BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

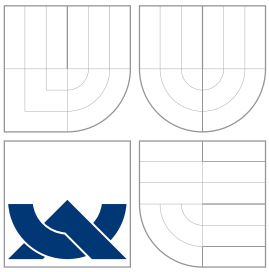
ACCELERATION OF OBJECT DETECTION USING CLASSIFIERS

DOCTORAL THESIS
DISERTAČNÍ PRÁCE

AUTHOR
AUTOR PRÁCE

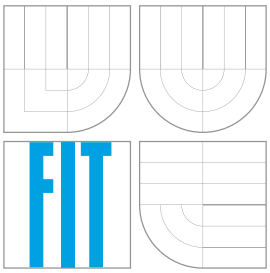
Ing. ROMAN JURÁNEK

BRNO 2012



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ACCELERATION OF OBJECT DETECTION USING CLASSIFIERS

AKCELEROVANÁ DETEKCE OBJEKTŮ POMOCÍ KLASIFIKÁTORŮ

DOCTORAL THESIS

DISERTAČNÍ PRÁCE

AUTHOR — Ing. ROMAN JURÁNEK

AUTOR PRÁCE

SUPERVISOR — Doc. Dr. Ing. PAVEL ZEMČÍK

VEDOUCÍ PRÁCE

BRNO 2012

Abstrakt

Detekce objektů v počítačovém vidění je složítá úloha. Velmi populární a rozšířená metoda pro detekci je využití statistických klasifikátorů a skenovacích oken. Pro učení kalsifikátorů se často používá algoritmus AdaBoost (nebo jeho modifikace), protože dosahuje vysoké úspěšnosti detekce, nízkého počtu chybných detekcí a je vhodný pro detekci v reálném čase. Implementaci detekce objektů je možné provést různými způsoby a lze využít vlastnosti konkrétní architektury, pro urychlení detekce. Pro akceleraci je možné využít grafické procesory, vícejádrové architektury, SIMD instrukce, nebo programovatelný hardware. Tato práce představuje metodu optimalizace, která vylepšuje výkon detekce objektů s ohledem na cenovou funkci zadanou uživatelem. Metoda rozděluje předem natrénovaný klasifikátor do několika různých implementací, tak aby celková cena klasifikace byla minimalizována. Metoda je verifikována na základním experimentu, kdy je klasifikátor rozdělen do předzpracovací jednotku v FPGA a do jednotky ve standardním PC.

Abstract

Detection of objects in computer vision is a complex task. One of most popular and well explored approaches is the use of statistical classifiers and scanning windows. In this approach, classifiers learned by AdaBoost algorithm (or some modification) are often used as they achieve low error rates, high detection rates and they are suitable for detection in real-time applications. Object detection run-time which uses such classifiers can be implemented by various methods and properties of underlying architecture can be used for speed-up of the detection. For the purpose of acceleration, graphics hardware, multi-core architectures, SIMD or other means can be used. The detection is often implemented on programmable hardware. The contribution of this thesis is to introduce an optimization technique which enhances object detection performance with respect to an user defined cost function. The optimization balances computations of previously learned classifiers between two or more run-time implementations in order to minimize the cost function. The optimization method is verified on a basic example – division of a classifier to a pre-processing unit implemented in FPGA, and a post-processing unit in standard PC.

Klíčová slova

Detekce objektů, AdaBoost, WaldBoost, Akcelerace, SIMD, Minimalizace ceny

Keywords

Object Detection, AdaBoost, WaldBoost, Acceleration, SIMD, Cost Minimization

Bibliographic citation

Roman Juránek: *Acceleration of Object Detection using Classifiers*, doctoral thesis, Brno, Brno University of Technology, Faculty of Information Technology, 2012.

Acceleration of Object Detection using Classifiers

Declaration

I declare that this dissertation thesis is my original work and that I have written it under lead of Doc. Dr. Ing. Pavel Zemčík. All sources and literature that I have used during elaboration of the thesis are correctly cited with complete reference to the corresponding sources.

.....

Roman Juránek

March 22, 2012

Acknowledgment

I would like to thank to many people. First of all to Pavel Zemčík, my supervisor, for his guidance, knowledge and (sometimes) crazy ideas. To my friends for their personal and professional support, especially to Michal Hradiš, Honza Navrátil, Aleš Láník and Jozef Mlích. I would like to also thank my family and my girlfriend, Marketa, for her support and love.

This work has been supported by the FIT VUT Brno project "Advanced recognition and presentation of multimedia data", FIT VUT, FIT-S-11-2, Centre of excellence in computer science "The IT4Innovations Centre of Excellence", EU, CZ 1.05/1.1.00/02.0070, "Smart Multicore Embedded Systems", Artemis JU, SMECY No. 100230, and and "Centre of Computer Graphics", MŠMT, LC06008.

Contents

1 Introduction **1**

2 Object Detection using Classifiers **4**
2.1 Classification Overview 5
2.2 Adaptive Boosting 7
2.3 Advanced Classifier Structures 10

3 Feature Extraction **15**
3.1 Haar-like Features 16
3.2 Local Binary Patterns 17
3.3 Local Rank Functions 18
3.4 Histograms of Oriented Gradients 20
3.5 Other Feature Types 21

4 Architectures and Acceleration **24**
4.1 Levels of Acceleration 25
4.2 Graphics Hardware 30
4.3 Programmable Hardware 31
4.4 Algorithmic Accelerations 33

5 Exploitation of SIMD for Feature Response Computation **35**
5.1 Representation of a Classifier 35
5.2 Input Pre-processing and Data Access 38
5.3 Feature Evaluation using SIMD 44
5.4 Implementation with Intel SSE 48

6 Classification Cost and its Minimization **53**
6.1 Classifier Properties 53
6.2 Cost Evaluation 54
6.3 Cost Minimization 55

7 Experiments and Results **58**
7.1 Classification Cost Measurements 58
7.2 Cost Minimization 62
7.3 Results Discussion 73

8 Conclusion **76**

List of Figures

2.1 Object detection examples 4
2.2 Scheme of machine learning 5
2.3 Classification of an image 6
2.4 Detection with sliding window 7
2.5 Decision stump learning 10
2.6 Attentional cascade of classifiers 11
2.7 Results of Viola and Jones cascade 12
2.8 Results of WaldBoost learning 13

3.1 Haar features 16
3.2 Integral image 16
3.3 Example of Haar feature response calculation 17
3.4 Calculation of the LBP feature 18
3.5 MB-LBP feature 18
3.6 Calculation of the LRF feature 19
3.7 Calculation of the HoG feature 20
3.8 Sparse Granular Features 21
3.9 3D Haar features 22
3.10 Patterns of motion and appearance 22
3.11 Gabor wavelets 23

4.1 Speed-up on multi-core architectures 27
4.2 Parallel processing of frames 27
4.3 Parallel processing of sub-windows 28
4.4 Parallel processing of weak hypotheses 28
4.5 SIMD instruction 29
4.6 Block structure of a circuit for object detection 32

4.7 Neighborhood suppression 33

5.1 Parametrization of features 36
5.2 Scheme of a strong classifier 38
5.3 Feature evaluation on an intensity image 39
5.4 Feature evaluation on an integral image 40
5.5 Feature evaluation on a pre-convolved image 40
5.6 Local rearrangement of a convolution image 41
5.7 Example of a feature in local image rearrangement 42
5.8 Feature evaluation using a pre-calculated response 43
5.9 Calculating a feature response by SIMD 44
5.10 Evaluation of multiple features by SIMD 46
5.11 Pre-calculation of feature responses 47
5.12 Pre-processing scheme in the library 49
5.13 Image layout 49
5.14 Comparison of classifiers 51
5.15 Comparison of the library with OpenCV 52

6.1 Example of classifier statistics 54
6.2 Composition of two different implementations of classification 56
6.3 Example of cost minimization 56

7.1 Statistics of classifiers 61
7.2 Optimization results for PC1 (LBP) 63
7.3 Optimization results for PC1 (LRD) 64
7.4 Optimization results for PC1 (LRD) 65
7.5 Optimization results for PC2 (LBP) 66
7.6 Optimization results for PC2 (LRD) 67
7.7 Optimization results for PC2 (LRP) 68
7.8 Comparison of optimization with measurements on PC1 69
7.9 Comparison of optimization with measurements on PC1 70
7.10 Comparison of optimization with measurements on PC2 71
7.11 Comparison of optimization with measurements on PC2 72

List of Tables

5.1 Ranges feature responses 36
5.2 Properties of run-times 50

7.1 Specifications of machines used in experiments 59
7.2 Feature evaluation costs 60

CHAPTER 1

Introduction

Many real life applications could use information about objects captured by a camera. In user interfaces, for example, a camera can be used as an alternative input device. The computer can capture the scene in front of the computer and analyse it in order to find the user's face and analyse the user's gestures. Such input can be used to control the computer without a keyboard or mouse. Other applications, like traffic control or surveillance systems, use detection of objects to automatically count people, check licence plates or record traffic violations. Object detection is an important part of such systems.

Visual object detection is one of the most challenging tasks in computer vision. The goal of object detection is to localize the target object in an input image using visual features. During past decades, researchers tried to solve this problem and developed a vast number of methods for machine learning, information extraction and object representation. Object detectors are in most cases based on detection from 2D imagery produced by a camera (operating in a visible light spectrum or in near infra red or even far infra red, depending on its application). Lately, the advances in the capturing of 3D data along with 2D imagery and intensive research allowed for exploitation of depth information in object detectors which results in more advanced and precise detectors. There exists consumer devices that can produce fast and reliable depth information – e.g. Microsoft Kinect (based on infra camera and stereo matching) and also a range of ToF (Time of Flight) cameras that can bring depth information directly from a single image sensor (based on light travel time). 2D intensity image is, however, still the main source for recognition and detection. This thesis focuses on the acceleration of object detection from 2D images using statistical classifiers. However, the principles described in the work can be used in more general

cases.

A very popular approach to the object detection is exploitation of statistical classifiers and scanning windows technique. The classifier is learned by a machine learning algorithm, typically supervised or semi-supervised. Among the large number of statistical machine learning methods, the most prominent in real-time object detection is the Adaptive Boosting algorithm and its modifications. This method became so popular (due to its simplicity and performance) that it found its way to commercial systems that use object detection. For example, some driving assistants in modern vehicles use detection of pedestrians, traffic signs and other objects to inform driver about situation in front of the vehicle. Most consumer digital still cameras can use face detection as side information for focusing and exposition measurement. The cameras even use the detection of a smile to release the shutter. Biometric systems (such as user recognition) use detection and analysis of faces. Such systems can be used for user access systems (user login to the computer, for example). Human-computer interfaces can use face detection and gesture tracking as an input for a computer that can be used by physically challenged persons. And there exists more applications – traffic surveillance, security, navigation, etc.

The approach with scanning windows has been known for a long time. In this principle, selection of sub-windows of the input image is analyzed by a classifier which makes decision about the presence or absence of an object. The first successful algorithms were developed in the late 1980's and early 1990's but they became popular after Viola and Jones in 2001 introduced their framework for rapid object detection. In their approach, they exploited the Adaptive Boosting algorithm in combination with inaccurate classifiers based on simple image features (Haar wavelets). It was demonstrated that this framework can produce very precise classifiers with low computational complexity suitable even for real-time applications. Since then, their approach was improved by using other learning schemes, classifier structures and image features.

A suitable property of AdaBoost-based detectors is that they are easily implemented in both software and hardware. But although the detection with this method is very fast, it is often not fast enough for the target applications that are gradually more and more demanding. Therefore, either more powerful hardware must be used, or some optimizations must take place. All examples given above would benefit from acceleration or optimization of the detection process. It would lead to a faster response time or lower power consumption. Borrowing from David Marr's three level model, optimizations can be done on *computational*, *algorithmic* and *implementational* levels. Exploitation of computational and algorithmic optimization can reduce complexity of computations by employing more effective algorithms and data representations. For example, important computational optimization is in the building of an attentional cascade of classifiers which directs computational power to

image areas where the occurrence of target objects is more probable. Anther example can be in the integral representation of an image which allows for the summing of an arbitrary sized rectangular area of an image in constant time instead of linear time (when using pixel values only). It is used for fast calculation of Haar feature responses. On the lower level, algorithms can be optimized at an implementation level by exploiting underlying hardware architecture – better use of an instruction set, use of parallelism, custom hardware, etc. A good example of this type of optimization is the porting of algorithms to graphic hardware (e.g. nVidia's Compute Unified Device Architecture) where many simple computational elements are available, or implementation of the detection in FPGA (Field Programmable Gate Array) chips.

