead on network per round (in milli-seconds)
4 rectangle node	280.2
3 rectangle vertical node	283.43
3 rectangle horizontal node	334.82
2 rectangle vertical node	294.86
2 rectangle horizontal node	410.3

Table 4.5 and Figure 4.3 show a comparison of the fifth approach with the sequential one, based on different numbers of slave nodes. These results show a significant improvement in speedup. A speedup of 326.1 times was obtained based on approach number five. Figure 4.3 shows the parallel execution time of the implementation as the number of slave nodes is increased. With 26 machines, an execution time per feature of 1.4 seconds is achieved.

Table 4.5 Comparison of the fifth approach with different number of nodes

	Uploading time to memory after receiving from master – done one time only (in seconds)	Average execution time for each round (in seconds)	Speedup (with respect to sequential execution)
Sequential algorithm on one PC	1780.6	456.5	----
Parallel and distributed one-level architecture (master – 5 slave nodes) on 6 PCs	343.4	5.8	78.7
Parallel and distributed one-level architecture (master – 10 slave nodes) on 11 PCs	408.7	3.1	147.3
Parallel and distributed one-level architecture (master – 15 slave nodes) on 16 PCs	525.8	2.2	207.5
Parallel and distributed one-level architecture (master – 20 slave nodes) on 21 PCs	654	1.6	285.3
Parallel and distributed one-level architecture (master – 25 slave nodes) on 26 PCs	797.1	1.4	326.1

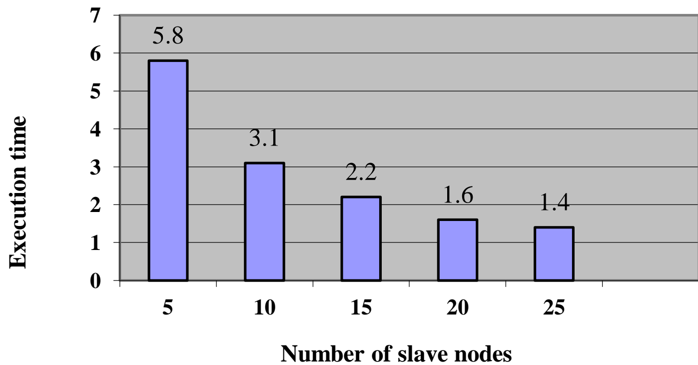


Figure 4.3 Parallel execution time based on the total number of slave workstations using the master – N slave nodes approach

To be able to predict the speedup for any number of machines available, the following predictive equation was developed for calculating parallel execution time based on the number of nodes attached to the master node (see Figure 4.4).

$$\text{Parallel execution} = (0.01 * \text{n}) + \left(\frac{0.18}{1000}\right) * (\frac{\text{m}}{\text{n}})$$

where n is the number of nodes attached to a master node, and m is the maximum number of features allocated to any slave nodes.

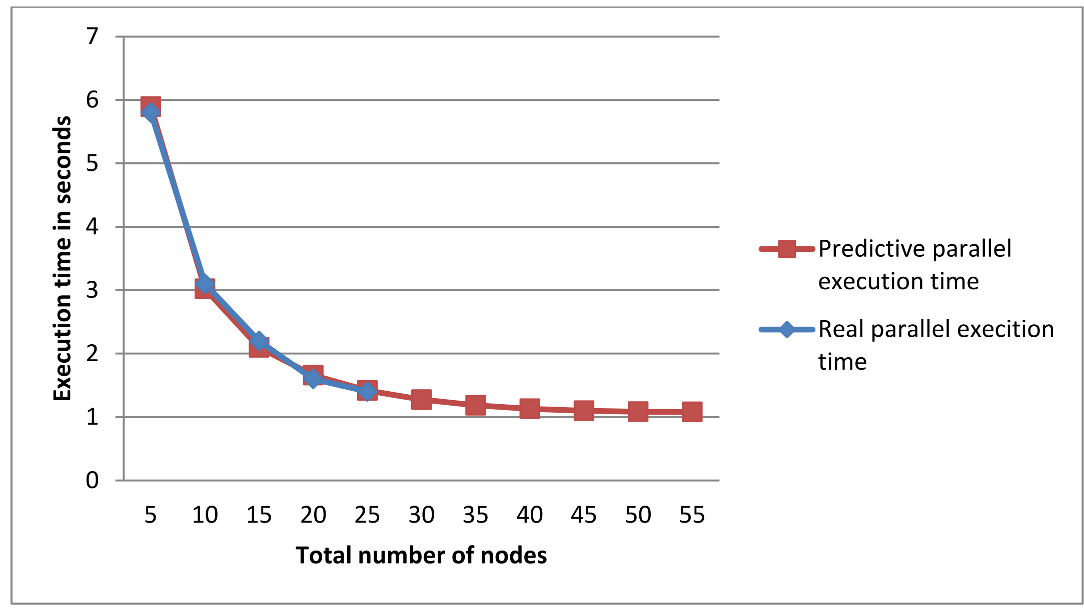


Figure 4.4 Real and predictive parallel execution time based on the total number of slave workstations using the master – N slave nodes approach

4.2 Pruning Techniques Testing Results

To detect objects, such as faces in an image, this framework follows the Viola and Jones algorithm, where features were extracted and used for face detection. Once these features were applied to an image, it was noticed that multiple rectangles were drawn on the same detected face. Also some false positive results were shown where a non-face was detected as a face, as shown in Figure 4.5. To handle these cases, a pruning algorithm has been used to filter out duplicate results and eliminate some of the false positive rectangles.

In this framework, three pruning techniques have been tested to see which one can achieve the best results.

4.2.1 Pruning by Clustering

For pruning purposes, each inserted image is converted into a two-dimensional array, such that wherever there is a rectangle drawn, a number “1” is inserted as a value on the index (x, y) that matches the (x, y) of the image, as seen in Figure 4.6 and Figure 4.7.

Figure 4.5 Multiple rectangles on each face and some false positive rectangles

After having obtained a two-dimensional array that has a 1-value at each candidate face position in the image, a scan window goes over the array to cluster any

group of 1-values that are next to each other using the 8-neighbors approach (Figure 4.8). As a result, each clustered group will be assigned a unique ID, as shown in Figure 4.7.

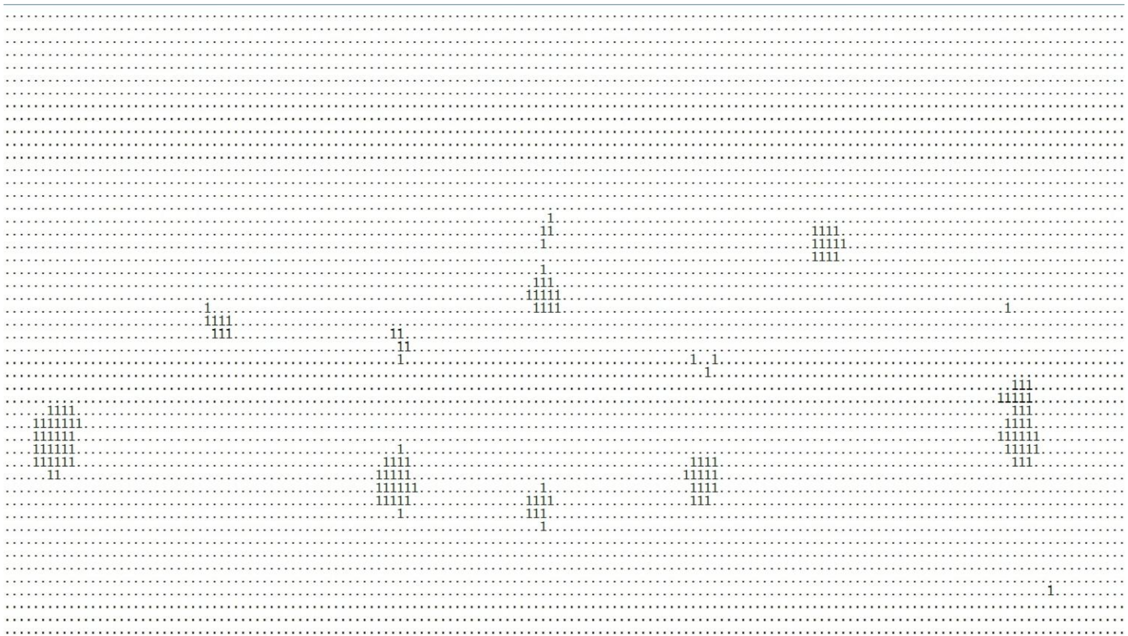


Figure 4.6 Each 1-value represents a rectangle in the real image in Figure 4.5

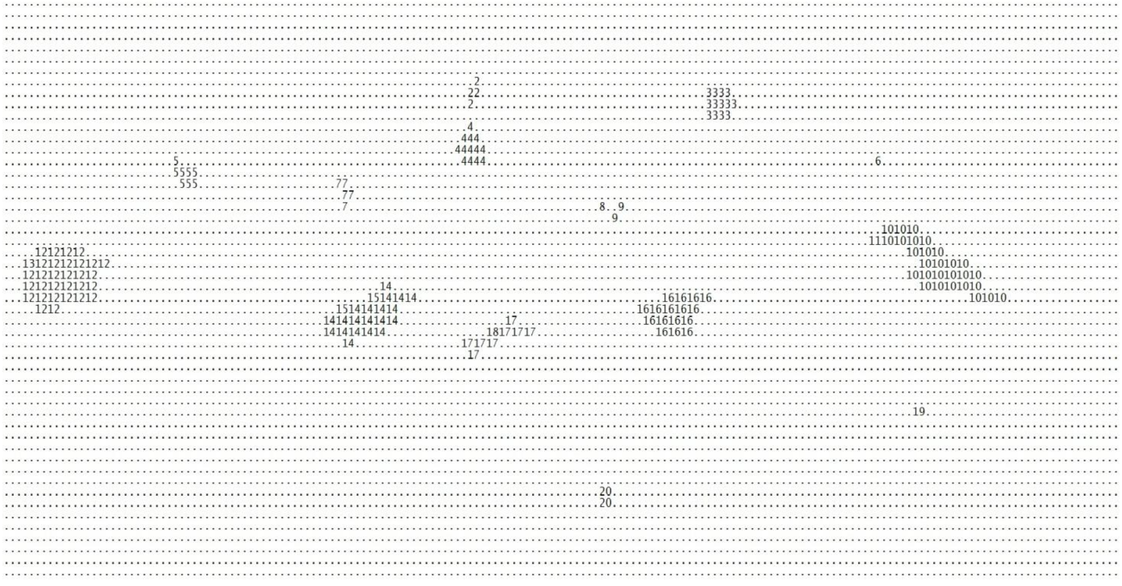


Figure 4.7 A unique ID for each group of 1-values

$P_{i-1,j-1}$	$P_{i-1,j}$	$P_{i-1,j+1}$
$P_{i,j-1}$	P_{ij}	$P_{i,j+1}$
$P_{i+1,j-1}$	$P_{i+1,j}$	$P_{i+1,j-1}$

Figure 4.8 The 8 neighbors of the center point

Pseudocode for pruning by clustering:

- Initialize a new two-dimensional array for IDs (called ID) that has the same size as the two-dimensional array of 1-values (called ONE)
- Set Id = 2
- For i = 1.. height

  For j = 1..width

    If $ONE_{i,j}$ = 1

      If $ID_{i-1,j-1} + ID_{i-1,j} + ID_{i-1,j+1} + ID_{i,j-1} + ID_{i,j+1} + ID_{i+1,j-1} + ID_{i+1,j} + ID_{i+1,j-1} > 0$

        If $ID_{i,j-1}$!= 0

          $ID_{i,j} = ID_{i,j-1}$

        If $ID_{i-1,j-1}$!= 0

          $ID_{i,j} = ID_{i-1,j-1}$

        If $ID_{i-1,j}$!= 0

          $ID_{i,j} = ID_{i-1,j}$

        If $ID_{i-1,j+1}$!= 0

$ID_{i,j} = ID_{i-1,j+1}$

If $ID_{i,j+1}$!= 0

$ID_{i,j} = ID_{i,j+1}$

If $ID_{i+1,j-1}$!= 0

$ID_{i,j} = ID_{i+1,j-1}$

If $ID_{i+1,j}$!= 0

$ID_{i,j} = ID_{i+1,j}$

If $ID_{i+1,j-1}$!= 0

$ID_{i,j} = ID_{i+1,j-1}$

Else

$ID_{i,j}$ = Id

Id++

- Now, all 1-values next to each other should have the same ID
- Obtain the center point of each clustered group by obtaning the average of all (x, y) which have the same ID
- If the size of any group is less than 3

  Delete it
- Draw a rectangle based on the center point of each group

4.2.2 Pruning by Distance and Number of Neighbors

In this technique, a scan window goes over the two-dimensional array and applies the 8-neighbors approach to only keep the 1-values for each cell that are surrounded by the maximum number of 1-values. From this point, the Euclidean distance equation

between two points is used to prune groups by keeping only the 1-values that have the highest number of neighbors, where the distance should be less than 20 because the classifier was trained based on images of 24x24 pixels.