The focus in this thesis is on *implementational* acceleration of the detection process. The contribution is the introduction of a technique for the composition of different implementations of object detection in order to enhance its performance with respect to a user defined cost function. The composition balances computations between two or more implementations in order to minimize the cost function. The method is based on an analysis of a previously learned classifier and knowledge of properties of the implementations which executes detection. The method is verified in the example where a classifier is divided into two evaluation phases. Tirst executed in a hardware unit and the second executed in software using implementation with different properties. The classifiers in the thesis are learned by the WaldBoost algorithm and they are frontal face detectors. Image features used are LBP (Local Binary Patterns) and LRF (Local Rank Functions). This combination of learning algorithm and image features can be considered as a state of the art in real-time object detection with scanning-window classifiers.

The thesis is structured as follows. The next section gives an introduction to object detection with classifiers and gives a detailed description of AdaBoost machine learning algorithm and describes advanced learning methods based on this algorithm. Feature extraction methods used in state of the art systems are described in Section 3. Section 4 gives a deeper insight into optimization methods that can be used to accelerate detection of objects. Section 5 describes in detail experimental implementations of the detection run-time exploiting data parallelism in different ways. The main contribution of this thesis – the optimization method – is described in Section 6 and experiments with the method. Experiments, result and application potential are discussed in Section 7. The thesis is concluded in Section 8 with some remarks for future research.

CHAPTER 2

Object Detection using Classifiers

Detection and localization of objects is a complex process where images or image sequences are analyzed in order to search for occurrences of a particular class of objects. The definition of object varies and it is largely application dependent. It is often defined by a set of annotated example images from which a machine learning algorithm automatically derives an internal object class model. Such object classes can be, for example, pedestrians, cars, faces, animals, etc. The output of a detector is information about object position and its size in the input image. Fig. 2.1 shows examples of the detection of facial features and cars.

Figure 2.1: Examples of detection of objects. Left, detection of face and facial features; right, detection of cars (Sources: BioID database, UIUC car database).

Many object detection methods with different properties exist. It is important to select the one which is suitable for a particular application in order to satisfy needed precision and speed of detection. The simplest one is perhaps template matching [10] which can detect a pattern corresponding to a template in the input image. There are more advanced methods which employ template matching in different ways. For example, in [35] they build templates from histograms of gradient orientations for

multiple views of an object. There are even methods that use a sequence of template matching and pooling steps resulting in a feature vector classified by a Support Vector Machine (SVM [87]) classifier [79, 78, 80]. Other methods are based on detection of object parts and searching for their consistent configuration corresponding to a target object [56, 20, 52, 19].

Perhaps the most widely used detection technique at present is based on sliding windows and classifiers [62, 88]. In this class of methods, each area of the input image is subject to a classification which decides if the target object is present or not in the area. Papageorgiou et al. [62] used a set of Haar features classified by an SVM classifier to get reliable detection. Viola and Jones used a similar idea but they used another learning algorithm — Adaptive Boosting which selects and orders elementary (weak) classifiers by their importance. Moreover, they proposed an attentional cascade which allowed for the detection to be executed in real-time. They reported 15 frames per second on a 384×288 pixel image which was the fastest object detection system at the time, about 15 times faster than Rowley et al. [70] and about 600 times faster than Schneiderman-Kanade detector [75].

2.1 Classification Overview

Formally, a classifier is a function $f : \chi \to \mathcal{N}$ that for input data $x \in \chi$ decides its category $y \in \mathcal{N}$ to which the data belongs. In the case of object detection with classifiers, the input is a sample image (represented by a set of features) and the decision is the *background* or *object* (i.e. $y \in \{-1, 1\}$).

The classifier is typically learned by a machine learning algorithm where input is a set of n training samples $\{(x_1, y_1) \dots (x_n, y_n)\}$. The output is the learned classification function f. Fig. 2.2 summarizes the learning process. Features are first extracted from training images — every image is then represented in the learning algorithm by its features, x, and a class label, y. Machine learning then generates a classification function which can distinguish between object and background images.

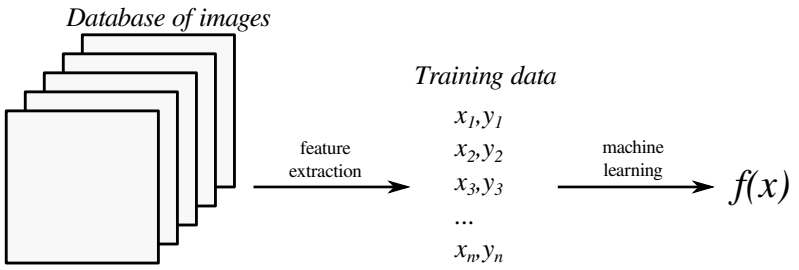


Figure 2.2: Scheme of machine learning. Features extracted from training images along with image labels form the training set used for learning the decision function.

The classification function can be learned by various methods. A learning method needs to deal with a large number of training samples with a high dimensionality of data representation and with noise in the data. Learning algorithms are often based on statistical properties of the training data in a feature space. The Nearest Neighbor (k-NN) classifier [18] assigns the sample to the class which makes a majority on k nearest training samples. When a large training set is used, the search for the nearest samples may be slow (in general, but it is implementation dependent) and thus it is not typically used for rapid object detection. Logistic regression [37] fits logit function to the feature space in order to maximize the probability of a correct classification. The Support Vector Machines (SVM) [87] search for the best separating hyperplane in the feature space. SVMs were successfully used for object detection by Papageorgiou et al. [62] in combination with Haar features and by Mutch and Lowe [59] in combination with Gabor filters. Artificial Neural Networks (ANN) [18] uses the sample as an input for a network of artificial neurons. A successful example of the application of ANN to object detection is face detection [70]. The Boosting algorithms [22, 23] constructs a classifier from simple inaccurate classifiers which taken together makes a very accurate one. Boosting has very interesting properties from the object detection point of view, and it is very suitable for rapid detection [88, 44, 83]. It is used in this thesis as the algorithm for classifier learning. The list above is certainly not complete: it points out some methods that can be used for learning the classifiers. For a review of other methods for classification learning, the reader may refer to [18, 5].

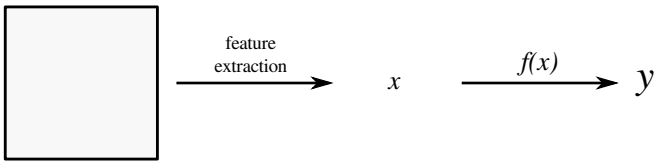


Figure 2.3: Classification of an image. Features extracted from an image are supplied to the previously learned function to estimate the label of the image.

The classifier, shown in Fig. 2.3, can decide if the input sample is or is not an object of interest. Object detection with classifiers is achieved by an analysis of each sub-window of input image by the classifier. The sub-windows are taken from all positions and scales (and possibly other transformation depending on the particular application). This method is called the sliding window detection. The number of analyzed samples is very high, reaching hundreds of thousands per image and thus the resulting classifier should have a very low *false alarm rate*. On the other hand, each object appears in multiple neighboring sub-windows and thus a *false negative rate* does not necessarily need to be zero and some detections can be, in fact, missed. It is only required that at least one of these positive sub-windows are hit by the

classifier.

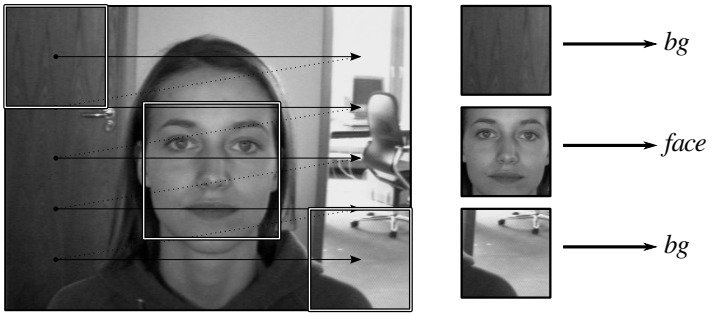


Figure 2.4: Detection with sliding window (image source: BioID face database).

The detection with sliding windows is exploited in many works. In [62] they use an overcomplete set of Haar-like features and an SVM classifier as the basis for a general object detection framework. A similar approach was used in [88] where they use AdaBoost. In [16] authors use histograms of oriented gradients as an input for an SVM classifier in order to detect pedestrians. In [84] the authors propose an emulator of key point detectors using a sliding window detector based on the WaldBoost algorithm.

In this chapter, the Adaptive Boosting learning algorithm and modifications of this method are described as grounds for this thesis.

2.2 Adaptive Boosting

The Adaptive Boosting [22] and other boosting methods [26, 24, 66, 71] is a method for combining weak classifiers $h_t : \chi \to \mathbb{R}$ into one strong classifier H_t. The combination is a weighted average where responses of the weak classifiers are multiplied by weights α determining their importance. A weak classifier is a function that decides object class with a smaller error rate than the random decision. The strong classifier decides the object class more reliably than the individual weak classifiers.

Historically, AdaBoost has its roots in PAC (Probably Approximately Correct) learning framework developed by Valiant [86, 28]. Boosting was first described by Schapire [72] where he introduced a polynomial algorithm for learning. A more efficient version was introduced by Freund [22] and Real AdaBoost was described by Freund and Schapire [23]. The AdaBoost, as a general machine learning algorithm, can be applied to a number of problems including object detection, recognition [45] and tracking [3].

2.2.1 The Algorithm

In the following section, the algorithm used by Viola and Jones [88] is described. They used a discrete variant of AdaBoost [22] where each weak classifier makes a decision about the object class and the strong classifier can thus be viewed as a voting procedure. They used simple decision stump weak classifiers (thresholds) and Haar features that are very simple to evaluate.

Algorithm 1 Discrete AdaBoost as used by Viola and Jones

Input: set of samples $(x_1, y_1) \dots (x_n, y_n)$ where $x \in \mathcal{R}^n$ and $y \in \{-1, 1\}$ for negative and positive samples, respectively.

Output: strong classifier H

1: Initialize weights $w_{1,i} = \frac{1}{2m}$ for samples with $y = 1$ and $w_{1,i} = \frac{1}{2l}$ for samples with $y = -1$, where m and l are the number of positive and negative samples, respectively.

2: **for** $t = 1 \dots T$ **do**

3: Normalize the weights,

$$w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$

so that w_t is probability distribution

4: For each feature, j, train a classifier h_j. The error is evaluated with respect to w_t,

$$\epsilon_j = \sum_j w_{t,i} |h_j(x_i) - y_i|$$

5: Choose classifier, h_t, with lowest error ϵ_t.

6: Update weights:

$$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$

where $e_i = 0$ if sample x_i is classified correctly, $e_i = 1$ otherwise, and $\beta = \frac{\epsilon_t}{1-\epsilon_t}$

7: **end for**

8: The final strong classifier is:

$$H(x) = sgn\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

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

In the principle, AdaBoost greedily selects weak classifiers from a large set. Algorithm 1 shows the learning process. The input is a set of annotated training samples $(x, y), x \in \chi, y \in \{-1, 1\}$, and a set of weak classifiers $h : \chi \to \{-1, 1\}$. For each training sample i, the algorithm keeps its weight w_i which represents its importance for learning. Samples with high weights have more influence on the selection of weak classifiers. The weight in some sense expresses how hard is to classify the given sample. Using the sample weight, the algorithm focuses on the *hard* samples. In the beginning, the weights are initialized evenly for all positive and

negative samples.

The algorithm is iterative. In each iteration, one weak classifier is selected to the strong classifier from a large pool. In more detail, the algorithm first adjusts the parameters of all the weak classifiers in order to behave best on the current set of samples taking their weights into account. The weak classifier with the lowest weighted classification error is added into the strong classifier. At the end of iteration, the samples are re-weighted using the error of the selected weak classifier. Through this mechanism, the algorithm gradually adapts to samples that are hard to classify, decreasing the classification error of the strong classifier.

It has been proven [23] that the training error decreases with each iteration of the algorithm. Moreover, it has been proven that the error is upper-bounded by an exponential function. This gives a theoretical warranty that the training error drops exponentially when weak classifiers (at least slightly) better than the random function are used. The algorithm execution time is proportional to: the number of training samples, the number of weak classifiers in the training and the number of weak classifiers selected to the strong classifier.