Pseudocode for pruning by distance and number of neighbors:

- Initialize a new two-dimensional array for number of neighbors (called N), which has the same size as the two-dimensional array of 1-values (called ONE)
- For i = 1.. height

  For j = 1..width

  If $ONE_{i,j}$ = 1

  $N_{j,i}$ =$ONE_{j,i}$ +$ONE_{j-1,i-1}$+$ONE_{j-1,i}$+$ONE_{j-1,i+1}$+$ONE_{j,i-1}$+$ONE_{j,i+1}$+ $ONE_{j+1,i-1}$+$ONE_{j+1,i}$ +$ONE_{j+1,i-1}$

  If $N_{j,i}$ < 3

  $N_{j,i}$ = 0
- Use the distance equation to get the distance between two points
- For each point:

  If the distance between one point and another one is greater than 0 and is less than or equal to 20

  Keep the point which has the highest value and delete the other one
- Draw a rectangle based on each of the remaining points

4.2.3 Pruning Using Recursive Elimination of Neighbors

In this technique, another way of pruning was implemented using a recursive elimination of neighbors based on overlapping sub windows. An ID is given to a sub-

window, and then based on the overlapping of the center point of that sub-window with the center point of any other sub-windows, all sub-windows which overlap at the center point are assigned the same ID. This process is repeated recursively. Among each group where each has a unique group ID, only one sub-window is selected based on the total number of surrounding neighbors. If the total number of neighbors is the same for two or more sub-windows, then only the sub-window that has the greatest value is selected. The value is obtained from the AdaBoost algorithm that was used to build classifiers at each stage of the cascade. Any sub-window will detect an object when the sum of alphas of all weak classifiers in the sub-window is greater than or equal to 0.5 multiplied by the sum of alphas that was obtained during training classifiers. This result is the threshold that will determine objects from non-objects, as seen in the following equation.

$$h(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \geq 0.5 \sum_{t=1}^{T} \alpha_t \\ 0, & \text{otherwise} \end{cases}$$

However, instead of returning 1 in the above equation, the sum of alphas of all weak classifiers in the sub-window will be returned as the value that will be used in pruning. As a result, whenever the number of neighbors of sub-windows is equal within the same group, the one that has the highest value will be selected and drawn around the detected object.

Pseudocode for pruning using recursive elimination of neighbors:

- Match each sub-window with other sub-windows
    - Only keep sub-windows that totally intersect with another sub-window and have the highest number of neighbors
- Match each of the remaining sub-windows with the other sub-windows

- Give sub-window an ID
- Recursively give the same ID to any sub-window intersecting with a sub-window having an ID, whenever the intersection occurs at the center point part of any sub-window, where the center point size is about one fourth of the sub-window size.

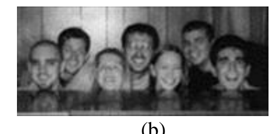
- Among each group of sub-windows, where each group has a unique ID
    - Select one sub-window from each group based on total number of neighbors. Where the total number of neighbors is equal for two or more sub-windows, select one sub-window based on sub-window value.
- Draw rectangles for all the remaining sub-windows that had 4 or more neighbors at the start

In the next section, we provide experimental results on pruning in face detection in different images where traditional face detection results in difficult pruning.

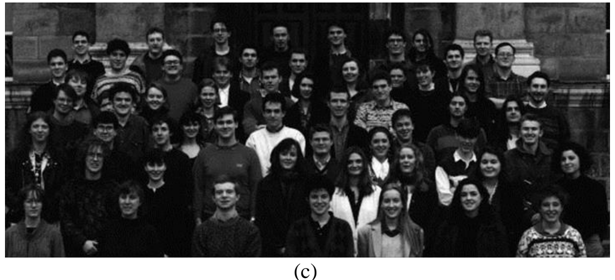
4.2.4 Results of Pruning Techniques

Figure 4.9 (a, b, and c) represents each image before the detection and pruning process, and Figure 4.10 (a, b, and c) represents face detection results before pruning.

(a)

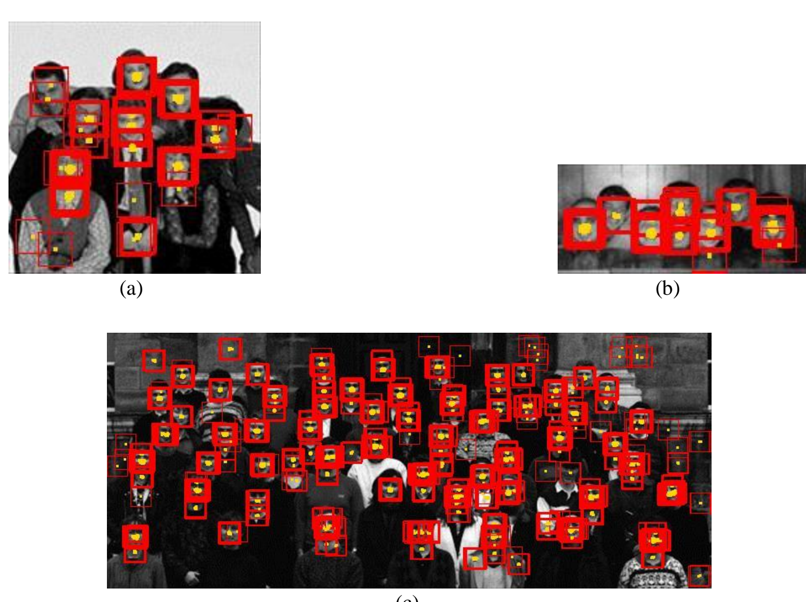


(b)



(c)

Figure 4.9 (a, b, c) example of three images used for testing



(a) (b)

(c)

Figure 4.10 (a, b, c) example of three images after detecting the expected faces and false positives

4.2.4.1 Results of Pruning by Clustering

The following table provides the results of each image after applying pruning by the clustering technique, which depends on giving an ID to each group.

Table 4.6 Example of three images after pruning based on ID

Before pruning	**After pruning**

4.2.4.2 Results of Pruning by Distance and Number of Neighbors

The following table provides the results of each image after applying pruning by distance and number of neighbors.

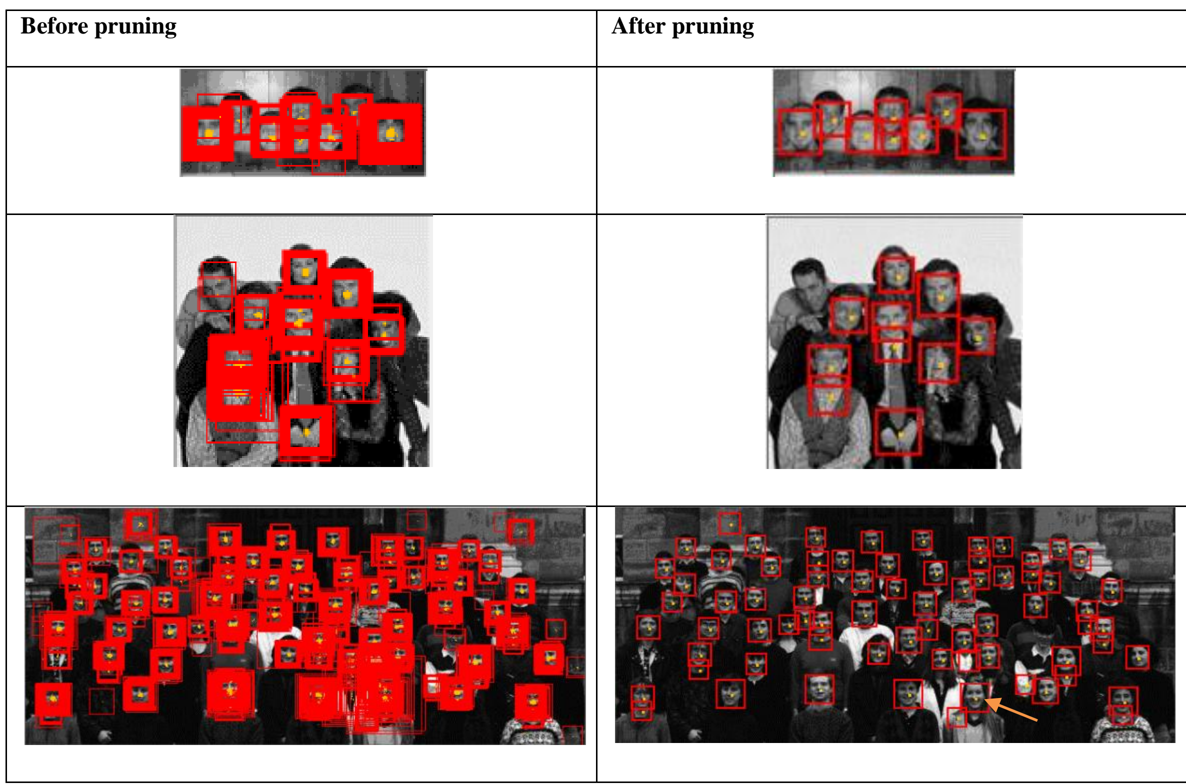
Table 4.7 Example of three images after pruning based on distance and number of neighbors

Before pruning	**After pruning**

4.2.4.3 Results of Pruning using Recursive Elimination of Neighbors

The following table provides the results of each image after applying pruning using the recursive elimination of neighbors.

Table 4.8 Example of three images after pruning based on recursive elimination of neighbors



By looking at the previous figures, it is noticed that the results of the last two algorithms are better than the first one. Comparing the last two algorithms to each other, it shows that in one case, indicated with an arrow in the last image on the right side of both Table 4.7 and Table 4.8, the last algorithm was able to select the correct rectangle and eliminate the wrong one, but for the same case when the second algorithm was applied, the correct rectangle was deleted after pruning and the wrong one was kept.

Because in the last algorithm pruning is based on two strong criteria, the number of neighbors and the value of the rectangle, the third algorithm (pruning using recursive elimination of neighbors) is the best algorithm for pruning.

4.3 A Near Real-time Re-training to Fix Results of Missed Objects and False Positives

After building a parallel and distributed framework using parallel AdaBoost and web services, re-training a classifier to delete false positive examples or detect missed examples that were not detected based on the original classifiers becomes much easier. As a result, the missed objects will be detected and false positive objects will be deleted after a short training. The re-training occurs only in the last stage.

4.3.1 The Proposed Framework Results for Cars

Table 4.9 provides some cases of missed objects, Table 4.10 provides some examples of false positives, and Table 4.11 shows correct detection based on the existing trained classifier, when the proposed framework was tested on images of cars. The stages of the classifier were built based on a small optimized training set. For cars, the training set size was 550 cars and 500 non-cars, and the optimized training set size was around 60 cars and 60 non-cars. The proposed framework was tested on one master and five slave workstations. Some images from the UIUC Image Database were used in testing [4, 5]. As a result of a short re-training on the last stage of the classifier, the false positives were weaker and the missed cars were stronger; the previously missed cars should be detected and the false positives should not be detected again as cars.

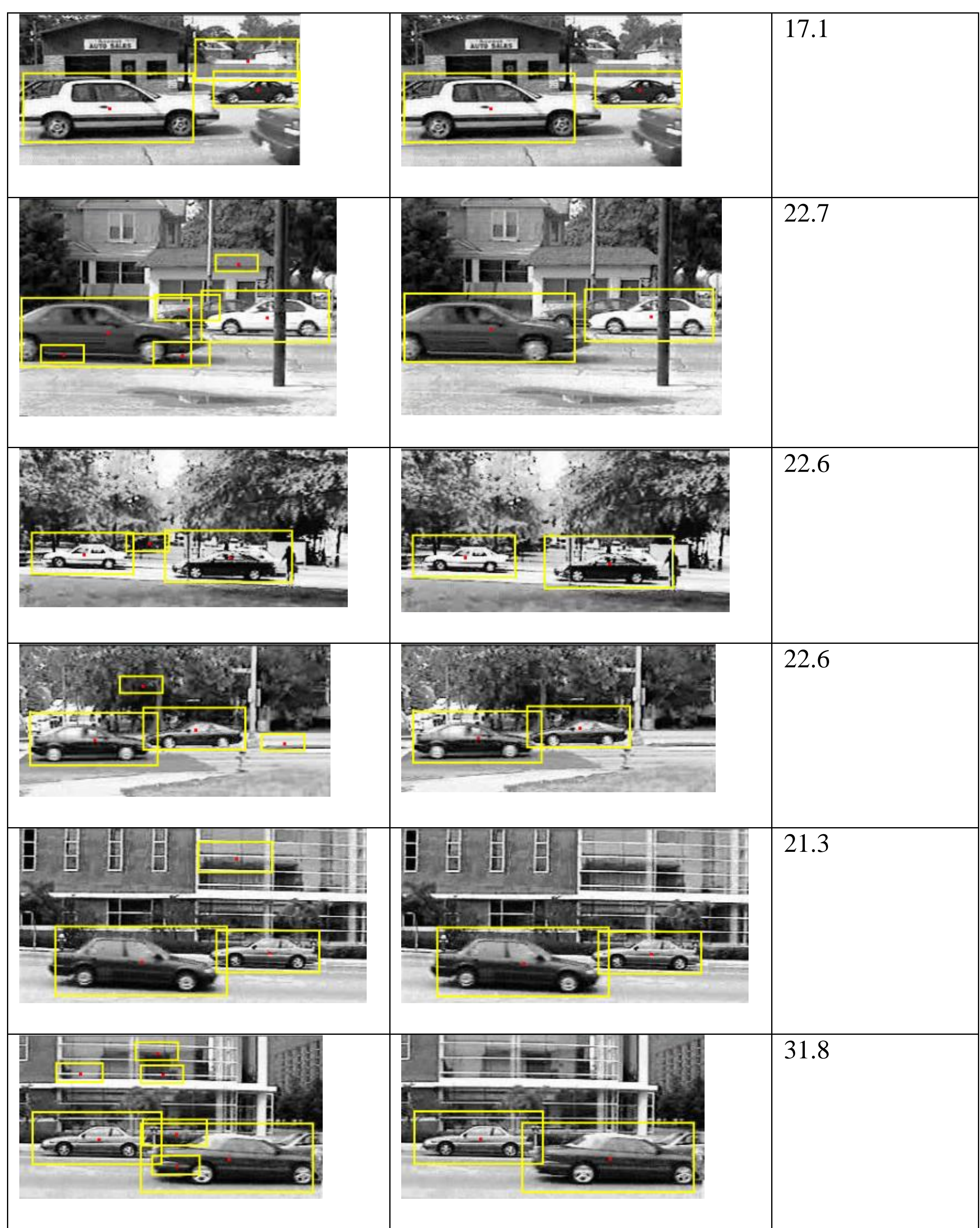
Table 4.9 Results of re-training to detect a missed car

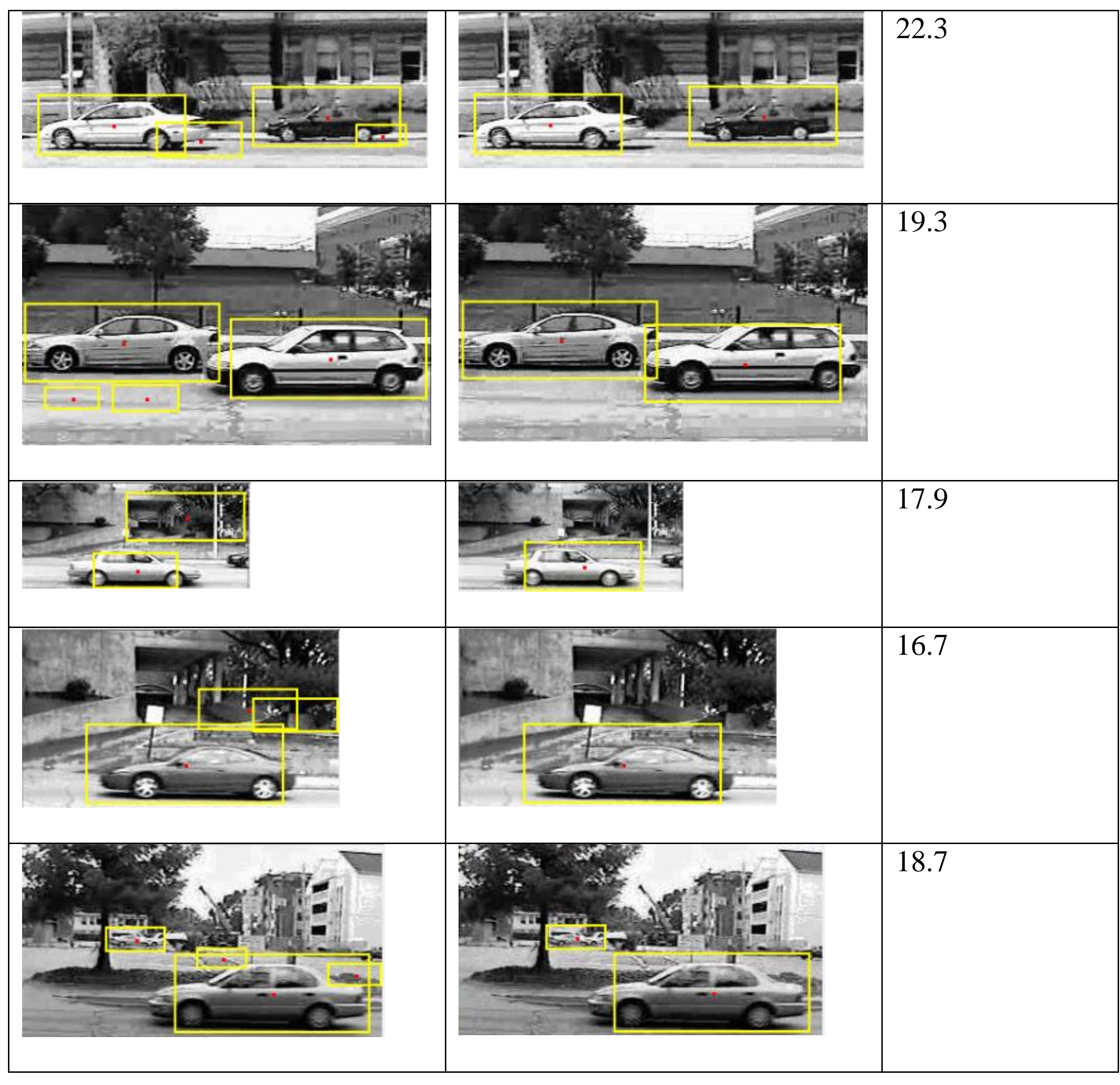
Output	**After re-training**	**Re-training time (seconds)**
		22.4
		23
		23.1
		22.4
		23.9

Table 4.10 Results of re-training to delete false positives

Output	After re-training	Re-training time (seconds)
		19.2
		28.5
		22.4
		21.2
		21.8
		23.4

		16.9
		19.7
		18
		21.3
		16.1
		18.3
		17.6


AUTO SALES
17.1
22.7
22.6
22.6
21.3
31.8



22.3

19.3

17.9

16.7

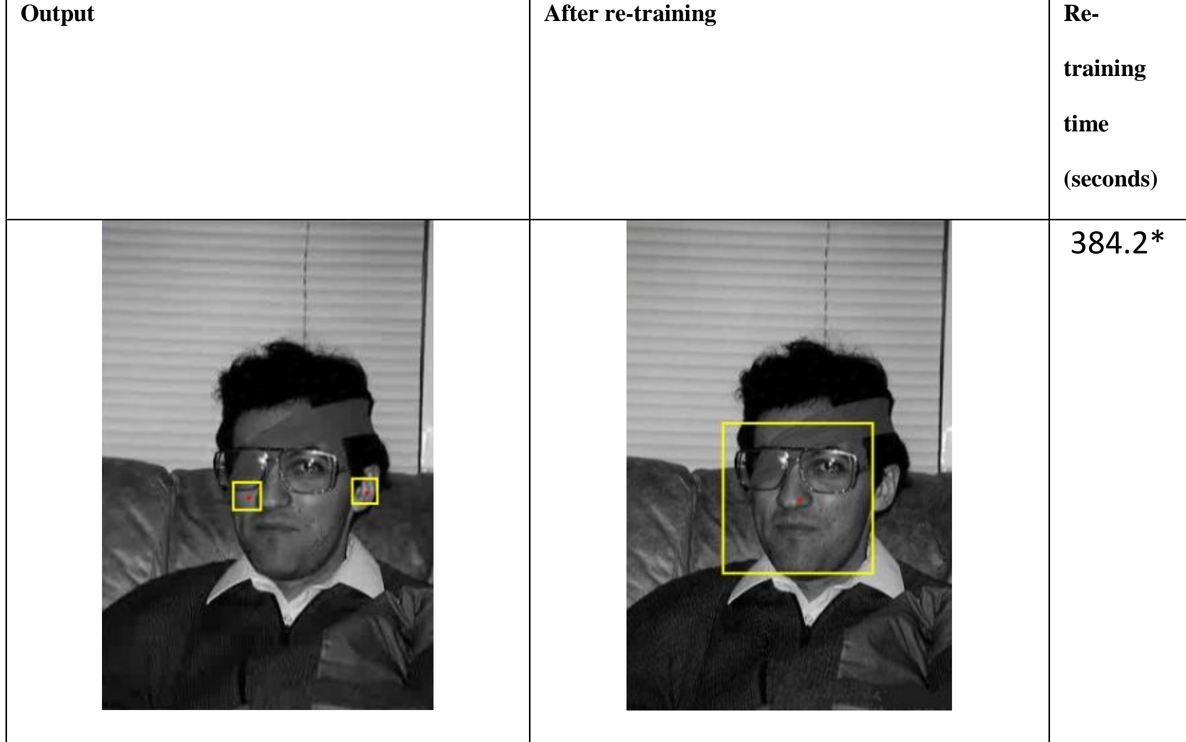
18.7

Table 4.11 Results of detecting cars by the proposed framework without re-training

4.3.2 The Proposed Framework Results for Faces

Table 4.12 provides some cases where faces were not detected, Table 4.13 shows false positives, and Table 4.14 demonstrates correct detection based on the existing trained classifier. The stages of the classifier were built based on a small optimized training set. For faces, the optimized training set size was around 800 faces and 800 non-faces. The proposed framework was tested on one master and five slave node workstations. Some images from the public Carnegie Mellon face database were used in testing. As a result of a short re-training on the last stage of the classifier, the false positives were weaker and the missed faces were stronger; the previously missed faces should be detected and the false positives should not be detected again as faces.