2.2.2 Weak Classifiers

Viola and Jones used simple decision stump weak classifiers based on a single Haar feature. Equation (2.1) shows the calculation of the weak classifier response, where $f(x)$ is the response of the feature, p is the polarity of the response and θ is the decision threshold.

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ -1 & \text{otherwise} \end{cases} \tag{2.1}$$

During learning, parameters p_j and θ_j for each weak classifier h_j (i.e. using different features as an input) are estimated in order to minimize the error function ϵ_j from Algorithm 1. Fig. 2.5 shows distributions of feature responses for positive and negative classes and the plot of ϵ_j for different settings of θ_j ($p_j = 1$ is assumed). The optimal setting of θ_j is in the minima of ϵ_j.

An improved AdaBoost algorithm, Real AdaBoost proposed by Schapire and Singer [73], allows for using real-valued weak classifiers which does not vote for an object class, but they rather express their confidence about the presence of the target object class. The weak classifiers in Real AdaBoost are in the form $h : \chi \to \mathcal{R}$ incorporating the α from Alhorithm 1 directly into the weak classifier. An example of such weak classifier is *space partitioning weak classifier* whose input is the discrete feature response $f : \chi \to \mathbb{N}$. The weak classifier contains a partitioning function $l : \mathbb{N} \to \mathbb{R}$ which assigns a response to a feature response. Such weak classifiers can be easily implemented by a look-up table. The learning of such weak classifiers

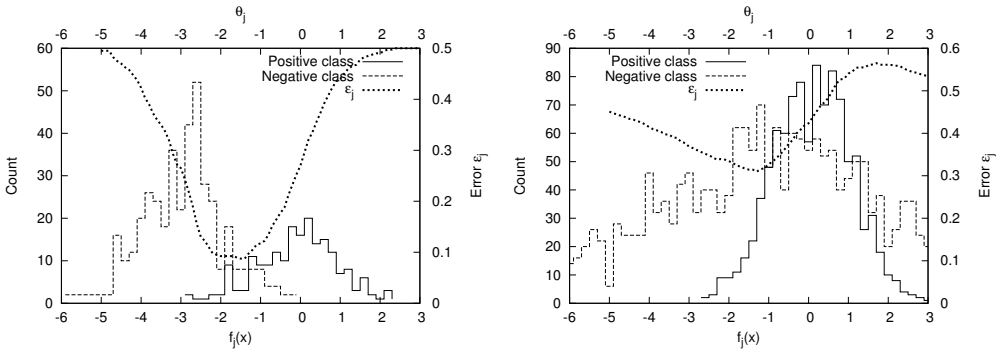


Figure 2.5: Examples of decision stump learning for two different features f_j. Each plot shows histograms of feature responses for positive and negative classes and the error function ϵ_j. (a) shows the case when the two classes are separated well by the feature; and (b) when the classes are not separated well.

consist of assigning the values to the look-up table function l_j in order to minimize the error function ϵ_j (which is defined slightly differently in Real AdaBoost [73]).

2.2.3 Algorithm Modifications

Discrete and Real AdaBoost were gradualy improved to address deferent aspects of learning. Asymmetric AdaBoost [89] take into account the fact that the classes in deteection tasks are highly skewed (there are much more background samples than face samples for example) and modifies the loss function of the Real AdaBoost such that the cost of a false negative is higher than the cost of a false positive classification. As a result, the classifiers have higher detection rates and lower false positive rates.

In [53] the authors propose FloatBoost algorithm which extends the Real AdaBoost with *conditional exclusion* which after a selection of every weak classifiers performs a *backtrack* [65]. The backtrack removes from the strong classifier all weak classifiers that can cause higher error rates. Classifiers learned by FloatBoost have lower error rates. It takes, however, longer time to train classifiers due to the backtrack phase.

And there are more modifications, e.g. AdaBoost.MH [24], Kullback-Lieber Boosting [55], AdaBoost with totally corrective updates [81], Linear Programming Boosting [17].

2.3 Advanced Classifier Structures

The original AdaBoost learning produces classifiers consisting of potentially many weak classifiers selected from a large set of weak classifiers. The number can easily

reach up to hundreds and thousands of weak classifiers. For each analyzed sub-window, *all* the selected weak classifiers have to be evaluated ($H(x)$ in Algorithm 1). The computational complexity of detection with classic AdaBoost classifiers is thus lower compared to approaches like SVM or ANN where all supplied features are typically used. Advanced structures of classifiers lower the computational demands even more by evaluating only a part of the weak classifiers. They use the fact that the AdaBoost orders weak classifiers by their importance and that the decision about the sample class can be reached by using only a subset of the most important weak classifiers.

2.3.1 The Attentional Cascade

The reason why the Viola and Jones' approach was so successful was due to the attentional cascade. A pure AdaBoost classifier, which consists of thousands of classifiers cannot be evaluated with the necessary speed. The attentional cascade of classifiers is a mechanism that directs computational power to image areas that are harder to recognize, and ultimately it lowers the computational complexity of detection by several orders of magnitude.

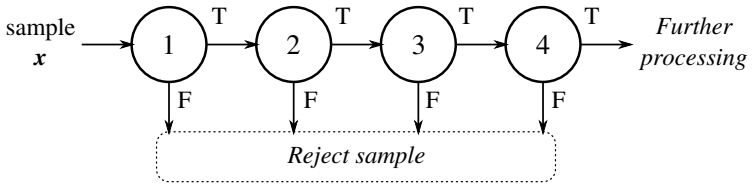


Figure 2.6: Attentional cascade of classifiers. In each node (stage) a sample x is classified. It can be rejected and the classification ends in the node, otherwise it is sent to the subsequent node.

Essentially, the cascade, shown in Fig. 2.6, is a degenerated decision tree, where nodes are classifiers (i.e. they consist of one or more weak hypotheses) which decides if a given sample is background or whether it may be an object. If the decision is an *object*, the sample is passed on to the subsequent node or otherwise it is thrown away. The cascade is trained so that first stages with very low complexity reject rapidly background samples and keep almost all positive samples. Later stages focus on achieving a low false alarm rate. Background samples are thus processed on average by very few weak classifiers, which ultimately lead to a rapid decrease in computational complexity as background samples constitute a majority of samples (typically more than 99 % of analyzed samples).

Fig. 2.7 shows ROC curve of face detectors trained by Viola and Jones [88]. Note that the detectors have very low false alarm rates and high detection date at the same time.

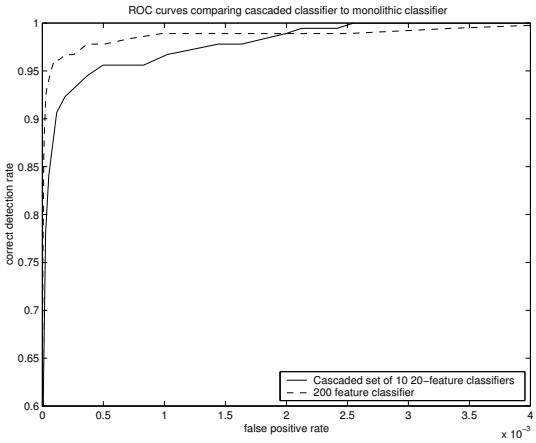


Figure 2.7: ROC curve for Viola and Jones face detector (Source [88]).

2.3.2 Soft Cascade

Although the cascade is an effective algorithmic acceleration of the detection process, it is a rather ad-hoc solution which was later improved [8, 9, 97]. The cascade, however, completely discards information between stages and thus every stage begins classification from scratch even though the classification problem is very similar in subsequent stages. An information propagated from one stage to the next one could help the classification. This idea was used by many authors in order to create better classification structures than the original cascade.

Xiao et al. [93] do not divide a classifier into stages and instead creates one long classifier. This classifier has on some points defined thresholds similar to those in the cascade. Sochman and Matas [82] and Huang et al. [41] use response of previous stage as the first weak classifier of the next stage. Classifiers produced by these methods are shorter and faster than those produced by the original cascade.

The most effective cascade-like detection structure is the soft-cascade by Bourdev and Brandt [6]. The soft-cascade produces one long classifier and sets thresholds after each weak classifier. It is similar to the original cascade where each stage consists of only one weak classifier. The thresholds are calculated after the classifier is learned and are restricted to be exponential functions with one parameter. The free parameter is optimized in order to give the best speed for a given target detection rate. The method for threshold selection proposed by the Bourdev [6] is not very practical nor optimal. Another method that produces the soft-cascade was proposed by the authors Šochman and Matas [83]. In this case the thresholds are set in an optimal way. The algorithm comes with a rigorous theoretical background and produces the fastest classifiers.

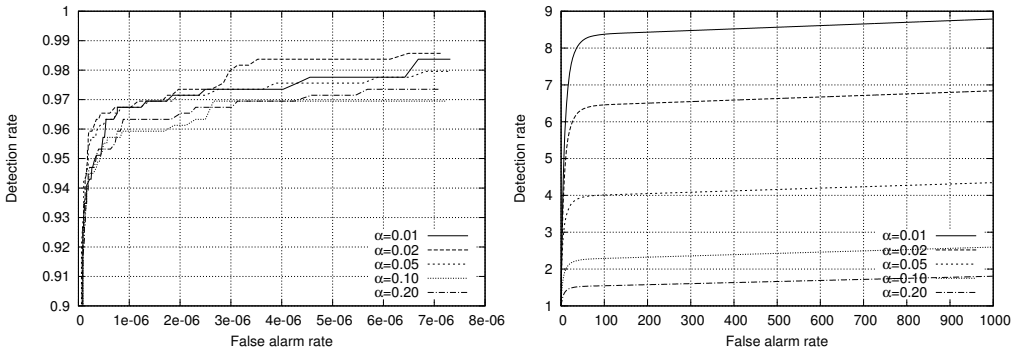
2.3.3 WaldBoost

WaldBoost is built on the top of Real AdaBoost [74, 23] and extends it with Wald's *Sequential Probability Ratio Test* (SPRT) [90]. SPRT is used to generate an optimal sequential decision strategy on measurements (weak classifiers) selected by AdaBoost. Given a weak learner algorithm, training data $\{(x_1, y_1) \ldots, (x_n, y_n)\}, x \in \chi, y \in \{-1, +1\}$ and a target false negative rate α, the WaldBoost algorithm finds a decision strategy with a miss rate lower than the α and the average evaluation time is minimal.

WaldBoost uses real AdaBoost to iteratively select the most informative weak classifiers h_t. The threshold θ_t is then selected in each iteration so that as many negative training samples are rejected as possible while satisfying the *false negative rate* constraint imposed by α.

After learning, the classifier is described by selected hypotheses h_t and thresholds θ_t. The evaluation of the classifier is Equation 2.2 which is a slightly modified version of $H(x)$ from Algorithm 1.

$$\begin{aligned} H_t(x) &= \sum_{i=1}^{t} h_i(x) \\ S_t &= \begin{cases} -1 & \text{when } H_t(x) > \theta_t \\ & \text{Otherwise evaluate } S_{t+1} \end{cases} \end{aligned} \tag{2.2}$$



(a) (b)

Figure 2.8: (a) ROC curves for classifiers with different false negative rate. (b) average speed of the classifiers. One should note that the classifier with $\alpha = 0.2$ calculates less than two weak classifiers per window on average while keeping a very high detection rate.

From the implementation point of view, given the sample x, the classifier sequentially evaluates functions $h_t(x)$, $t \in 0, 1.., T$, and accumulates the strong classifier response H_t. In each step t the H_t is compared to θ_t and the sample is rejected when $H_t(x) < \theta_t$, otherwise the evaluation proceeds with the step $t + 1$.

The WaldBoost algorithm learns an optimal decision strategy for target false negative rate and it can be considered as a state of the art in learning ensemble classifiers. Fig. 2.8 shows ROC curves and average speed plots of face detectors learned with the WaldBoost algorithm with different settings of α.

CHAPTER 3

Feature Extraction

The performance of object detection is to a large extent influenced by the underlying feature extraction method. There are two main properties of features extracted from an image – *a)* descriptive power and *b)* computational complexity. The goal in rapid object detection is to use computationally simple yet, highly descriptive features. In the vast majority of cases, these two properties are mutually exclusive. Thus there are computationally simple features with low descriptive power (e.g. isolated pixels, sum of area intensity) or complex and hard to compute features with high descriptive power (Gabor wavelets [51], HoG [16], SIFT and SURF [57, 4], etc.). A close to ideal approached Viola and Jones [88] with their Haar features calculated in constant time from an integral representation of an image. Another good example can be Local Binary Patterns (LBP) [60, 98] that are very simple to compute while they keep a very high descriptive power.