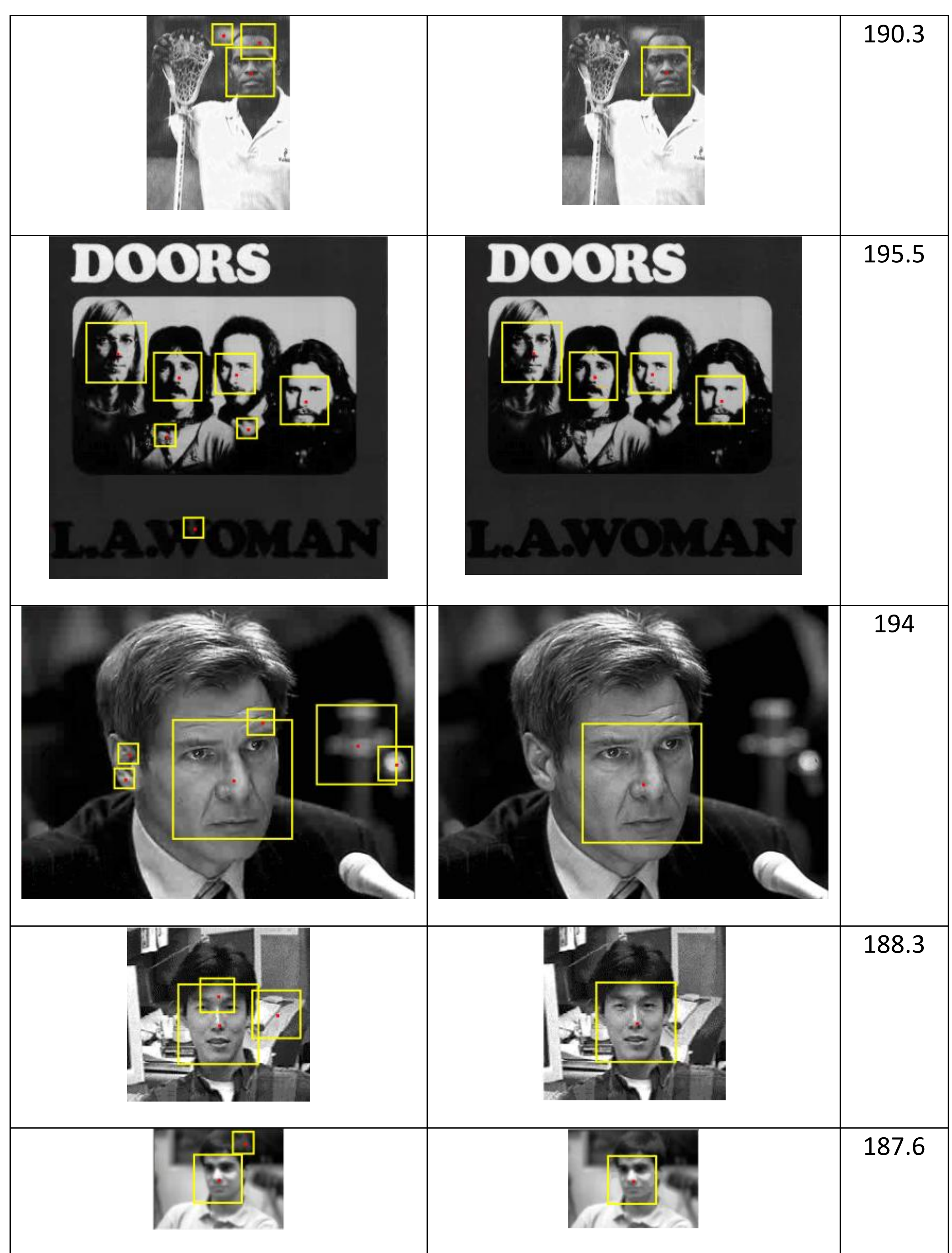
Table 4.12 Results of re-training to detect a missed face

Output	**After re-training**	**Re-training time (seconds)**
		384.2*

*Total time after two attempts

Table 4.13 Results of re-training to delete false positives

Output	**After re-training**	**Re-training time (seconds)**
		190.8
		190.6
		208.3
		194.9


190.3
DOORS
L.A.WOMAN
DOORS
L.A.WOMAN
195.5
194
188.3
187.6

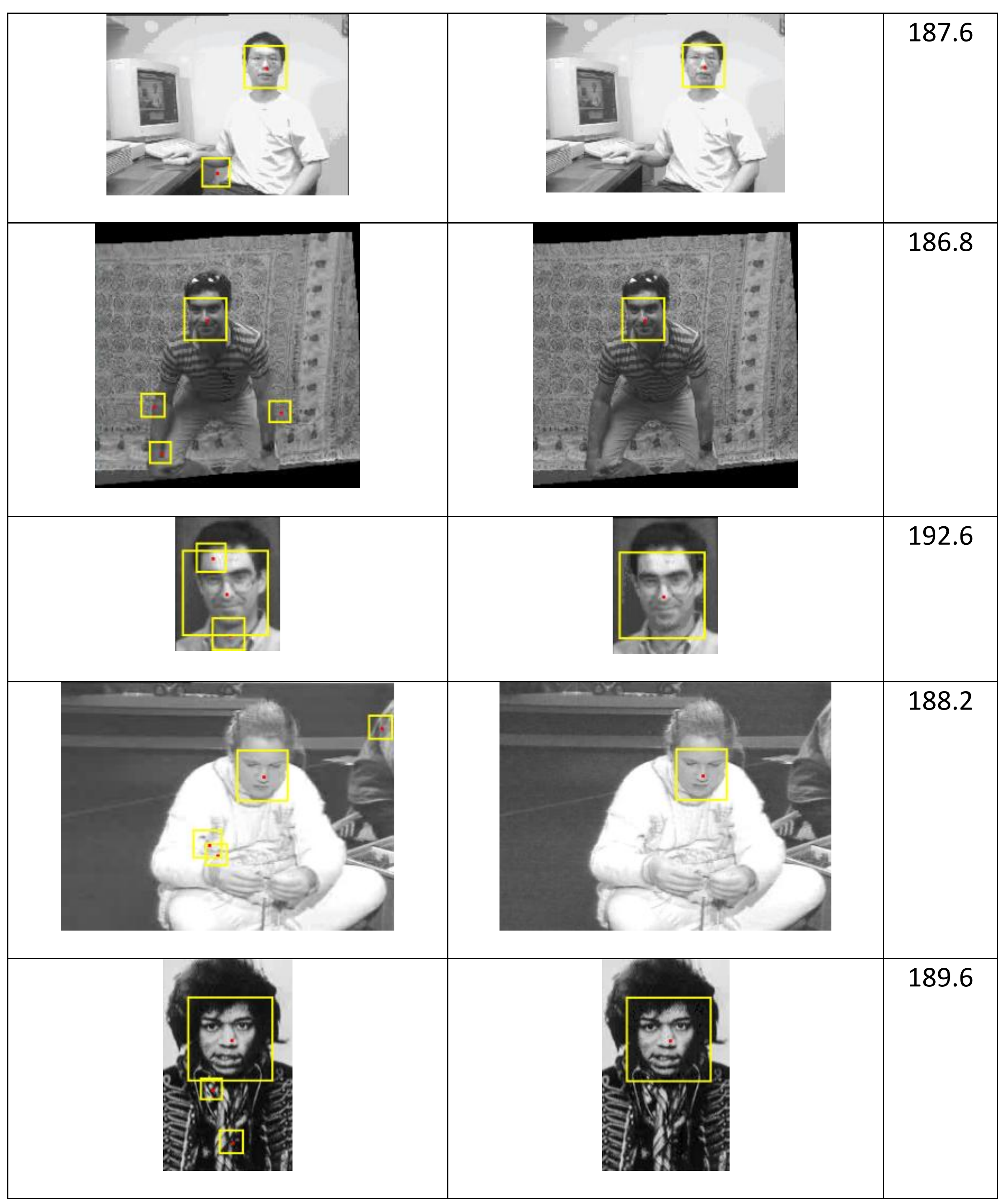
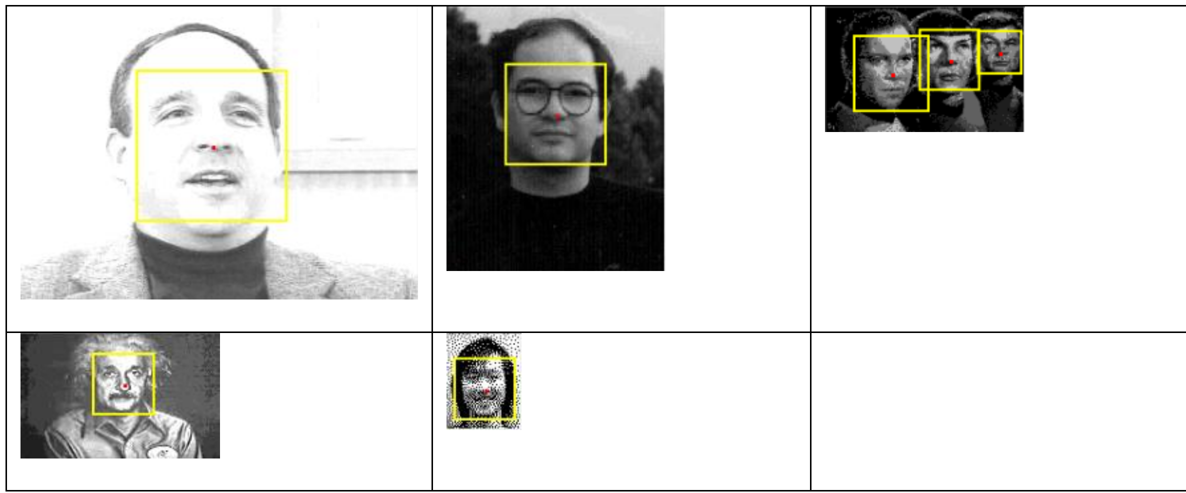

187.6
186.8
192.6
188.2
189.6

Table 4.14 Results of detecting faces by the proposed framework without re-training



4.4 Power Analysis

This work proposes the use of web services and parallel execution to reduce the average execution time of extracting one feature. This section summarizes the statistical power analysis performed with the goal of testing the alternative hypothesis and estimating the sample size used for the experiment based on our development of the following approaches:

1. Parallel on one machine that only uses TPL
2. Web services and parallel execution on one-level hierarchy
3. Web services and parallel execution on two-level hierarchy
4. Web services and parallel execution on one-level hierarchy and N slave nodes

The power of a statistical test is defined as the probability that the null hypothesis will be rejected by the test when the null hypothesis is false and confirming the

alternative hypothesis when the alternative hypothesis is true [70]. Two opposing hypotheses are given as the following:

- Null Hypothesis Ho (same, equal, no diff, and no change)
- Alternate Hypothesis Ha (complement of Ho)

The test will have one of two conclusions: either accept Ho or reject Ho. There are 3 types of tests:

- Right-tailed test (greater than)
- Left-tailed test (less than)
- Two-tailed test (not equal to)

The type of the test used is stated in Ha. The type of test is chosen based on what the researcher intends to show.

- The first statistical hypothesis for this work is as follows:

  The average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy of 6 workstations, each having a quad-core processor, is less than the average while using parallel execution on one workstation.

$$Ho \rightarrow \quad \mu_{w_p_o} \geq \mu_p$$
$$Ha \rightarrow \quad \mu_{w_p_o} < \mu_p$$

where $\mu_{w_p_o}$ is the average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy of 6 workstations and μ_p

is the average execution time of extracting one feature using parallel execution on one workstation.

In other words, the alternative hypothesis is that the average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy of 6 workstations is less than the average while using parallel execution on one workstation.

The significance of an observed difference is determined by the chosen Level of Significance (α). Commonly, 5% (0.05) or 1% (0.01) are used for α.

In order to determine the needed sample size (*n*), three parameters are used in this decision:

- The level of significance (1% or 5%) used (*α*)
- The required power (80%) of the test
- Effect size (*d*)

Table 4.15 [71] shows that the effect size will be "*d*", based on a large set using a t-test on Means calculation.

Table 4.15 Effect size values

	Effect size Index	**Small**	**Medium**	**Large**
t-test on Means	***d***	0.20	0.50	0.80
t-test on Correlations	*r*	0.10	0.30	0.50
F-test ANOVA	*f*	0.10	0.25	0.40
F-test regression	f^2	0.02	0.15	0.35
Chi-Square Test	*w*	0.10	0.30	0.50

$$\text{Sample size (n)} = \frac{N \times p(1-p)}{[[(N-1)\times(d^2 \div z^2)]+p(1-p)]}$$

where, *N*=6, *P*=80%, *d*=5% and *z*=1.96, then the sample size (*n*) can be calculated as:

n=(6*0.8(.2))/((5)*(.05)^2/1.96^2)+0.8*0.2 = 6.

The parameters below are computed in order to decide whether to accept or reject the hypotheses.

$$Mean\ (\bar{X}) = \frac{\sum x}{n} = 24.42$$

$$Variance\ (\sigma^2) = \frac{\sum(x-\bar{x})^2}{n} = 0.0492$$

$$Standard\ Deviation\ (\sigma) = \sqrt[2]{\sigma^2} = 0.2217$$

$$Degree\ of\ freedom\ (d.f) = n - 1 = 5$$

$$Confidence\ Level\ (1-\propto) = 95\%$$

$$Significance\ (\propto) = 5\%$$

$$Critical\ t = TINV\ (\propto, d.f) = 2.571$$

$$Standard\ Errors\ (Sx) = \frac{\sigma}{\sqrt{n}} = 0.0905$$

$$Lower\ limit = \bar{X} - Critical\ t * Sx = 24.19$$

$$Upper\ limit = \bar{X} + Critical\ t * Sx = 24.66$$

$$Hypothesis\ (Ho) = 50\%$$

$$t\ value = \frac{\bar{X} - Ho}{Sx} = 264.3$$

To reach our conclusion, the t value and critical t value are used. If the t value is greater than the critical t (probability Ho is true is low), Ho is rejected. In this test: t value (264.3) > critical t (2.571). This means Ho is rejected and Ha is accepted.

Using the power analysis reveals that a sample set of 6 is sufficient to prove that the use of web services and parallel execution on one-level hierarchy on 6 workstations yielded a reduction in the average execution time of extracting one feature, compared to the average execution time yielded by using parallel on one workstation. The sample used is sufficient to prove the stated hypothesis with a high degree of accuracy and confidence.

- The second statistical hypothesis for this work is as follows:

  The average execution time of extracting one feature using web services and parallel execution on a two-level hierarchy of 21 workstations, each having a quad-core processor, is less than the average while using web services and parallel execution on a one-level hierarchy of 6 workstations.

  Ho → $\mu_{w_p_t} \geq \mu_{w_p_o}$
  Ha → $\mu_{w_p_t} < \mu_{w_p_o}$

  where $\mu_{w_p_t}$ is the average execution time of extracting one feature using web services and parallel execution on a two-level hierarchy of 21 workstations and $\mu_{w_p_o}$ is the average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy of 6 workstations.

In other words, the alternative hypothesis is that the average execution time of extracting one feature using web services and parallel execution on a two-level hierarchy of 21 workstations is less than the average while using web services and parallel execution on a one-level hierarchy of 6 workstations.

$Sample\ size\ (n) = 20$

$Mean\ (\bar{X}) = \frac{\sum x}{n} = 6.4$

$Variance\ (\sigma^2) = \frac{\sum(x-\bar{x})^2}{n} = 0.0368$

$Standard\ Deviation\ (\sigma) = \sqrt[2]{\sigma^2} = 0.1919$

$Degree\ of\ freedom\ (d.f) = n - 1 = 19$

$Confidence\ Level\ (1-\propto) = 95\%$

$Significance\ (\propto) = 5\%$

$Critical\ t = TINV\ (\propto, d.f) = 2.093$

$Standard\ Errors\ (Sx) = \frac{\sigma}{\sqrt{n}} = 0.0429$

$Lower\ limit = \bar{X} - Critical\ t * Sx = 6.31$

$Upper\ limit = \bar{X} + Critical\ t * Sx = 6.49$

$Hypothesis\ (Ho) = 50\%$

$t\ value = \frac{\bar{X} - Ho}{Sx} = 137.5$

To reach our conclusion, the t value and critical t value are used. If the t value is greater than the critical t (probability Ho is true is low), Ho is rejected. In this test: t value (137.5) > critical t (2.093). This means Ho is rejected and Ha is accepted.

Using the power analysis reveals that a sample set of 20 is sufficient to prove that the use of web services and parallel execution on a two-level hierarchy of 21 workstations yielded a reduction in the average execution time of extracting one feature, compared to the average execution time yielded by using web services and parallel execution on a one-level hierarchy of 6 workstations. The sample used is sufficient to prove the stated hypothesis with a high degree of accuracy and confidence.

- The third statistical hypothesis for this work is as follows:

  The average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy and N slave nodes (for one master and 25 slave node workstations), each having a quad-core processor, is less than the average while using web services and parallel execution on a two-level hierarchy of 21 workstations.

$$\text{Ho} \rightarrow \quad \mu_{w_p_o_N} \geq \mu_{w_p_t}$$
$$\text{Ha} \rightarrow \quad \mu_{w_p_o_N} < \mu_{w_p_t}$$

  where $\mu_{w_p_o_N}$ is the average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy and N slave workstations (for a total of 26 workstations) and $\mu_{w_p_t}$ is the average execution time of extracting one feature using web services and parallel execution on a two-level hierarchy of 21 workstations.

In other words, the alternative hypothesis is that the average execution time of extracting one feature using web services and parallel execution on a one-level hierarchy and N slave workstations (for a total of 26 workstations) is less than the average while using web services and parallel execution on a two-level hierarchy of 21 workstations.

$Sample\ size\ (n) = 24$

$Mean\ (\bar{X}) = \frac{\sum x}{n} = 1.41$

$Variance\ (\sigma^2) = \frac{\sum(x-\bar{x})^2}{n} = 0.022$

$Standard\ Deviation\ (\sigma) = \sqrt[2]{\sigma^2} = 0.1484$

$Degree\ of\ freedom\ (d.f) = n - 1 = 23$

$Confidence\ Level\ (1-\propto) = 95\%$

$Significance\ (\propto) = 5\%$

$Critical\ t = TINV\ (\propto, d.f) = 2.069$

$Standard\ Errors\ (Sx) = \frac{\sigma}{\sqrt{n}} = 0.0303$

$Lower\ limit = \bar{X} - Critical\ t * Sx = 1.35$

$Upper\ limit = \bar{X} + Critical\ t * Sx = 1.48$

$Hypothesis\ (Ho) = 50\%$

$t\ value = \frac{\bar{X} - Ho}{Sx} = 30.13$

To reach our conclusion, the t value and critical t value are used. If the t value is greater than the critical t (probability Ho is true is low), Ho is rejected. In this test: t value (30.13) > critical t (2.069). This means Ho is rejected and Ha is accepted.

Using the power analysis reveals that a sample set of 24 is sufficient to prove that the use of web services and parallel execution on a one-level hierarchy and N slave workstations (total of 26 workstations) yielded a reduction in the average execution time of extracting one feature, compared to the average execution time yielded by using web services and parallel execution on a two-level hierarchy of 21 workstations. The sample used is sufficient to prove the stated hypothesis with a high degree of accuracy and confidence.

4.5 Summary

Using parallel AdaBoost, as well as web services, we succeeded in making the re-training process faster. As a result, whenever an object is not detected based on the existing trained classifier, a fast re-training can occur in near real time to help in detecting objects that are hard to detect using regular trained classifiers, and deleting false positives which are not easily deleted using the regular trained classifiers.

CHAPTER 5: CONCLUSION

We have developed an adaptive near real-time re-trainable object detection framework based on the AdaBoost algorithm. Such a system is extremely useful in real life situations where a system fails to detect a case or encounters a false positive in the detection process. The re-training time of Adaboost as implemented by Viola and Jones is on the order of days. We developed a novel hybrid parallel and distributed implementation of the AdaBoost algorithm that exploits the multiple cores in a CPU via lightweight threads, and also uses multiple machines via web services to achieve extremely efficient training and re-training time. We demonstrated a nearly linear acceleration based on the number of processors available to us, and could accomplish the learning of a feature in the AdaBoost algorithm within a few seconds. We achieved a speed 326.1 times faster than sequential AdaBoost on one workstation by using parallel AadBoost on 26 workstations, each having a quad-core processor, resulting in a learning time of only 1.4 seconds per feature. We also developed a method for determining a small optimized training set that still achieves 100% detection on the original dataset, which helps to further reduce the re-training time. The significant speedup of the proposed parallel and distributed AdaBoost granted us the ability to extract the optimized subset from the entire set. The optimized subset was yielded by replacing the images of lowest weight with other images, until getting a small subset that could achieve high

detection rates. Finally, we encapsulated all of the above capabilities into an adaptive re-trainable object detection framework and tested its effectiveness on many different images where object detection through existing classifiers failed. The system could be re-trained to do correct detection and removal of false positives within minutes or seconds, depending on optimized subset size.

5.1 Future Directions

Improvements can be made to the framework by using a graphics processing unit (GPU) together with a CPU to accelerate the framework computation process. Also, the framework capabilities can be adjusted to work directly in the cloud where the computing process relies on sharing computing resources, rather than using local workstations to handle applications.

REFERENCES

[1] I. Palit and C. K. Reddy, "Scalable and Parallel Boosting with MapReduce," IEEE Transactions on Knowledge and Data Engineering, vol. 24, pp. 1904-1916, 2012.

[2] P. Viola and M. Jones, "Robust real-time face detection," in Proceedings of the Eighth IEEE International Conference on Computer Vision (ICCV'01), 2001, p. 747.

[3] D. C. Lee, "Boosted Classifier for Car Detection," ed: unpublished, http://www.cs.cmu.edu/~dclee, 2007.

[4] S. Agarwal, et al., "Learning to detect objects in images via a sparse, part-based representation," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, pp. 1475-1490, 2004.

[5] S. Agarwal and D. Roth, "Learning a Sparse Representation for Object Detection," presented at the Proceedings of the 7th European Conference on Computer Vision-Part IV, 2002.

[6] B. Weber, "Generic Object Detection using AdaBoost," UCSC Technical Report, 2008.

[7] Y. Ming-Hsuan, et al., "Detecting faces in images: a survey," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, pp. 34-58, 2002.

[8] E. Osuna, et al., "Training support vector machines: an application to face detection," in Proceedings of Computer Vision and Pattern Recognition, 1997, pp. 130-136.

[9] H. A. Rowley, et al., "Neural network-based face detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, pp. 23-38, 1998.

[10] B. Moghaddam, et al., "Bayesian face recognition," Pattern Recognition, vol. 33, pp. 1771-1782, 2000.

[11] E. Hjelmås and B. K. Low, "Face detection: A survey," Computer vision and image understanding, vol. 83, pp. 236-274, 2001.

[12] M.-H. Yang, et al., "Detecting faces in images: A survey," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, pp. 34-58, 2002.

[13] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in Proceedings of Computer Vision and Pattern Recognition (CVPR 2001), 2001, pp. I-511-I-518 vol.1.

[14] P. Viola and M. J. Jones, "Robust Real-Time Face Detection," Int. J. Comput. Vision, vol. 57, pp. 137-154, 2004.

[15] C. Zhang and Z. Zhang, "A survey of recent advances in face detection," Technical report, Microsoft Research, 2010.

[16] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," Journal of Computer and System Sciences, vol. 55, pp. 119-139, 1997.

[17] P. Buhlmann and B. Yu, "Special Invited Paper. Additive Logistic Regression: A Statistical View of Boosting: Discussion," The Annals of Statistics, pp. 377-386, 2000.

[18] P. L. Bartlett and M. Traskin, "AdaBoost is Consistent," Journal of Machine Learning Research, vol. 8, pp. 2347-2368, 2007.

[19] K. Levi and Y. Weiss, "Learning object detection from a small number of examples: the importance of good features," in Proceedings of Computer Vision and Pattern Recognition (CVPR 2004), 2004, pp. II-53-II-60 Vol.2.

[20] Z. Guo, et al., "A Fast Algorithm of Face Detection for Driver Monitoring," in Proceedings of the International Conference on Intelligent Systems Design and Applications, 2006, pp. 267-271.

[21] T. Theocharides, et al., "A parallel architecture for hardware face detection," in Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006, p. 452.

[22] R. Lienhart, et al., "Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection," in Proc. DAGM 25th Pattern Recognition Symp., 2003, pp. 297-304.

[23] R. Schapire and Y. Singer, "Improved Boosting Algorithms Using Confidence-rated Predictions," Machine Learning, vol. 37, pp. 297-336, 1999.

[24] Z. Huang and X. Shi, "A distributed parallel AdaBoost algorithm for face detection," in Proceedings of Intelligent Computing and Intelligent Systems (ICIS), 2010, pp. 147-150.

[25] K. Zeng, et al., "Parallization of Adaboost Algorithm through Hybrid MPI/OpenMP and Transactional Memory," in 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, 2011, pp. 94-100.