The feature extraction is strongly application dependent and not all features are suitable for all tasks. For example, Haar-like features and LBPs were successfully applied to the problem of face detection [88, 98], while in the task of pedestrian detection, a better choice is Histograms of Oriented Gradients (HoG) [16] which encode shape rather than image texture. When analyzing videos, features extracted from sequences – dynamic features – can provide important information for detection of target objects [44, 15, 99]. Selection of proper features for a particular task is thus the keystone of a successful application.

This chapter describes the most common feature extraction methods in the field of rapid object detection with boosted classifiers. It should be noted that the list of features is not complete and more feature types suitable for the detection exist.

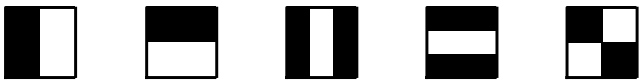


Figure 3.1: Shapes of convolution kernels typically used to calculate a Haar feature response.

3.1 Haar-like Features

The Haar-like features, or simply Haar features, are wavelet features which extract local frequency information. They are based on the theoretical work by A.Haar [27] who described the composition of arbitrary functions from specific wavelets. Similar to Gabor wavelets [51], the Haar wavelets responds to oriented edges and bars in images. It has been argued that this information is crucial for object description [68] as neurons with similar properties were found in primate and human visual system [42, 58, 78]. The Haar features were successfully exploited for feature extraction by Papageorgiou et al. [62, 61] in a general object detection framework and, most notably, by Viola and Jones [88] in their real-time object detection system. The Haar features are the most widely used method feature extraction in many implementations of object detection (both hardware and software) e.g. [88, 91, 85, 25, 50, 7, 43].

The features are based on the calculation of convolution of an input image with Haar wavelet. Typically used wavelets are shown in Fig. 3.1. The great advantage of the Haar features is the simplicity of their evaluation. When using an integral representation of the image [14, 88], each feature can be calculated in constant time regardless of its size. The Lienhart and Maydt [54] extend the integral representation with a new integral image which allow for calculating features rotated by 45 degrees.

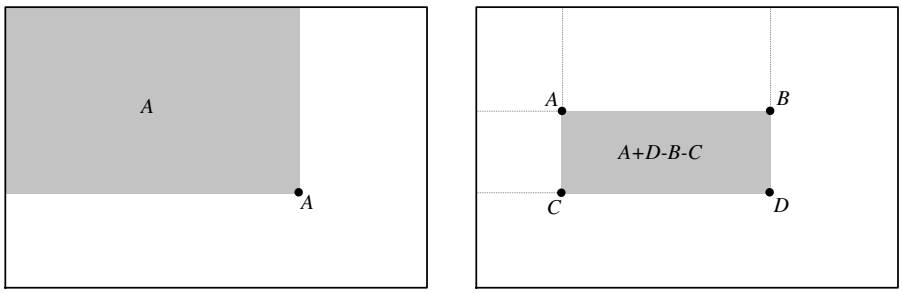


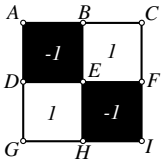
Figure 3.2: Principle of integral representation of an image: (a) point A stores the sum of image values marked by the gray area; (b) the sum of a rectangular area can be achieved by accessing only the values in the corners of the rectangle.

Standard integral image is calculated by (3.1). Every pixel of the integral image stores a sum of pixels in the rectangle defined by image origin and the pixel. Knowing values A, B, C and D in the corners of a rectangular area (see Fig. 3.2 right), the sum s of the area can be obtained by (3.2).

$$I(x,y) = \sum_{u<x, v<y} i(u,v) \tag{3.1}$$

$$s = A + D - B - C \tag{3.2}$$

Summing of rectangular areas is a basic operation needed for a Haar feature response calculation – responses of individual blocks are added together to produce the feature response (Fig. 3.3). The response of an arbitrary Haar feature is then calculated from an integral image by accessing only the corner points of individual feature blocks.



-(A-B-D+E)+(B-C-E+F)+(D-E-G+H)-(E-F-H+I)

-A+2B-C+2D-4E+2F-G+2H-I

Figure 3.3: An Example of the calculation of Haar feature response. Knowing values from an integral image in the corners of the feature blocks (A to I), the response can be calculated by the formulas written on the right side – i.e. simple operations with values from an integral image.

It is obvious that the response of the feature depends on the local contrast and thus it is not invariant to changes in light conditions in the image which is information that is not interesting in a vast majority of cases. The feature response can be normalized in order to suppress this information. The typical choice for normalization is the local energy of the image or local standard deviation of the values in the image.

3.2 Local Binary Patterns

Historically, the Local Binary patterns (LBP) were introduced as a local descriptor for an analysis of static textures [60, 63] and later for analysis of dynamic textures [99]. It has also been successfully used as a feature for face detection [98] and recognition [2, 11] and many other tasks. The feature (see Fig. 3.4) is based on the sampling of the local neighborhood and constructing a binary code from the values of samples. The feature captures the local shape of image intensity but not the intensity and energy itself which makes it invariant to monotonous changes in image intensity.

In the most common form, the feature is evaluated by (3.3), where $\mathbf{v}$ is a set of samples, c is value of the central sample and N is the number of samples.

$$LBP(\mathbf{v}, c) = \sum_{i=0}^{N} (v_i > c) 2^i \tag{3.3}$$

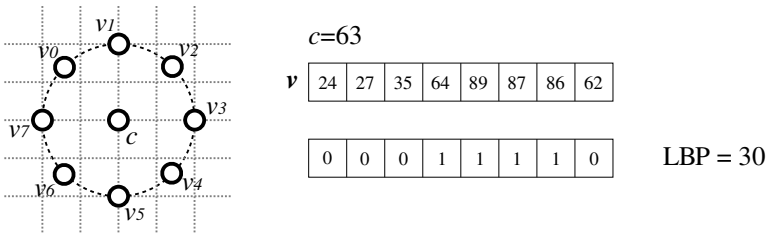


Figure 3.4: Evaluation of the Local Binary Pattern feature. Values of $\mathbf{v}$ and c are obtained from an image and then the feature response is evaluated.

There are other forms of LBP which can be used. The modifications address both, sampling and sample evaluation. The samples do not need to be located in a circle. They can be in a regular grid or placed completely irregularly. The sampling function is not restricted, and convolution with different kernels can be used to obtain the sample values. The number of samples can be changed to create features with more bits. The samples do not need to be compared with thecentral sample by $v_i > c$ and other measure can be chosen (e.g. $|v_i - c| > t$ which yields 1 when a sample is similar to the center and 0 otherwise). By the modification of these parameters, features with different properties can be created. For example, in CS-LBP [30] opposite samples on the perimeter of the circle are compared creating a 4 bit code. Kalal [49] uses simple 2 bit LBP for image gradient estimation.

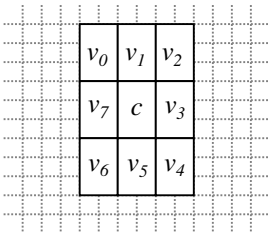


Figure 3.5: Multi-block LBP feature. Samples are taken by convolution of the image with a rectangular kernel. In this case a 2×3 pixel kernel is used.

In the field of object detection, the most common choice of samples is a convolution with a rectangular kernel (see Fig. 3.5), which can be obtained from an integral representation of an image in constant time [98].

3.3 Local Rank Functions

The *Local Rank Function* (LRF) image features [94, 39, 34] were developed at the Faculty of Information Technology as an alternative to the commonly used state of the art image features such as LBP and Haar. The features were designed with their

hardware implementation on mind. Their implementation by circuitry is very simple and their performance is comparable to the above mentioned features.

The LRF features are based on the formalism described in [34] which evaluates the response of a feature as a function of ranks of a selected subset of coefficients obtained from an image by a *sampling function*. This formalism is very general and allows for the construction of many different types of features with different properties. The feature is defined by the image sampling function, positions of samples in the image, selection of ranked coefficients and the function evaluating the response.

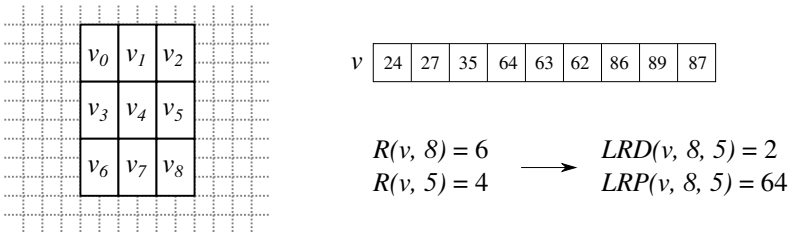


Figure 3.6: The evaluation of the LRF-based features with two ranks. Samples $\mathbf{v}$ first obtained from an image, ranks of selected items are calculated and finally the feature response is calculated using a formula for the particular feature type.

A reasonable choice for image features is rectangular samples located in a regular 3×3 grid as shown in Fig. 3.6. This configuration resembles a multi-block LBP feature [98]. Samples taken from the image form a vector of coefficients $\mathbf{v}$ from which the feature is evaluated using ranks of selected items. The rank of an item is calculated by (3.4) which calculates the position of k-th item in an ordered sequence of $\mathbf{v}$.

$$R(\mathbf{v}, k) = \sum \begin{cases} 1, & \text{if } v_k < v_i \\ 0, & \text{Otherwise} \end{cases} \tag{3.4}$$

Functions evaluating a feature response are shown in 3.5. The simplest feature *Local Rank* just returns the rank of an item which is an integer from interval $\langle 0, 8 \rangle$. More complex one are *Local Rank Differences* which returns the difference of ranks of two items (which is in the range $\langle -8, 8 \rangle$), and *Local Rank Patterns* which return rank values concatenated to a single integer number in the range $\langle 0, 99 \rangle$.

$$LR(\mathbf{v}, a) = R(\mathbf{v}, a) \tag{3.5a}$$

$$LRD(\mathbf{v}, a, b) = R(\mathbf{v}, a) - R(\mathbf{v}, b) \tag{3.5b}$$

$$LRP(\mathbf{v}, a, b) = 10R(\mathbf{v}, a) + R(\mathbf{v}, b) \tag{3.5c}$$

Features based on local ranks were successfully used in rapid object detection [34] with a performance similar to Haar-like features and LBP.

3.4 Histograms of Oriented Gradients

Another feature extraction method frequently used in pattern recognition is the Histogram of oriented gradients (HoG). The feature, in the simplest form, captures the distribution of image gradient orientations in a local image area discarding their spatial relations [16, 100]. In principle, the feature first collects a histogram where each bin reflects the amount of gradients in the particular direction in the area defined by the feature. The histogram is then processed in order to evaluate the feature response.

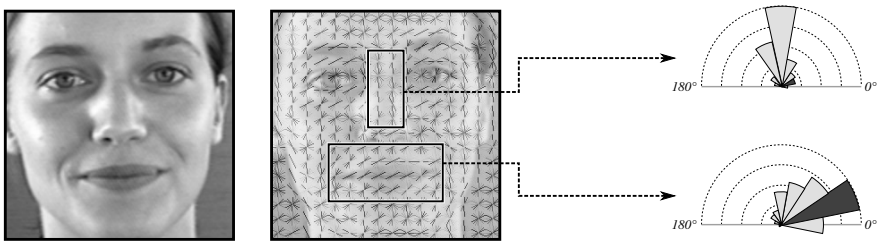


Figure 3.7: Calculation of two different HoG features. On the left is original image. In the middle, local histograms are superimposed on the original image. Areas of features is marked by the rectangles. On the right are histograms corresponding to the features. In this case, the response of the features is the value of the second histogram bin marked by the dark gray. (Image source: BioID face database)

Image gradients can be calculated by various methods [21]. The simplest way is to obtain image derivatives in x and y directions by Equations (3.6) and calculate gradient direction θ and magnitude r for each pixel by Equation (3.7). The histogram of magnitudes in different directions is then created.

$$I_x = I * \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \qquad I_y = I * \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}^T \tag{3.6}$$

$$r = \sqrt{I_x^2 + I_y^2} \qquad \theta = atan\left(\frac{I_y}{I_x}\right) \tag{3.7}$$

The histogram is then processed in order to obtain the feature response. Multiple choices on how to calculate the response exist. The most obvious one is to select one bin and take its normalized value (shown in Fig. 3.7). Or the response can be an orientation with the highest magnitude. In some applications (especially recognition tasks), the whole histogram can serve as a feature.