[26] A. Lazarevic and Z. Obradovic, "The distributed boosting algorithm," presented at the Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, California, 2001.

[27] A. Lazarevic and Z. Obradovic, "Boosting Algorithms for Parallel and Distributed Learning," Distributed and Parallel Databases, vol. 11, pp. 203-229, 2002.

[28] S. Merler, et al., "Parallelizing AdaBoost by weights dynamics," Computational Statistics & Data Analysis, vol. 51, pp. 2487-2498, 2007.

[29] V. Galtier, et al., "Implementation of the AdaBoost Algorithm for Large Scale Distributed Environments: Comparing JavaSpace and MPJ," in Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, 2009, pp. 655-662.

[30] G. Escudero, et al., "Boosting Applied to Word Sense Disambiguation," in Proceedings of ECML-00, 11th European Conference on Machine Learning, 2000, pp. 129-141.

[31] R. Busa-Fekete and B. Kégl, "Bandit-aided boosting," in OPT 2009: 2nd NIPS Workshop on Optimization for Machine Learning, 2009.

[32] G. Wu, et al., "MReC4.5: C4.5 Ensemble Classification with MapReduce," in Proceedings of the 4th ChinaGrid Annual Conference (ChinaGrid'09), 2009, pp. 249-255.

[33] B. Panda, et al., "PLANET: massively parallel learning of tree ensembles with MapReduce," Proceedings of the VLDB Endowment, vol. 2, pp. 1426-1437, 2009.

[34] W. Fan, et al., "The application of AdaBoost for distributed, scalable and on-line learning," presented at the Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining, San Diego, California, USA, 1999.

[35] S. Gambs, et al., "Privacy-preserving boosting," Data Mining and Knowledge Discovery, vol. 14, pp. 131-170, 2007.

[36] F. Lozano and P. Rangel, "Algorithms for parallel boosting," in Proceedings of the Fourth International Conference on Machine Learning and Applications, 2005, pp. 368-373.

[37] L. Breiman, "Bagging predictors," Machine Learning, vol. 24, pp. 123-140, 1996.

[38] A. L. Blum and R. L. Rivest, "Training a 3-node neural network is NP-complete," Neural Networks, vol. 5, pp. 117-127, 1992.

[39] D. R. Hush, "Training a sigmoidal node is hard," Neural Computation, vol. 11, pp. 1249-1260, 1999.

[40] R. Schapire, "The strength of weak learnability," Machine Learning, vol. 5, pp. 197-227, 1990.

[41] Y. Freund, "Boosting a weak learning algorithm by majority," Information and computation, vol. 121, pp. 256-285, 1995.

[42] Z.-H. Zhou, Ensemble Methods: Foundations and Algorithms: Chapman & Hall/CRC Press, 2012.

[43] R. E. Schapire and Y. Freund, Boosting: Foundations and Algorithms: The MIT Press, 2012.

[44] A. Mayr, et al., "The evolution of boosting algorithms-from machine learning to statistical modelling," Methods Inf Med, vol. 53, pp. 419 – 427, 2014.

[45] L. Breiman, "Bagging predictors," Mach. Learn., vol. 24, pp. 123-140, 1996.

[46] M. Sewell, "Ensemble methods," Relatório Técnico RN/11/02, University College London Departament of Computer Science, 2011.

[47] C. P. Papageorgiou, et al., "A General Framework for Object Detection," presented at the Proceedings of the Sixth International Conference on Computer Vision, 1998.

[48] K. Roebuck, Biochips: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors: Emereo Pty Limited, 2011.

[49] M. Kamel and A. Campilho, Image Analysis and Recognition: 10th International Conference, ICIAR, Aveiro, Portugal, June 26-28, 2013, Proceedings: Springer Publishing Company, Incorporated, 2013.

[50] R. Lienhart, et al., "Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection," in Pattern Recognition. vol. 2781, B. Michaelis and G. Krell, Eds., ed: Springer Berlin Heidelberg, 2003, pp. 297-304.

[51] R. Pless. Project3: Faces. Available: http://www.cs.wustl.edu/~pless/559/Projects/faceTrainingData.zip

[52] R. Lienhart and J. Maydt, "An extended set of Haar-like features for rapid object detection," in IEEE ICIP, 2002, pp. 900–903.

[53] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," presented at the Proceedings of the Second European Conference on Computational Learning Theory, 1995.

[54] P. Harrington, Machine Learning in Action: Manning Publications Company, 2012.

[55] R. Caruana and A. Niculescu-Mizil, "Data mining in metric space: an empirical analysis of supervised learning performance criteria," presented at the Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, Seattle, WA, USA, 2004.

[56] G. Rätsch, et al., "Soft Margins for AdaBoost," Machine Learning, vol. 42, pp. 287-320, 2001.

[57] R. A. Servedio, "Smooth boosting and learning with malicious noise," J. Mach. Learn. Res., vol. 4, pp. 633-648, 2003.

[58] J. K. Bradley and R. E. Schapire, "Filterboost: Regression and classification on large datasets," in Advances in Neural Information Processing Systems, 2007, pp. 185-192.

[59] M. Collins, et al., "Logistic regression, AdaBoost and Bregman distances," Machine Learning, vol. 48, pp. 253-285, 2002.

[60] C. Shen, et al., "Face detection from few training examples," in 15th IEEE International Conference on Image Processing, 2008, pp. 2764-2767.

[61] O. Tuzel, et al., "Region covariance: a fast descriptor for detection and classification," presented at the Proceedings of the 9th European conference on Computer Vision - Volume Part II, Graz, Austria, 2006.

[62] S. Z. Li and Z. Zhang, "FloatBoost learning and statistical face detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, pp. 1112-1123, 2004.

[63] J. Wu, et al., "Fast Asymmetric Learning for Cascade Face Detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 30, pp. 369-382, 2008.

[64] P. Minh-Tri and C. Tat-Jen, "Fast training and selection of Haar features using statistics in boosting-based face detection," in IEEE 11th International Conference on Computer Vision (ICCV), 2007, pp. 1-7.

[65] X. Li, et al., "Improving adaboost for classification on small training sample sets with active learning," The Sixth Asian Conference on Computer Vision ACCV, Korea, 2004.

[66] N. C. Oza, "Online bagging and boosting," in IEEE International Conference on Systems, Man and Cybernetics, 2005, pp. 2340-2345.

[67] P. M. Roth, et al., "On-line Conservative Learning for Person Detection," in 2nd Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005, pp. 223-230.

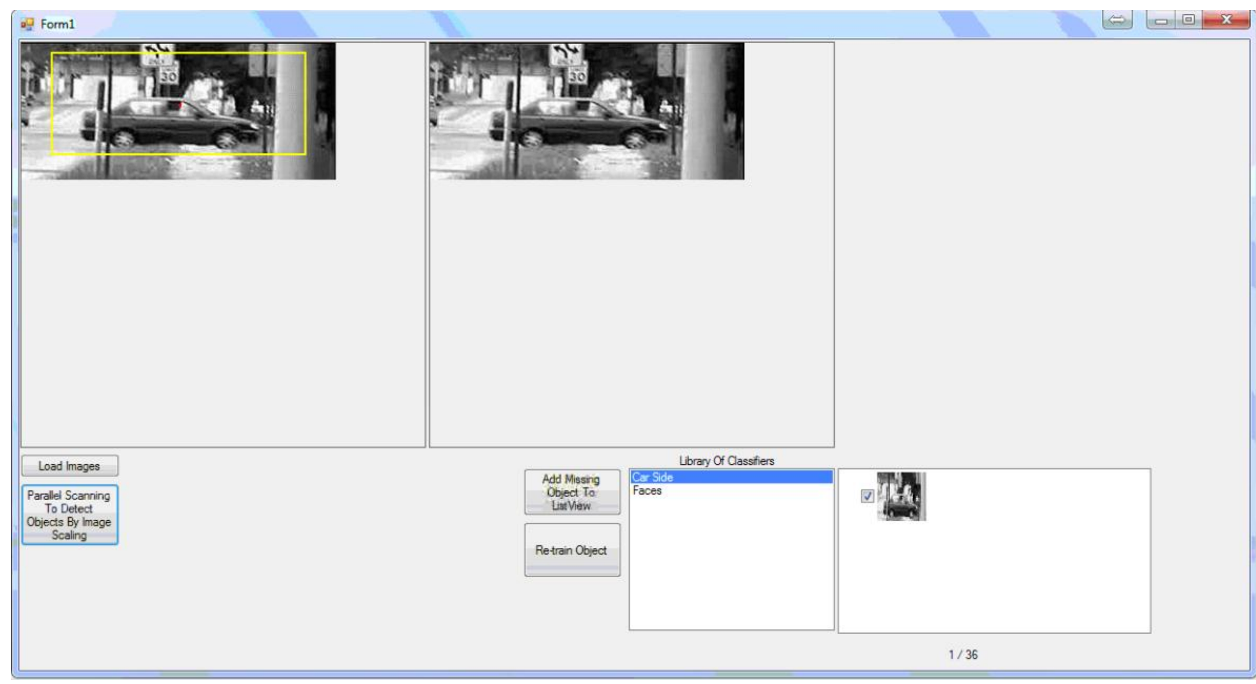
[68] M. Abualkibash, et al., "Highly Scalable, Parallel and Distributed AdaBoost Algorithm using Light Weight Threads and Web Services on a Network of Multi-Core Machines," International Journal of Distributed & Parallel Systems, vol. 4, p. 29, May 2013.

[69] D. Leijen, et al., "The design of a task parallel library," presented at the Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, Orlando, Florida, USA, 2009.

[70] J. Cohen, Statistical power analysis for the behavioral sciences (rev: Lawrence Erlbaum Associates, Inc, 1977.

[71] S. K. Thompson, "Adaptive cluster sampling," Journal of the American Statistical Association, vol. 85, pp. 1050-1059, 1990.