3.5 Other Feature Types

Sparse Granular Features

Feature extraction methods mentioned above are rather standard and often used in practical applications. However, this field is still explored in order to search for more efficient extractors in terms of computational complexity and descriptive power.

Sparse Granular Features proposed by Huang et al. [40] are a generalization of Haar features. In this case a feature is composed from blocks whose meaning is the same as in Haar features. The difference is that the number of blocks is not restricted and the block weights are not restricted either. Such features are very close to the general convolution filter.

Figure 3.8: First four features selected by Vector Boosting and optimized by the heuristic algorithm (source [40]).

The number of all features that can be constructed in a given window is extremely large due to the amount of free parameters – number of blocks, weights of blocks, block positions and sizes. The number of features is far larger than the number of Haar features) and thus it is not possible (or it is not reasonable in most cases) to search exhaustively for the best feature in the complete set. Huang et al. used a heuristic search method that is initialized by a set of Haar features which is iteratively adapted by moving the blocks, adding new blocks, deleting the blocks, etc.

3D Haar Features

The natural generalization of Haar wavelets from 2D to 3D space was proposed by Cui et al. [15]. Instead of an integral image, the authors compute integral volume, in which the sums of 3D blocks can be easily calculated.

Shapes of features are shown in Figure 3.9. This variant of Haar features provides information about changes of appearance in time and thus can be in certain applications more robust than purely static Haar features.

Patterns of Motion and Appearance

Another generalization of Haar features are patterns of motion and appearance proposed by Jones et al. [44]. These features operate on the differences of image pairs. From two consecutive images, five differentials are generated – the difference

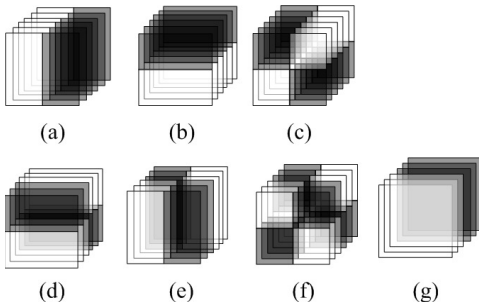


Figure 3.9: Seven types of 3D haar features (Source [15]).

of the first frame and one pixels shifts of the second frame – Δ, L, R, U, D frames shown in Figure 3.10. These differential images give elemental motion information on which feature responses are calculated.

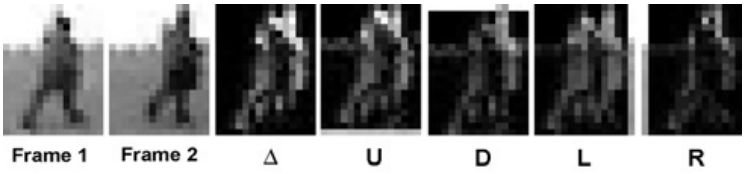


Figure 3.10: Source images on the left are processed and differentials are generated (source [44]).

The features are defined as rectangular regions which calculate the difference between Δ and one of L, R, U, D frames, or Haar wavelets calculated in the differential or intensity frames. These features thus provide information about the motion and appearance of certain areas of image. These features are efficient in applications where motion is an important property of detected objects (e.g. pedestrian detection).

Gabor Wavelets

Convolution of an input image with a Gabor wavelet [51] (see Figure 3.11) can be used as a feature. Similar to Haar features, the Gabor wavelets provide a response to local frequencies with a good tradeoff between spatial and frequency localization. They are computationally costly and do not seem to be more effective than Haar features in practical applications of object detection [13]. They have, however, interesting applications in the field of object recognition and categorization [68, 78, 76, 77].

PCA, LDA and others

Other linear functions that can be used as features are projections obtained through PCA or LDA [5]. More advanced methods of feature extraction exist. The features do not need to be pre-defined, and they can be *learned* from data in order to get features that are fine-tuned for detection or recognition of a particular object type

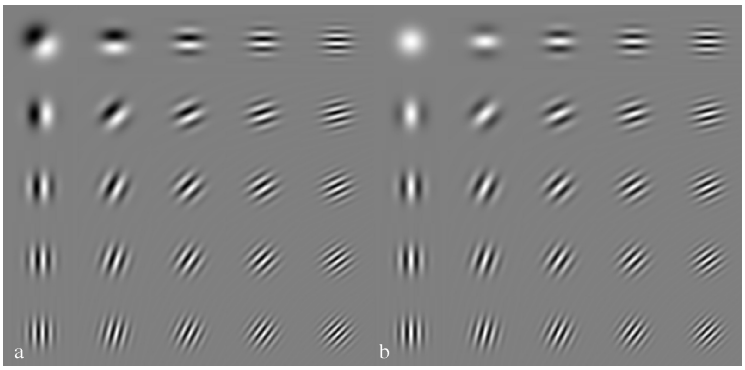


Figure 3.11: Example of gabor wavelets (source [51]).

[1, 20, 69]. Such features are very effective but, similar to Gabor wavelets, they are computationally costly as they often calculate general convolution with an input image or do other transformations.

CHAPTER 4

Architectures and Acceleration

Viola and Jones [88] developed the first system able to operate in real-time (i.e. with a sufficient frame rate and low delay). They reported 15 frames per second which was at the time about fifteen times faster than the comparable state of the art system [70]. Other systems used classifications schemes and feature extraction methods which were too demanding in terms of computational resources. Although the detection scheme employed by AdaBoost Cascade or WaldBoost is very effective, it is suitable for real-time applications only in certain conditions (e.g. low resolution image, low frame rate, etc.). When detection of small objects is required, high resolution images have to be analyzed. For the detection of fast events, high speed video streams have to by processed. In surveillance systems, detection of multiple types of objects and processing of multiple streams are required. All these applications are computationaly demanding and using some acceleration techniques would improve their performance.

Acceleration can be done in several ways. On the algorithmic level, the speed-up can be achieved by modification of classifier structure, for example cascade classifiers is more optimal than pure AdaBoost. Another modification is that the classifier can use information from the current sub-window in order to predict responses on neighboring positions of a sub-window [95]. Other approaches are possible as well.

Besides the algorithmic acceleration, the implementation can be fine-tuned to better exploit the underlying computational architecture. Most of the contemporary CPUs offer two or more computing cores. This allows for execution of a parallel code to accelerate computations. The CPUs also offer advanced instructions such as SSE in the Intel CPUs, or NEON in ARM, which are not typically directly used in compilers (due to lack of constructs in source programming languages), but they are used rather for automatic optimization of some operations (such as simple loops).

Such instructions can be used as well for the acceleration of some computations during object detection.

Another architecture that can be used with benefit for the acceleration is graphics hardware. From an historical point of view, the graphics hardware was used for processing of geometrical primitives (vertexes, lines, triangles, etc.) and rasterize them on a screen. The best known interfaces for the graphics hardware control are SGI's OpenGL and Microsoft's DirectX. The GPUs process geometry using *shaders* – small sub-routines executed on the graphics processor. Shaders were at first restricted so as to be very simple. As the technology went forward, most restrictions were dropped, but the shaders could still process only geometry and raster images. Recent advances in GPU technology dropped the restrictions completely and user programs can execute unrestricted code. Graphics hardware which is able to execute such a code is called *General Purpose GPU* or GP-GPU. Example of GP-GPU architecture is nVidia's CUDA (Compute Unified Device Architecture). Most power of the GP-GPU is in parallelism. The code is organized in *kernels* which are logically organizes in grids and blocks and automatically mapped to the computing elements in the graphics processor.

Of course, there is the possibility to implement the detection in custom hardware such as FPGA or ASIC chips. In such architectures, special hardware structures can be created in order to compute a classifier response. It is, however, tricky to implement the detection directly in such devices as the designer has to cope with limited memory and other resource limitations. The performance of the resulting hardware unit is typically comparable with the best PC and GPU implementations. The benefits of hardware implementation include low power consumption and the possibility of integration in embedded systems.

4.1 Levels of Acceleration

In order to better understand where the main sources of acceleration of object detection are, it is necessary to describe the detection algorithm which is basically an application of WaldBoost Equation (2.2) from Section 2.3.3 on every position of the input image. The algorithm, described in the Listing 4.1, contains three key functions – `eval_stage` for evaluation of a response of a weak classifier on a particular position in the input image; `eval_classifier` for evaluation of the response of the strong classifier on a particular position in the image; and `scan_image` for the processing of the whole input image. The results of image scanning are positive responses which are stored in list `r`.

```
float eval_stage(Image img, Stage s, int x, int y)
{
    int response;
```

```
    // 's' contains feature parameters:
    // - position relative to x,y, size, etc.
    // calculate feature response according to parameters in 's'
    // e.g. LRD, LRP, LBP or other
    return s.prediction[response];
}

float eval_classifier(Image img, Classifier c, int x, int y)
{
    float response = 0.0;
    for (int i = 0; i < c.stage_count; ++i)
    {
        Stage s = c.stages[i];
        response += eval_stage(img, s, x, y);
        if (response < s.threshold)
        {
            return response;
        }
    }
    return response;
}

void scan_image(Image img, Classifier c, Results r)
{
    r.clear();
    for (int y = 0; y < img.height - c.height; ++y)
        for (int x = 0; x < img.width - c.width; ++x)
        {
            float response = eval_classifier(img, c, x, y);
            if (response > 0)
            {
                r.add(x, y, response);
            }
        }
    // 'r' contains list of positive detections
}
```

Listing 4.1: Pseudocode of the detection with all important parts.

4.1.1 Parallelization on Multi-core Architectures

On the multi-core architectures a process can be parallelized by the control-flow transformation employing more computing elements. These are pretty straightforward methods for performance improvement which can be exploited by rather simple means (like threads, OpenMP [12], Intel Thread Building Blocks (TBB) [67], Posix threads, etc). This parallelization can occur in the `scan_image` or `eval_classifier` functions, or even above the `scan_image` function, taking into account the framework in Listing

4.1.

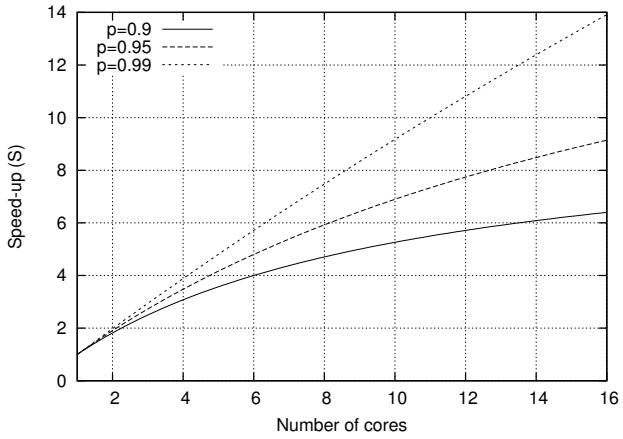


Figure 4.1: Speed-up limit on multi-core architectures as a function of the number of processing elements. For the highly parallelizable problems (i.e. those with $p \approx 1$) the speed-up S is nearly linear.

The speed-up can be described in the terms of Amdahl's law (see Fig. 4.1), and it is limited by Equation (4.1) where s is the speed-up of particular parallelized part and p fraction of computations performed by the non-parallelized part. When p is close to 1 (i.e. most computations are done in parallel), the speed-up is nearly linear with the number of computing elements.

$$S = \frac{1}{(1-p) + \frac{p}{s}} \tag{4.1}$$

Processing of multiple frames

Speed-up in this case, illustrated in Fig. 4.2, is almost equal to the number of frames processed concurrently as the overhead is minimal. It could be achieved by the parallel execution of the `scan_image` function.

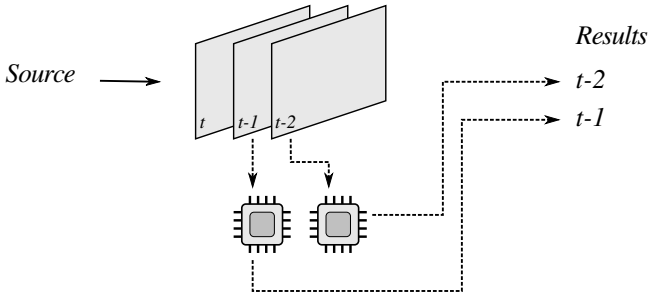


Figure 4.2: Frames going from a source (camera, video, etc.) are distributed between processing nodes. Each node processes the whole frame and returns results.

Processing multiple sub-windows

In this case, illustrated in Fig. 4.3, the input image is divided into non-overlapping parts which are processed by dedicated processing elements (i.e parallel exection of `eval_classifier` function on different image locations). The results can be collected separately for each part and merged after the frame is processed, or they can be written to shared memory. This method is suitable for multi-core CPUs [29] and especially for architectures such as GP-GPU where hundreds of general-purpose computing elements are available [32].

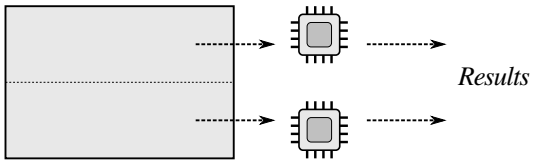


Figure 4.3: The source image is divided into parts, and each part is processed by a processing node.

Processing of multiple weak classifiers in the same sub-window

In this case, illustrated in Fig. 4.4, the whole image is processed by the sequential control flow (i.e. sub-windows are processed one-by-one) and features evaluated within the window are passed to different computing elements. This is beneficial in schemes such as AdaBoost where the known number of features is evaluated in each window. In soft-cascades, however, the number of features evaluated in the window is not known in advance and thus this kind of parallelization is problematic to use.

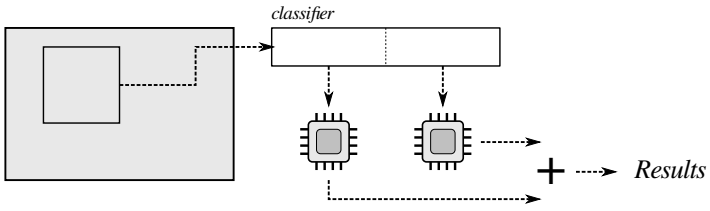


Figure 4.4: During the classification of a sub-window, different parts of classifier are processed by dedicated computing nodes. The results of the nodes are merged, producing classification result for the sub-window.

4.1.2 Acceleration on SIMD Architectures

Beside the traditional execution of a code on more computing elements, there is data parallelism where more data items (sometimes called vectors) are processed by one instruction within a computing element – SIMD (Single Instruction Multiple Data).

CPUs typically contain standard instruction set which processes integers and floats. This set is extended with a set of vector instructions which work over vectors of data stored in registers. Vector instructions typically include standard arithmetic and logic instructions, instructions for data access and other data manipulation instructions (load/store, packing, unpacking, etc.). This is the case of general purpose CPUs like Intel, AMD or PowerPC. Besides the general purpose CPUs, there are GP-GPU, successors of traditional GPUs that can execute parallel kernels on data and be viewed as advanced SIMD processors.

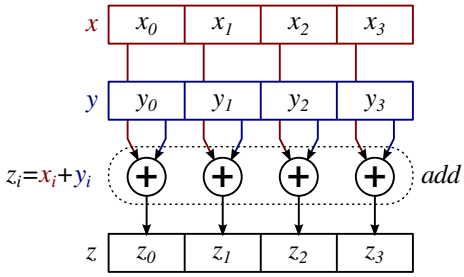


Figure 4.5: Vector instruction for adding in 4-wide SIMD unit. The input x and y is treated as a vector with 4 items which are processed and output z is calculated.

Probably the most influential and widespread SIMD architecture is used in Intel processors and their SSE (Streaming SIMD Extension) instruction set. The predecessor of the SSE dates back to 1998 when Intel introduced their MMX (Multi Media Extension) instruction set which was similar to SPARC VIS or MIPS MDMX. These instructions extended standard x86 instruction set with 60 vector instructions working over 64 bit registers. In the SSE, the MMX functionality was preserved and the set was extended with new vector instructions, presently working with 128 bit vectors. Intel progressively updated the set in their new CPUs reaching over 200 vector instructions in SSE4. The instruction sets are upgraded with new instruction sets such as Advanced Vector Extensions (AVX), or Fused Multiply-Add (FMA).

The SIMD principles can be used with great benefits in applications where large data is processed, for example image processing (FIR filtering). Each instruction can load a bunch of data and apply an operation on them. Then, for example, memory copying can be accelerated by moving blocks of data instead of individual bytes. On the other hand, not all algorithms can be vectorized as the flow control depends on the data in many cases.

Most programming languages, however, have no constructs which enable vector processing and the compiler has to figure out itself where the SIMD can be used – auto-vectorization. Such a technique is applicable for simple loops (such as matrix multiplication and dot product, etc.). The code has to be sometimes rewritten to employ the data parallelism (e.g. using assembler) and in some cases, the whole

problem has to be re-structured in order to make it more convenient for SIMD processing. This implies inevitable human labor.

Data Parallelism in Object Detection

The main application of data parallelism is in feature extraction. There are two main options on how to employ it. First, keep the evaluation code *as is* and calculate N spatially close responses simultaneously. While this method is pretty clear, it causes some problems since SIMD architectures may be sensitive to data alignment and unaligned memory references may be slow. The second method is to calculate only a single feature at the time and process *all feature data* in parallel using SIMD instructions. This method also brings problems but they are solved by a simple rearrangement of image pixels in memory. The advantage of this approach is that all features can be processed by a fixed instruction chain *without* loops [31, 46]. Application of SIMD (particularly Intel SSE) for feature extraction is discussed in detail in Section 5.

4.2 Graphics Hardware

Implementation of object detection in GPU was historically done using programmable shaders [64, 33]; however, contemporary state of the art is in GP-GPU programming languages, such as CUDA or OpenCL [32]. GP-GPUs programmed using one of these languages present one of the most powerful and efficient computational devices. When used for object detection, GP-GPUs can be seen as a SIMD device with a high level of parallelism. Unfortunately, the high level of parallelism is difficult to employ in WaldBoost detection as the amount of computation in adjacent positions in the image is not correlated and in general is quite unpredictable, which in fact heavily complicates the usage of the computational resources.

The efficient implementation of object detection using CUDA [32] solves problems of two main domains: the classifier operating on one fixed-size window, and parallel execution of this classifier on different locations of the input image. The problem of object detection by statistical classifiers can be divided into these steps:

- loading and representing the classifier data
- image pre-processing
- classifier evaluation
- retrieving results

The constant data containing the classifier (image features' parameters, prediction values of the weak hypotheses summed by the algorithm and WaldBoost thresholds)

could be accommodated in texture memory or constant memory of the CUDA architecture. These data are accessed during the evaluation of each feature at each position, so the demands for access speed are critical. Programs that are run on the graphics hardware using CUDA are executed as *kernels*; where each kernel has a number of blocks and each block is further organized into threads. The code of the threads consumes hardware resources: registers and shared memory. This limits the number of threads that can be efficiently executed in a block (both the maximal and minimal number of threads).

One thread computes one or more locations of the scanning window in the image. The image window locations are therefore divided into rectangular tiles which are solved by different thread blocks. Experiments showed that the suitable number of threads per block was around 128. Executing blocks for only 128 pixels of the image would not be efficient, so we chose that one thread calculated more than one position of window – a whole column of pixels in a rectangular tile. A good consequence of this layout is the easy control of the resources used by one block: the number of threads is determined by the width of the tile, and the height controls the whole number of processed window positions by the block. The tile can extend over the whole height of the image or just a part of it.

When the kernel is started, the image data is referenced by texturing units from the multi-resolution pyramid and the parameters of the classifier are read from the constant memory. When the window position is recognized as the searched object, the coordinates are written into the global memory. In order to avoid collisions of concurrently running threads and blocks, atomic increment (`atomicInc()`) of one shared word in the global memory is used for synchronization. This operation is rather costly, but the positive detections are so rare that this means of output can be afforded. As a consequence, the results of the whole process are at the end available in one spot of the global memory, which can be easily made available on the host computer.

The main property of the CUDA implementation is that the CUDA outperforms the CPU implementation mainly for high resolution videos. This can be explained by extra overhead connected with transferring the image to the GPU, starting the kernel programs, retrieving the results, etc. These overhead operations typically consume constant time independent of the problem size, so they are better amortized in high-resolution videos.

4.3 Programmable Hardware

The run-time for object detection does not necessarily have to be only implemented in software; programmable hardware is one of the options as well, namely Field Programmable Gate Arrays (FPGA) [92, 91, 85, 96, 25, 36, 50, 43]. While the

algorithms of object detection are in principle the same for software and hardware, the hardware platform offers features largely different from the software and thus the optimal methods to implement detection in programmable hardware are often different from the ones used in software. In many cases, the hardware implementation can be more efficient than the software implementation.

The key features, important for object detection that are beneficial for hardware implementation, include: *massive parallelism* achievable with good performance/-electrical power ratio, *variable data path width* in hardware adjustable to exact algorithmic needs, *simple implementation of bit manipulation and logical functions*, and nearly seamless *complex control and data flow implementation.* Of course, the hardware implementation also has severe limitations, the most important being the limited complexity of the hardware circuits, expensive computational resources for complex mathematical functions, relatively limited memory structures and, in most cases, lower clock speed comparing to the processors.

Complete Detection in Programmable Hardware

The typical methods of complete object detection in programmable hardware are feasible to implement using a sequential engine (possibly micro-programmable), that performs detection location by location, weak classifier by weak classifier until a decision is reached. As the evaluation of each weak classifier is relatively complex, the operation of the sequential unit is pipelined and so that several instances can be running in parallel. Different image locations, in general, require a different number of weak classifiers in oredr to be evaluated. These facts lead to relatively complex timing and synchronization of processing; however, very good performance can be achieved [96].

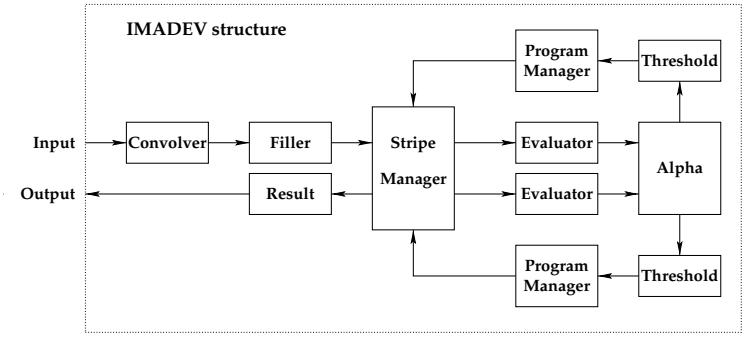


Figure 4.6: Block structure of object detection circuit proposed in [96].

Synthesized Object Detection Circuits

In a situation, where complete evaluation of the detection is not required (e.g. in the case when a powerful CPU is available), programmable hardware can be exploited

for pre-processing rather than for complete detection. Probably the best approach is synthesis of fixed-function circuits from the results of the machine learning process on demand for each classifier. Such a synthesized circuit is most efficient when processing small fixed number of weak classifiers for every evaluated position. While some of the weak classifiers are in such case evaluated unnecessarily (assuming the WaldBoost based classifiers), the average price of weak classifier implementation is still often much lower than in the sequential machine described above. The main advantage of this approach is that all the weak classifiers can be evaluated in parallel. However, as each weak classifier consumes chip resources, only a very small number of weak classifiers can be implemented this way.

4.4 Algorithmic Accelerations

Exploiting Neighbors for Faster Detection

In scanning window object detection using a soft cascade detector, each image sub-window is processed independently. However, much information is shared between neighboring positions and utilizing this information can increase the speed of detection. In order to utilize the shared information, a *suppression classifiers* [95] can be learned to predict the responses of the original detection classier at neighboring positions. Computation of the original detector can then be suppressed at positions for which this prediction is negative and confident enough (see Fig. 4.7).