APPENDIX A: SAMPLES OF IMPLEMENTATION CODE




```
public Bitmap FindObject_Cascade_optimized_parallel(Bitmap image,
IntegralImage_Serializable im)
        {
            int W = W_orign;
            int H = H_orign;
            int[,] Faces_array = new int[image.Width, image.Height];
            double[,] Faces_array_values = new double[image.Width, image.Height];
            int blockNumber = 0;
            for (int l = 0; l <= (image.Height - H); l++)
            {

                Parallel.For(0, (image.Width - W + 1), c =>
                {
                    double res = 0;
                    for (int stageNo = 0; stageNo <
bestFeatureList_Objects_stages.Count; stageNo++)
                    {
                        if (stageNo == 0)
```

```
                        {
                            res =
track_Objects_basedOn_Features_Cascade_optimized_parallel(im, l, c, stageNo, W,
H);
                            if (res != 0)
                            {
                                Faces_array[c, l] = 1;
                                Faces_array_values[c, l] = res;

                            }
                            else
                            {
                                Faces_array[c, l] = 0;
                                Faces_array_values[c, l] = 0;
                            }

                        }
                        else
                        {
                            if (Faces_array[c, l] == 1)
                            {

                                res =
track_Objects_basedOn_Features_Cascade_optimized_parallel(im, l, c, stageNo, W,
H);
                                if (res != 0)
                                {

                                    Faces_array_values[c, l] += res;

                                }
                                else
                                {
                                    Faces_array[c, l] = 0;
                                    Faces_array_values[c, l] = 0;
                                }
                            }

                        }
                    }

                });
            }

            int[,] Faces_array_id = new int[image.Width, image.Height];
            for (int i = 1; i < image.Height - 1; i++)
            {
                for (int j = 1; j < image.Width - 1; j++)
                {
                    int tl_id = Faces_array[j - 1, i - 1];
                    int tc_id = Faces_array[j, i - 1];
                    int tr_id = Faces_array[j + 1, i - 1];
                    int cl_id = Faces_array[j - 1, i];
                    int cr_id = Faces_array[j + 1, i];
                    int bl_id = Faces_array[j - 1, i + 1];
                    int bc_id = Faces_array[j, i + 1];
```

```
                    int br_id = Faces_array[j + 1, i + 1];
                    if (Faces_array[j, i] == 1)                     {
                        Faces_array_id[j, i] = Faces_array[j, i] + tl_id + tc_id +
tr_id + cl_id + cr_id + bl_id + bc_id + br_id;
                        if (Faces_array_id[j, i] < 3)
                            Faces_array_id[j, i] = 0;
                    }
                }
            }
            int local_counter1 = 0;
            for (int i = 0; i < Faces_array.GetLength(1); i++)
            {
                for (int j = 0; j < Faces_array.GetLength(0); j++)
                {
                    if (Faces_array[j, i] > 0)
                    {
                        DrawRectangle dr = new DrawRectangle();
                        dr.Width = W *reversscalePercent;
                        dr.Height = H *reversscalePercent;
                        dr.X = (j * reversscalePercent);
                        dr.Y = (i * reversscalePercent);
                        dr.Size = dr.Width * dr.Height;
                        dr.X_center = dr.X + (dr.Width / 2);
                        dr.Y_center = dr.Y + (dr.Height / 2);
                        dr.Exist_flag = 1;
                        dr.ScalePercent = scalePercent;
                        dr.Neighbors = Faces_array_id[j, i];
                        dr.Rec_value = (float)Faces_array_values[j, i];
                        dr.R = new Rectangle((int)dr.X, (int)dr.Y, (int)dr.Width,
(int)dr.Height);
                        Bitmap croped_image = ImageUtility.CropImage(borig, dr.R);
                        dr.Img = croped_image;
                        DrawRec_allScaleNoPruning.Add(dr);
                        DrawRec_allScaleNoPruning_index_flag.Add(1);
                        if (Faces_array_global_2DArray[(int)dr.X_center,
(int)dr.Y_center].Exist_flag > 0)
                        {

                            if (dr.Neighbors >
Faces_array_global_2DArray[(int)dr.X_center, (int)dr.Y_center].Neighbors)
                            {
                                Faces_array_global_2DArray[(int)dr.X_center,
(int)dr.Y_center] = dr;
                            }
                            if (dr.Neighbors ==
Faces_array_global_2DArray[(int)dr.X_center, (int)dr.Y_center].Neighbors)
                            {
                                if (dr.Rec_value >
Faces_array_global_2DArray[(int)dr.X_center, (int)dr.Y_center].Rec_value)
                                {
                                    Faces_array_global_2DArray[(int)dr.X_center,
(int)dr.Y_center] = dr;
                                }
                            }
```

```
                        }
                        else
                            Faces_array_global_2DArray[(int)dr.X_center,
(int)dr.Y_center] = dr;

                    }
                }
            }


            return image;
        }

public double
track_Objects_basedOn_Features_Cascade_optimized_parallel(IntegralImage_Serializab
le im, int l, int c, int stageNo, int Rec_W, int Rec_H)
        {

            int numberOfRounds = bestFeatureList_Objects_stages[stageNo].Count;
            double sum_alpha1 =
bestFeatureList_Objects_stages[stageNo][numberOfRounds - 1].Sum_alpha *
Sum_alpha_Multiplication_value_Objects[stageNo

            double final_value = 0;
            object lock_local = new object();
            for (int i = 0; i < numberOfRounds; i++)
            {
                MyRectangle mr = bestFeatureList_Objects_stages[stageNo][i];

                if (mr.Type == "2RH")
                {
                    final_value = final_value + mr.twoRH_jaggedArray(im, mr.Width,
mr.Height, mr.Y + l, mr.X + c, l, c, mr.Theta, mr.Alpha, mr.Sign, Rec_W, Rec_H);

                }
                if (mr.Type == "2RV")
                {
                    final_value = final_value + mr.twoRV_jaggedArray(im, mr.Width,
mr.Height, mr.Y + l, mr.X + c, l, c, mr.Theta, mr.Alpha, mr.Sign, Rec_W, Rec_H);

                }
                if (mr.Type == "3RH")
                {
                    final_value = final_value + mr.ThreeRH_jaggedArray(im,
mr.Width, mr.Height, mr.Y + l, mr.X + c, l, c, mr.Theta, mr.Alpha, mr.Sign, Rec_W,
Rec_H);

                }
                if (mr.Type == "3RV")
                {
                    final_value = final_value + mr.ThreeRV_jaggedArray(im,
mr.Width, mr.Height, mr.Y + l, mr.X + c, l, c, mr.Theta, mr.Alpha, mr.Sign, Rec_W,
Rec_H);

                }
                if (mr.Type == "4R")
```

```
                {
                    final_value = final_value + mr.FourR_jaggedArray(im, mr.Width,
mr.Height, mr.Y + l, mr.X + c, l, c, mr.Theta, mr.Alpha, mr.Sign, Rec_W, Rec_H);

                }
            }

            if (final_value > sum_alpha1)
                return final_value - sum_alpha1;
            else
                return 0;

        }

public void pruning7_3()
        {

            DrawRectangle[] test1 =
(DrawRectangle[])DrawRec_allScaleNoPruning.ToArray().Clone();
            DrawRectangle[] test2 =
(DrawRectangle[])DrawRec_allScaleNoPruning.ToArray().Clone();
            List<DrawRectangle> results = new List<DrawRectangle>();
            int no = 4;
            List<int> test1_indx = new List<int>();


            for (int i = 0; i < test1.Length; i++)
            {
                DrawRectangle dr_temp = test1[i];
                int indx = i;
                for (int j = 0; j < test2.Length; j++)
                {
                    if (i != j)
                    {
                        {
                            Rectangle r1 = test1[i].R;
                            Rectangle r2 = test2[j].R;

                            if (r1.IntersectsWith(r2))
                            {

                                {
                                    Rectangle r_ij = Rectangle.Intersect(r1, r2);
                                    if ((r_ij.Width * r_ij.Height) >= (r1.Width *
r1.Height) )

                                    {
                                        if (test2[j].Neighbors >=
dr_temp.Neighbors)

                                        {

                                            dr_temp = test2[j];
                                            indx = j;
                                        }


                                    }
```

```
                                }
                            }
                        }
                    }
                }
                test1_indx.Add(indx);
            }
            test1_indx = test1_indx.Distinct().ToList();
            DrawRectangle[] test111 = new DrawRectangle[test1.Length];
            for (int i = 0; i < test1_indx.Count; i++)
            {

                test111[test1_indx[i]] = test1[test1_indx[i]];
            }
            for (int i = 0; i < test111.Length; i++)
                if (test111[i] != null)
                    results.Add(test111[i]);



                test1 = results.ToArray();
                test2 = results.ToArray();
                int id = 1;
                int counter = 0;

                for (int i = 0; i < test1.Length; i++)
                {

                    if( test1[i].Locker==0)
                    for (int j = 0; j < test2.Length; j++)
                    {
                        if (i != j )
                        {
                            int center_rec_widthOver2_i =
(int)Math.Floor(test1[i].Width * .2);
                            int center_rec_heightOver2_i =
(int)Math.Floor(test1[i].Height * .2);
                            int center_rec_newX_center_i = (int)test1[i].X_center
- center_rec_widthOver2_i;
                            int center_rec_newY_center_i = (int)test1[i].Y_center
- center_rec_heightOver2_i;
                            Rectangle r1 = new Rectangle(center_rec_newX_center_i,
center_rec_newY_center_i, center_rec_widthOver2_i * 2, center_rec_heightOver2_i *
2);
                            int center_rec_widthOver2_j =
(int)Math.Floor(test1[j].Width * .2);
                            int center_rec_heightOver2_j =
(int)Math.Floor(test1[j].Height * .2);
                            int center_rec_newX_center_j = (int)test1[j].X_center
- center_rec_widthOver2_j;
                            int center_rec_newY_center_j = (int)test1[j].Y_center
- center_rec_heightOver2_j;
                            Rectangle r2 = new Rectangle(center_rec_newX_center_j,
center_rec_newY_center_j, center_rec_widthOver2_j * 2, center_rec_heightOver2_j *
2);
```

```
                              if (r1.IntersectsWith(r2))
                              {


                                  {
                                      Recursive_2DArray_Overlaping(j, ref
test2[j],ref test1, ref id, ref counter);
                                      ++id;
                                      counter = 0;

                                  }
                              }
                          }

                      }

                  }

                  DrawRectangle[]
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value = new DrawRectangle[id];
                  for (int i = 0; i <
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value.Length; i++)
                  {
                      DrawRectangle dr = new DrawRectangle();
                      dr.Id = 0;
                      dr.Neighbors = -1;
                      Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[i] =
dr;
                  }
                  for (int i = 0; i < test1.Length; i++)
                  {

                      if (test1[i].Neighbors >
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[test1[i].Id].Neighbors)
                      {

Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[test1[i].Id] = test1[i];
                      }
                      else if ((test1[i].Neighbors ==
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[test1[i].Id].Neighbors))
                      {
                          if (test1[i].Rec_value >
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[test1[i].Id].Rec_value)
                          {

Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[test1[i].Id] = test1[i];
                          }
                      }

                  }
                  results = new List<DrawRectangle>();
                  for (int i = 0; i <
Rec_belongTo_max_number_of_Neighbors_then_max_rec_value.Length; i++)
                  {
                      if
(Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[i].Id > 0)
```

```
results.Add(Rec_belongTo_max_number_of_Neighbors_then_max_rec_value[i]);
                }
                results = results.Distinct().ToList();


            DrawRectangle[] list_of_results = results.ToArray();

            Graphics g1 = picBox_Original.CreateGraphics();
            Brush aBrush = (Brush)Brushes.Red;

System.Diagnostics.Debug.WriteLine("*********************************** " +
list_of_results.Length);
            ImageList il = new ImageList();
            il.ImageSize = new Size(48, 48);
            il.ColorDepth = ColorDepth.Depth24Bit;
            lbl_selectedFaces.Text = list_of_results.Length.ToString();
            for (int h = 0; h < list_of_results.Length; h++)
            {
                if (list_of_results[h] != null)
                {

                    if (list_of_results[h].Img != null &&
list_of_results[h].Neighbors >= 4)
                    {
                        il.Images.Add(list_of_results[h].Img);
                        selectedNonFaces_DrawRectangle.Add(list_of_results[h]);

                        g1.FillRectangle(aBrush, list_of_results[h].X_center,
list_of_results[h].Y_center, 3, 3);
                        g1.DrawRectangle(new Pen(Color.Yellow, 2),
list_of_results[h].X, list_of_results[h].Y, list_of_results[h].Width,
list_of_results[h].Height);
                    }
                }

            }

            listView1.View = View.LargeIcon;
            listView1.MultiSelect = true;
            listView1.CheckBoxes = true;
            listView1.GridLines = true;
            listView1.LargeImageList = il;

            for (int i = 0; i < il.Images.Count; i++)
            {
                ListViewItem lvi = new ListViewItem();
                lvi.ImageIndex = i;

                listView1.Items.Add(lvi);

            }
            foreach (ListViewItem item in listView1.Items)
            {
                item.Checked = true;
            }
```

```
        }

public Features[]
Upload_some_Features_ToMemory_Parallel_onSlave_IntegralImage_Serializable(List<Int
egralImage_Serializable> IntImgList, int index, int size)
        {

            int imageSize = Total_of_F_NF;
            int FeaturesSize = size;
            Features[] feature = new Features[FeaturesSize];
            for (int i = 0; i < feature.Length; i++)
                feature[i] = new Features(imageSize);


            try
            {
                int feature_count = 0;
                int count = 0;
                DateTime startTimeFeaturesCalc = DateTime.Now;
                for (int i = 0; i < Rec_Array_variables.Count; i++)
                {
                    if (count >= index && (count < (index + size)))
                    {

                        float[] meanImage_Array = new float[IntImgList.Count];
                        float[] meanImageS_Array = new float[IntImgList.Count];
                        float[] variance_Array = new float[IntImgList.Count];
                        float[] A1_Array = new float[IntImgList.Count];
                        float[] A2_Array = new float[IntImgList.Count];
                        float[] A3_Array = new float[IntImgList.Count];