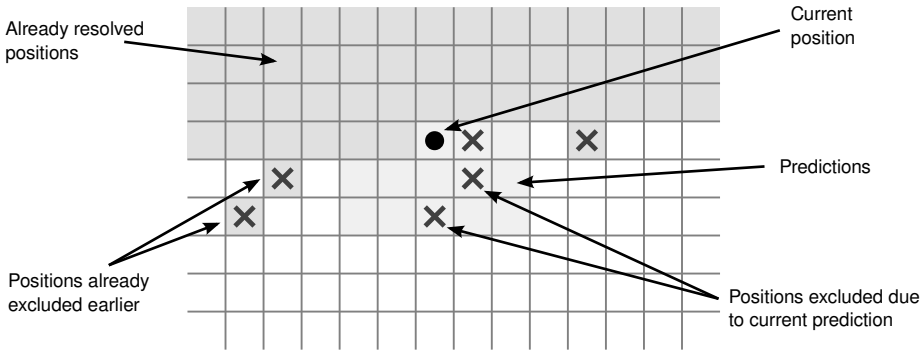


Figure 4.7: Principle of neighborhood suppression. The classifier evaluated on a position and prediction of decisions for neighboring positions.

In the case of *space partitioning weak hypotheses* (see Section 2.3.3), the suppression classifiers can be made computationally very efficient by re-using the features h_t computed by the original classifier. In that case, adding the suppression classifiers just increases size of the look-up table $l : \mathbb{N} \to \mathbb{R}$. As a result, the response of every feature is transformed by a set of look-up tables in order to obtain responses on both the current position and the neighboring positions. Using suppression classifier

can decrease the amount of computations, even several times without the loss of detection performance.

Early Non-Maxima Suppression

Detection of objects by a scanning window usually employs some kind of *non-maxima suppression* in order to select a position with the highest classifier response from a small neighborhood in position and scale. The suppressed detections have no influence on the resulting detection and it may not be necessary to compute the detectors completely in these positions. In other applications only the highest response on a number of samples is of interest as well. Examples of such applications are speaker and person recognition where a short utterance or face image is matched by a classifier to templates from a database.

The main idea of *Early non-Maxima Suppression* [32] (EnMS) is to perform non-maxima suppression already during computation of classifiers and stop computing classifiers for objects having very low probability that they will reach the best score in the set of the competing objects. EnMS employs learning framework similar to WaldBoost, using the Conditioned Sequential Probability Ratio Test [32].

CHAPTER 5

Exploitation of SIMD for Feature Response Computation

Object detection speed is one of the important properties of an implementation of object detection run-time. Speed is dependent on the classification algorithm, image features used, image representation, implementation of the feature extraction algorithm and other possible aspects. In the soft cascade detectors, the feature extraction constitutes a vast majority of computations during the object detection and thus speed-up of the extraction algorithm speeds up the whole detection.

This section presents methods that use properties of SIMD architectures for the acceleration of an evaluation of a feature response. It is focused around the WaldBoost classification scheme and LBP, LRP and LRD low-level features which are especially SIMD-friendly. Ideas presented in this section are mainly from [31, 34, 46] with some added new ideas.

5.1 Representation of a Classifier

5.1.1 Features and Weak Classifiers

Three types of features are primarily considered in this work – Local Binary Patterns, Local Rank Differences and Local Rank Patterns. The methods presented in the following sections are not, however, constrained to use these features, which are used rather as suitable examples. Only space partitioning weak classifiers are considered in this thesis since they exhibit good results and are easy to implement by using look-up tables.

The previous definitions of features in Sections 3.2 and 3.3 were fairly general – they allowed for using a various number of image samples, spatial arrangements of the samples, sampling functions, etc. A more practical definition from the implementation

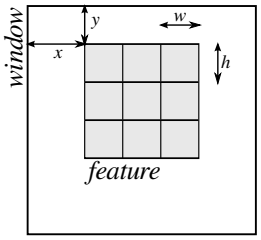


Figure 5.1: Parametrization of features. The position x, y is given relative to an analyzed window and the size of the feature block w, h is given in pixels.

	min	max
LBP	0	255
LRD	-8	8
LRP	0	99

Table 5.1: Studied types of features and their response ranges.

point of view follows. The features are evaluated from a set of spatially close samples taken from an input image arranged in a 3×3 regular grid. Sampling function is a convolution with a rectangular kernel. The image feature, as shown in Fig. 5.1, is composed from nine blocks, and the samples can be expressed as sums of pixels which fall into the particular block. These samples are then evaluated by a feature evaluation formula which outputs the response of the feature. Possible responses of such features are summarized in Table 5.1.

The feature is parametrized by its position x, y relative to the classified window and by the size of its blocks w, h, and possibly by other parameters needed for evaluation (e.g. rank indices in LRD and LRP). Response of a feature is used as an input for the weak classifier which for each possible response of the feature assigns a real number.

```
float eval_lbp_stage(Image img, Stage s, int x, int y)
{
    int samples[9];
    // Gather samples defined by feature
    // position and size: s.x, s.y, s.w, s.h

    // Critical part of the evaluation
    int lbp = 0;
    for (int i = 0; i < 8; ++i)
    {
        lbp |= (samples[i] > samples[4]) << i;
    }

    return s.prediction[lbp]; // response of weak classifier
}
```

```
float eval_lrd_stage(Image img, Stage s, int x, int y)
{
    int samples[9];
    // Gather samples

    int countA=0; countB=0;
    for (int i = 0; i < 9; ++i)
    {
        if (samples[i] > samples[s.A]) countA++;
        if (samples[i] > samples[s.B]) countB++;
    }
    return s.prediction[(countB - countA) + 8];
}

float eval_lrp_stage(Image img, Stage s, int x, int y)
{
    int samples[9];
    // Gather samples

    int countA=0; countB=0;
    for (int i = 0; i < 9; ++i)
    {
        if (samples[i] > samples[s.A]) countA++;
        if (samples[i] > samples[s.B]) countB++;
    }
    return s.prediction[10 * countB + countA];
}
```

Listing 5.1: LBP, LRD and LRP evaluation codes as typically implemented.

The Listing 5.1 extends the code in Listing 4.1 with functions for the evaluation of features – implementation of feature extraction algorithms described previously in Section 3.2 and 3.3. The sample gathering phase obtains samples from an image. The particular method depends on the type of the input image and its pre-processing. The rest of the code, the loop, calculates the response. It should be noted that the feature evaluation can be repeated even millions of times during the detection, and doing this inefficiently results in a serious bottleneck.

5.1.2 Strong Classifier

A strong classifier learned by the WaldBoost algorithm contains an ordered set of weak classifiers, each containing one image feature. The classifier, illustrated in Fig. 5.2, is parametrized typically by the size of the detection window, number of the weak classifiers and type of features.

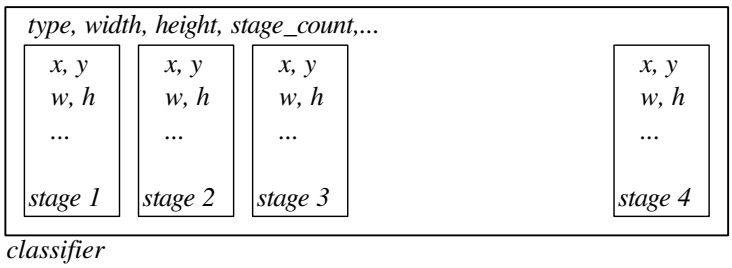


Figure 5.2: Schematic view of a strong classifier which is basically a sequence of weak hypotheses and their parameters.

5.2 Input Pre-processing and Data Access

Lets focus on what is the input for the feature extraction. The input image is in the form of an intensity raster. This can be pre-processed in order to make the representation clearer for the detection and to simplify the detection by the pre-calculation of necessary values. In our case, several types of pre-processing make sense.

1. *No pre-processing* – Evaluation directly on the intensity image. The coefficients for feature evaluation can be calculated directly by summing the intensities in the image. While this is suitable for small features, large features are computed very slowly. This approach is not efficient either in terms of computational complexity or memory access.

2. *Integral image* – An integral image allows for the summing of an arbitrary rectangular area in constant time and is more suitable for feature evaluation. It is advantageously used for the calculation of Haar-like features. Each rectangular area can be summed by accessing its corners in the integral image. On the other hand, it is efficient for features of larger sizes which are not necessary in detectors.

3. *Convolution images* – The best choice (in most cases) is to pre-calculate sampling functions. Feature evaluation is then only a matter of the loading of the coefficients and their processing.

4. *Pre-calculated feature* – Some feature types are allowed to be pre-calculated in order to minimize the amount of computations in the detection phase. In the case of this thesis, the LBP feature response can be pre-calculated for every position in the image.

5.2.1 Evaluation without Pre-Processing

The intensity image can be used for a feature evaluation without any pre-processing in an obvious way. Nine samples, defined by feature parameters x, y, w, h, has to be obtained from image f – pixels belonging to the feature blocks are simply summed up by Equation 5.1, producing values for feature evaluation.

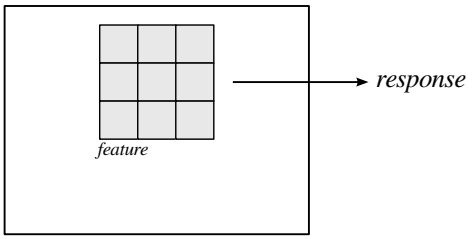


Figure 5.3: Features can be evaluated directly on an intensity image by summing pixels in the blocks.

In Equation 5.1, the m and n are horizontal and vertical indices of block ($m \in \{0, 1, 2\}, n \in \{0, 1, 2\}$) and the values of S are used as samples for feature evaluation.

$$S^{m,n} = \sum_{u=0}^{w-1} \sum_{v=0}^{h-1} f(x + mw + u, y + nh + v) \tag{5.1}$$

Although the simplest, this method is the worst one in most cases. The complexity of a sample calculation grows linearly with the number of pixels in a feature block. This is efficient for only very small features (up to 2×2 pixels per block). This method is selected as a reference in this thesis since only small features are considered.

5.2.2 Using Integral Image

Pre-processing with an integral image is known as a solution that allows for summing arbitrary areas of the input in constant time. By accessing only corner pixels of each area, values can be summed.

The integral image, defined in Section 3.1 by Equation 3.1, can be used by an application of Equation 3.2 to each block. This means that the 16 values in the corners of feature blocks (see Fig. 5.4) has to be accessed in order to obtain the nine values for feature evaluation. Coordinates S of the pixels in integral image F are defined by Equation 5.2.

$$S = \{x, x + w, x + 2w, x + 3w\} \times \{y, y + h, y + 2h, y + 3h\} \tag{5.2}$$

The integral image is especially convenient for the evaluation of large features. Small features, however, are calculated with less efficiency.

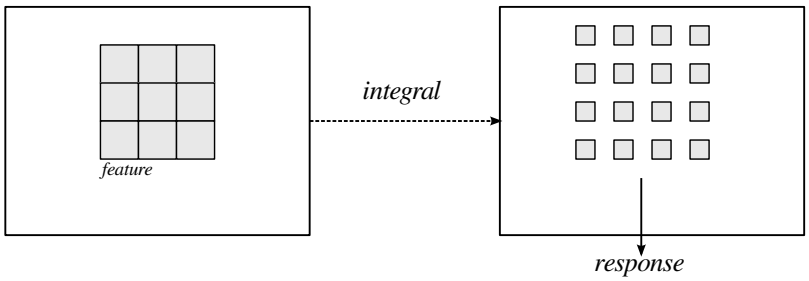


Figure 5.4: Feature evaluation on an integral image. The feature represented by blocks in intensity image (left), is represented by a set of samples in the integral image (right). From these samples, a response can be calculated.

5.2.3 Using Convolution Images

A sampling function for feature evaluation can be pre-calculated as well (i.e. for each type of sampling, an image with sample values can be created). This necessarily means that for each sampling function (feature size) a new image has to be calculated which means some overhead. This overhead is lowered by the fact that the sample values do not need to be calculated for each feature during the classification phase.

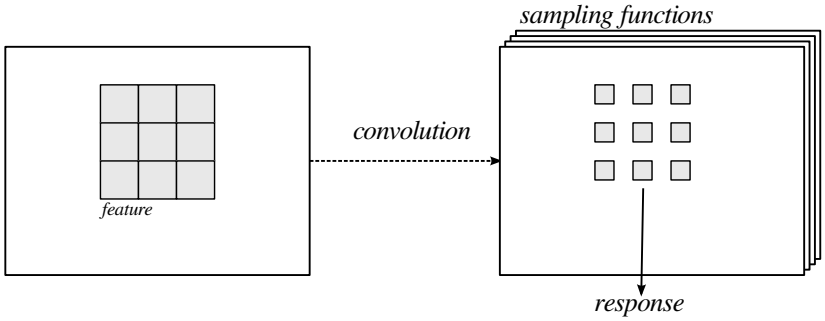


Figure 5.5: Feature evaluation on a pre-convolved image. First, the intensity image (left) is convolved with all needed sampling functions and convolved images (right) are created. A feature is then represented as a set of nine pixels in a particular image.