                        float[] A4_Array = new float[IntImgList.Count];
                        float[] value_Array = new float[IntImgList.Count];
                        int[] value_Array_indx = new int[IntImgList.Count];

                        switch (Rec_Array_variables[i].Type)
                        {
                            case "4R":


                                Parallel.For(0, IntImgList.Count, img =>
                                {

                                    meanImage_Array[img] =
(IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][Rec_Array_variables
[i].Image_W] + IntImgList[img].IntegralImage[0][0] -
(IntImgList[img].IntegralImage[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);
                                    meanImageS_Array[img] =
(IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][Rec_Array_variable
s[i].Image_W] + IntImgList[img].IntegralImageS[0][0] -
(IntImgList[img].IntegralImageS[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);
```

```
                                    variance_Array[img] = meanImageS_Array[img] -
(meanImage_Array[img] * meanImage_Array[img]);
                                    variance_Array[img] =
(float)((variance_Array[img] >= 0) ? Math.Sqrt(variance_Array[img]) : 1);


                                    A1_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 2 - 1];

                                    A2_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1];

                                    A3_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1];

                                    A4_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 2 - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital - 1];
```

```
                                    A1_Array[img] = (float)((A1_Array[img]) /
variance_Array[img]);
                                    A2_Array[img] = (float)((A2_Array[img]) /
variance_Array[img]);
                                    A3_Array[img] = (float)((A3_Array[img]) /
variance_Array[img]);
                                    A4_Array[img] = (float)((A4_Array[img]) /
variance_Array[img]);

                                    value_Array[img] = (A1_Array[img] +
A3_Array[img]) - (A2_Array[img] + A4_Array[img]);

                                    value_Array_indx[img] = img;

                                });

                                break;
                            case "3RH":
                                Parallel.For(0, IntImgList.Count, img =>
                                {


                                    meanImage_Array[img] =
(IntImgList[img].IntegralImage[Image_H][Image_W] +
IntImgList[img].IntegralImage[0][0] - (IntImgList[img].IntegralImage[0][Image_W] +
IntImgList[img].IntegralImage[Image_H][0])) / (Image_H * Image_W);
                                    meanImageS_Array[img] =
(IntImgList[img].IntegralImageS[Image_H][Image_W] +
IntImgList[img].IntegralImageS[0][0] - (IntImgList[img].IntegralImageS[0][Image_W]
+ IntImgList[img].IntegralImageS[Image_H][0])) / (Image_H * Image_W);


                                    variance_Array[img] = meanImageS_Array[img] -
(meanImage_Array[img] * meanImage_Array[img]);
                                    variance_Array[img] =
(float)((variance_Array[img] >= 0) ? Math.Sqrt(variance_Array[img]) : 1);
                                    A1_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 3 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 3 - 1];
                                    A2_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 3 - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small + 2 *
Rec_Array_variables[i].W_capital / 3 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 3 - 1] -
```

```
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + 2 * Rec_Array_variables[i].W_capital / 3 - 1];
                                    A3_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + 2 * Rec_Array_variables[i].W_capital / 3 - 1]
+ IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small + 2 *
Rec_Array_variables[i].W_capital / 3 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital - 1];


                                    A1_Array[img] = (float)((A1_Array[img]) /
variance_Array[img]);
                                    A2_Array[img] = (float)((A2_Array[img]) /
variance_Array[img]);
                                    A3_Array[img] = (float)((A3_Array[img]) /
variance_Array[img]);

                                    value_Array[img] = (A2_Array[img] -
A1_Array[img] - A3_Array[img]) + (meanImage_Array[img] *
((Rec_Array_variables[i].W_capital * Rec_Array_variables[i].H_capital) / 3) /
variance_Array[img]);

                                    value_Array_indx[img] = img;

                                });


                                break;
                            case "3RV":
                                Parallel.For(0, IntImgList.Count, img =>
                                {


                                    meanImage_Array[img] =
(IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][Rec_Array_variables
[i].Image_W] + IntImgList[img].IntegralImage[0][0] -
(IntImgList[img].IntegralImage[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);
                                    meanImageS_Array[img] =
(IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][Rec_Array_variable
s[i].Image_W] + IntImgList[img].IntegralImageS[0][0] -
(IntImgList[img].IntegralImageS[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);

variance_Array[img] = meanImageS_Array[img] - (meanImage_Array[img] *
meanImage_Array[img]);
                                    variance_Array[img] =
(float)((variance_Array[img] >= 0) ? Math.Sqrt(variance_Array[img]) : 1);
```

```
                                    A1_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small - 1];
                                    A2_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small + 2 *
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small + 2 *
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small - 1];
                                    A3_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small + 2 *
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small + 2 *
Rec_Array_variables[i].H_capital / 3 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small - 1];

                                    A1_Array[img] = (float)((A1_Array[img]) /
variance_Array[img]);
                                    A2_Array[img] = (float)((A2_Array[img]) /
variance_Array[img]);
                                    A3_Array[img] = (float)((A3_Array[img]) /
variance_Array[img]);

                                    value_Array[img] = (A2_Array[img] -
A1_Array[img] - A3_Array[img]) + (meanImage_Array[img] *
((Rec_Array_variables[i].W_capital * Rec_Array_variables[i].H_capital) / 3) /
variance_Array[img]);

                                    value_Array_indx[img] = img;


                                });

                                break;

                            case "2RH":


                                Parallel.For(0, IntImgList.Count, img =>
```

```
                                {
                                    meanImage_Array[img] =
(IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][Rec_Array_variables
[i].Image_W] + IntImgList[img].IntegralImage[0][0] -
(IntImgList[img].IntegralImage[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);
                                    meanImageS_Array[img] =
(IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][Rec_Array_variable
s[i].Image_W] + IntImgList[img].IntegralImageS[0][0] -
(IntImgList[img].IntegralImageS[0][Rec_Array_variables[i].Image_W] +
IntImgList[img].IntegralImageS[Rec_Array_variables[i].Image_H][0])) /
(Rec_Array_variables[i].Image_H * Rec_Array_variables[i].Image_W);

                                    variance_Array[img] = meanImageS_Array[img] -
(meanImage_Array[img] * meanImage_Array[img]);
                                    variance_Array[img] =
(float)((variance_Array[img] >= 0) ? Math.Sqrt(variance_Array[img]) : 1);

                                    A1_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small - 1];
                                    A2_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital / 2 - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small - 1];


                                    A1_Array[img] = (float)((A1_Array[img]) /
variance_Array[img]);
                                    A2_Array[img] = (float)((A2_Array[img]) /
variance_Array[img]);


                                    value_Array[img] = (A2_Array[img] -
A1_Array[img]);

                                    value_Array_indx[img] = img;

                                });
```

```
                                break;
                            case "2RV":

                                Parallel.For(0, IntImgList.Count, img =>
                                {
                                    meanImage_Array[img] =
(IntImgList[img].IntegralImage[Image_H][Image_W] +
IntImgList[img].IntegralImage[0][0] - (IntImgList[img].IntegralImage[0][Image_W] +
IntImgList[img].IntegralImage[Image_H][0])) / (Image_H * Image_W);
                                    meanImageS_Array[img] =
(IntImgList[img].IntegralImageS[Image_H][Image_W] +
IntImgList[img].IntegralImageS[0][0] - (IntImgList[img].IntegralImageS[0][Image_W]
+ IntImgList[img].IntegralImageS[Image_H][0])) / (Image_H * Image_W);

variance_Array[img] = meanImageS_Array[img] - (meanImage_Array[img] *
meanImage_Array[img]);
                                    variance_Array[img] =
(float)((variance_Array[img] >= 0) ? Math.Sqrt(variance_Array[img]) : 1);

                                    A1_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 2 - 1];
                                    A2_Array[img] =
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital / 2 - 1] +
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital - 1] -
IntImgList[img].IntegralImage[Rec_Array_variables[i].L_small +
Rec_Array_variables[i].H_capital - 1][Rec_Array_variables[i].C_small +
Rec_Array_variables[i].W_capital / 2 - 1] -
IntImgList[img].IntegralImageS[Rec_Array_variables[i].L_small -
1][Rec_Array_variables[i].C_small + Rec_Array_variables[i].W_capital - 1];


                                    A1_Array[img] = (float)((A1_Array[img]) /
variance_Array[img]);
                                    A2_Array[img] = (float)((A2_Array[img]) /
variance_Array[img]);


                                    value_Array[img] = (A2_Array[img] -
A1_Array[img]);

                                    value_Array_indx[img] = img;

                                });
```

```
                                break;
                            }
                            Array.Sort(value_Array, value_Array_indx);

                            Parallel.ForEach(value_Array_indx, img =>
                            {

                                feature[feature_count].F[img] = value_Array[img];
                                feature[feature_count].Indices[img] =
value_Array_indx[img];
                            });
                            feature_count++;
                        }
                        count++;

                    }
                }
                catch (Exception ex)
                {

                }
                return feature;
            }

public void Read_Featurs_FromMemory_Parallel_Multi_Slave_sorted_data(int position)
        {
            try
            {
                Mini_epsilon = float.PositiveInfinity;
                int[] Y = new int[Total_of_F_NF];
                for (int i = 0; i < Y.Length; i++)
                {
                    if (i < no_of_Faces)
                        Y[i] = 1;
                    else
                        Y[i] = -1;
                }

                Parallel.For(0, (Rec_type.Length), i =>
                {
                                        DecisionStump_sorted_data(Y, Rec_type[i],
(i + position), Rec_Array_variables[i + position]);
                });
            }
            catch (Exception e)
            {

            }
        }

private void DecisionStump_sorted_data(int[] Y1, Features Rec, int s,
Rectangle_variables rec_var)
        {

            try
```

```
            {
                float[] data = Rec.F;
                int[] indices = Rec.Indices;
                int w = rec_var.W_capital;
                int h = rec_var.H_capital;
                int y = rec_var.L_small;
                int x = rec_var.C_small;
                string type = rec_var.Type;

                int index2 = 0;
                float minimumError = float.PositiveInfinity;

                float _threshold = float.NaN;
                int _sign = 0;


                float totalError = 0;
                for (int i = 0; i < Y1.Length; i++)
                    totalError += Weights[i];

                float positive_weights_sum = 0;
                float negative_weights_sum = 0;
                for (int i = 0; i < Y1.Length; i++)
                    negative_weights_sum += Y1[i] != 1 ? Weights[i] : 0;
                positive_weights_sum = totalError - negative_weights_sum;
                float positive_weights_sum_below = positive_weights_sum;
                float negative_weights_sum_below = negative_weights_sum;
                float currentError;
                float e_p;
                float e_n;

                for (int i = 0; i < Y1.Length; i++)
                {
                    int index = indices[i];

                    if (Y1[index] == 1)
                        positive_weights_sum_below -= Weights[index];
                    else
                        negative_weights_sum_below -= Weights[index];




                    e_p = positive_weights_sum_below + (negative_weights_sum -
negative_weights_sum_below);
                    e_n = negative_weights_sum_below + (positive_weights_sum -
positive_weights_sum_below);


                    if (e_p < e_n)                    {
                        currentError = e_p;
                        if (minimumError > currentError)
                        {
                            minimumError = currentError;
                            _threshold = data[i];
```

```
                                _sign = -1;

                                index2 = i;                            }

                            }
                            else                         {
                                currentError = e_n;
                                if (minimumError > currentError)
                                {
                                    minimumError = currentError;
                                    _threshold = data[i];
                                    _sign = +1;
                                    index2 = i;
                                }

                            }

                        }

                        int[] hClassification = new int[data.Length];
                        float epsilon = 0;
                        for (int ii = 0; ii < data.Length; ii++)
                        {
                            hClassification[indices[ii]] = data[ii] > _threshold ? _sign :
-_sign;
                            epsilon += hClassification[indices[ii]] != Y1[indices[ii]] ?
Weights[indices[ii]] : 0;
                        }

                        lock (olock)
                        {
                            if (epsilon < Mini_epsilon)
                            {

                                Mini_epsilon = epsilon;
                                Mini_position = s;
                                Mini_alpha = (float)Math.Log((1 - epsilon) / epsilon);
                                Mini_Beta = (float)epsilon / (1 - epsilon);
                                Theta = _threshold;
                                Min_sign = _sign;
                                Mini_hClassification = (int[])hClassification.Clone();
                                W = w;
                                H = h;
                                X = x;
                                Y = y;
                                Type = type;

                            }
                        }
                    }
                    catch (Exception e)
                    {

                    }

                }
```