A sampling function $c = f * g$ is pre-calculated by the convolution of the input image f with kernel $g^{w.h}$ representing a sample. Feature is parametrized by its position in an image x, y and its size w, h which is also size of the convolution kernel g.

$$S = \{x, x + w, x + 2w\} \times \{y, y + h, y + 2h\} \tag{5.3}$$

A feature in this image is then always represented by values in coordinates defined by Cartesian product S in Equation 5.3. Therefore, obtaining feature data has constant complexity. The drawback is the necessity to pre-calculate an image for

every feature size and thus it is important to keep the number of different sizes low.

Block Re-arrangement

Convolved images can be stored (or rather rearranged) in the memory in a special way to put pixels belonging to the same feature close together in order to minimize the number of memory accesses and to make further computations more SIMD friendly.

The block rearrangement stores responses produced by the adjacent positions of the convolution kernel in consecutive memory addresses. Such a memory layout can be viewed as an image, where the responses produced by the adjacent positions of convolution kernel are its sub-images. Although, the data is localized better, which is important for SIMD processing, there is a need for a more complex addressing which has to take into account the block structure of the image. Given a feature with parameters x, y, w, h, where x, y is the *absolute* feature position in an image. The data for the feature is in the image for w, h sampling on coordinates calculated by Equation 5.4 in the block defined by Equation 5.5.

$$(x', y') = (x \div w, y \div h) \tag{5.4}$$

$$(u, v) = (x \mod w, y \mod h) \tag{5.5}$$

This rearrangement can be efficiently used in the pre-processing as an intermediate step for the feature response pre-calculation described in Section 5.2.4.

Local Re-arrangement

A locally rearranged image is suitable for fast loading of image feature data by SIMD instructions. It has the same structure as a block-rearranged image, but in addition it rearranges small blocks of 2×2 pixels to be stored in a single 32 bit word, as shown in Fig. 5.6.

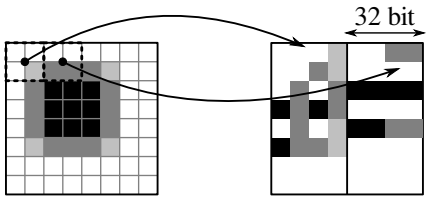


Figure 5.6: On the left, a source block rearranged image and the local rearrangement of the image.

In this memory layout, loading continuous 64 bit-aligned 64 bits from two consecutive rows, 4×4 convolution results can be obtained (i.e. two 64 bit data

accesses and load 16 data items). A feature data is located in a 3×3 sub-window of this block. The localization of data for the feature is very high, but more complex addressing than in the case of block rearrangement is needed, though. Given a feature with parameters x, y, w, h, where x, y is the *absolute* feature position in an image. The convolution image in which the data for the feature is located is defined by w, h. The feature is located in image block u, v (Equation 5.8) in a 4×4 area with coordinates x'', y'' (Equation 5.7). And the shift of the feature within the 4×4 area is defined by m, n (Equation 5.9).

$$(x', y') = (x \div w, y \div h) \tag{5.6}$$

$$(x'', y'') = 2\,(x' \div 2, y' \div 2) \tag{5.7}$$

$$(u, v) = (x \mod w, y \mod h) \tag{5.8}$$

$$(m, n) = (x' - x'', y' - y'') \tag{5.9}$$

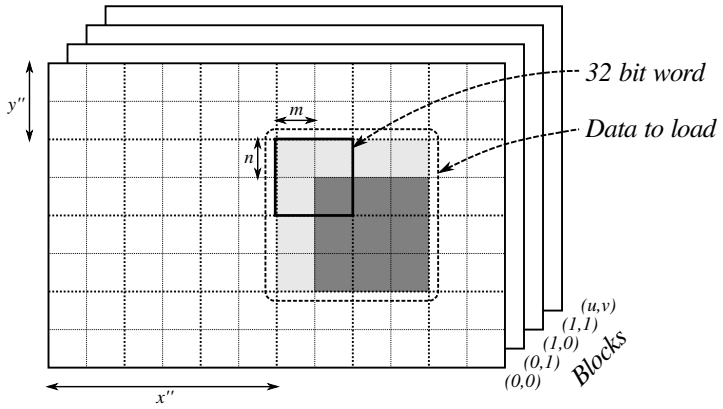


Figure 5.7: Addressing in a locally rearranged image. The image, divided into block (u, v), is stored in memory so that each 2×2 pixels is located in a word. From the feature position x, y, its parameters for addressing can be calculated.

The advantage of such a rearrangement is that data for every feature can be loaded efficiently by only two 64 bit data accesses. Even though this pre-processing seems to be rather complex, it can be implemented very efficiently and the complex addressing can be to great extent pre-calculated and implemented using look-up tables.

5.2.4 Feature Pre-calculation

Not only can the sampling function be pre-calculated, but the feature response can be pre-calculated too. The motivation is that during detection, the feature response on different positions have to be obtained. This action may even be repeated millions

of times, and some features might be accessed more than one time. Pre-calculating the response can thus speed-up some cases.

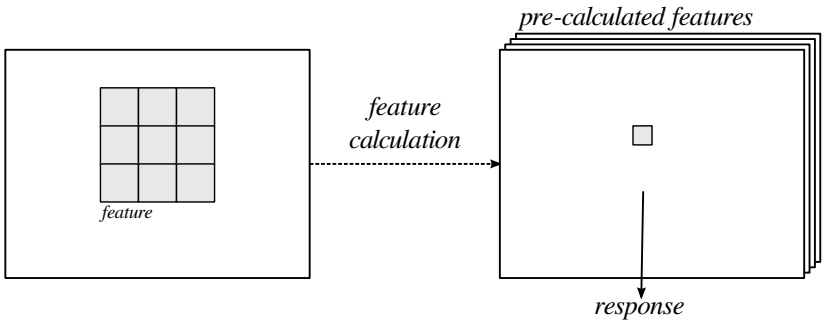


Figure 5.8: Feature evaluation using a pre-calculated response.

What is needed is a feature response for every position in an image which considers all feature parameters. This means that not every feature types are suitable for this kind of pre-processing. In the case of the LBP feature, only images for different feature sizes have to be calculated (e.g. 4 images when using sample sizes up to 2×2 pixels). But in the case of the LRD feature with same sample sizes as the LBP, the number is 144 because not only responses for different feature sizes have to be calculated, but different combination of ranks must be considered too.

The efficiency of this method is dependent on the number of accesses to each feature. When a feature is needed more than one time during the whole detection, it may be reasonable to pre-calculate it. On the other hand, when it is needed only once, or even not at all, pre-processing is inefficient. This is the reason why the LBP pre-calculation is reasonable. On each position of an image, there are four pre-calculated values, and there is large probability that all of them are needed during the detection. Moreover, each feature is likely to be needed more than once. The LRD pre-calculation, on the other hand, has on each position 144 pre-calculated values. It is also very unlikely that all of them are needed during the detection, and most of the computations in the pre-processing are thus unnecessary. In the case of non-parametrized features (except for position and size) like LBP, this method is efficient – obtaining a response of a feature is reduced to only one memory reference.

$$(x', y') = (x \div w, y \div h) \tag{5.10}$$

$$(u, v) = (x \mod w, y \mod h) \tag{5.11}$$

As a source for feature pre-calculation any type of image can be used. It can be calculated directly on an intensity image or from an integral image. The most convenient way, however, is to use block-rearranged convolution images as the features

are represented *always* as 3×3 pixel blocks and the responses can be calculated by a simple SIMD code (see Section 5.3.3). The addressing, similar to the convolved images is, however, a little more complex. Let us consider a feature with parameters x, y, w, h where the x, y is the *absolute* position in an image. The image with the feature response is defined by w, h (and possibly by other parameters). The feature response is on the coordinates x', y' in block u, v from equations (5.10) and (5.11).

5.3 Feature Evaluation using SIMD

5.3.1 Calculating a Single Response

This subsection deals with the situation when only a single feature response is evaluated. It is the standard way on how to implement the classifier evaluation on an image. Fig. 5.9 shows how the feature evaluation using SIMD works in principle. First, the samples are loaded from the image into an SIMD register and then the data are processed by a fixed chain of instructions in order to get the response.

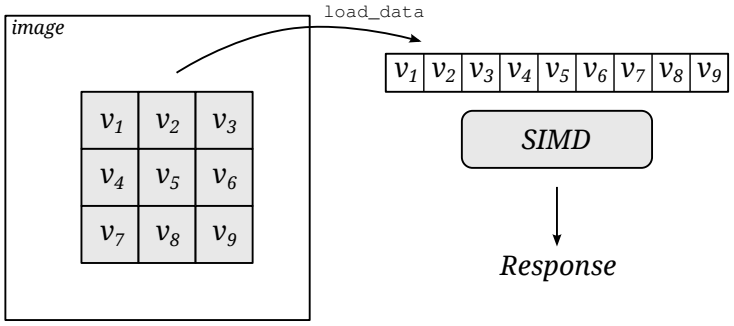


Figure 5.9: All data for feature evaluation can be loaded as an SIMD vector and processed in a data-parallel fashion.

The pseudocode in Listing 5.2 shows the feature response calculation for LBP, LRP and LRD. Given a feature with parameters x, y, w, h we can obtain the feature data from a particular image representation (integral, intensity or convolved) by function `load_data` which deals with addressing in the particular image representation. Most convenient, in this case, are local rearranged convolution images, but any type can be used. The data type `vector` stands for the SIMD data type (e.g. `__m128i` register type in the case of Intel's SSE). The function `expand` expands a particular sample to its full vector width and `sum_vector` sums all vector items. All used operations have their equivalent in common SIMD instruction sets in modern CPUs.

```
float eval_lbp_simd(int x, int y, TStage s)
{
    vector weights = {1, 2, 4, 8, 16, 32, 64, 128};
    vector samples = load_data(img, x+s.x, y+s.y, s.w, s.h);
```

```
    vector center = expand(samples, 4);
    int lbp = sum_vector((samples > center) * weights);
    return s.alpha[lbp];
}

float eval_lrd_simd(int x, int y, TStage s)
{
    vector samples = load_data(img, x+s.x, y+s.y, s.w, s.h);
    vector A = expand(samples, s.A);
    vector B = expand(samples, s.B);
    lrd = sum_vector(samples > A) - sum_vector(samples > B);
    return s.alpha[lrd + 8];
}

float eval_lrp_simd(int x, int y, TStage s)
{
    vector samples = load_data(img, x+s.x, y+s.y, s.w, s.h);
    vector A = expand(samples, s.A);
    vector B = expand(samples, s.B);
    lrp = 10 * sum_vector(samples > A) + sum_vector(samples > B);
    return s.alpha[lrp];
}
```

Listing 5.2: LBP, LRD and LRP evaluation codes using vector processing functions.

In the evaluation of the LBP feature (`eval_lbp_simd` function), right after data loading, the first central sample is expanded to its full register width. The expanded value is then compared to all others producing intermediate results which is used as a mask for a vector with weights. Each weight corresponds to a value of a bit in the LBP code. Masked weights are summed up producing the LBP response.

The evaluation of LRD and LRP (`eval_lrd_simd` and `eval_lbp_simd` functions) are very similar to each other. They are parametrized by values A and B which are indices of items in the data from which ranks are calculated. The values `v[a]` and `v[b]` are expanded to their full register width and compared to all other values. Intermediate results can be interpreted as vectors in which items that are higher than item A (resp B) are marked (i.e. the number of such items are ranks of item A (resp. B)). Ranks are calculated by summing the intermediate results and the values are simply used to calculate the LRD or LRP value.

5.3.2 Calculating Multiple Responses

After each evaluated weak hypothesis a decision about rejection is made, when evaluating a classifier. This is due to the structure of WaldBoost classifiers which is sequential in nature. While this is important in early stages of classification, in later stages the decision has to be made less often (as shown in experiments) and the price

for evaluating features separately could be very high. Moreover "bunch" evaluation of weak classifiers is beneficial in AdaBoost classifiers where *all* weak classifiers have to be evaluated.

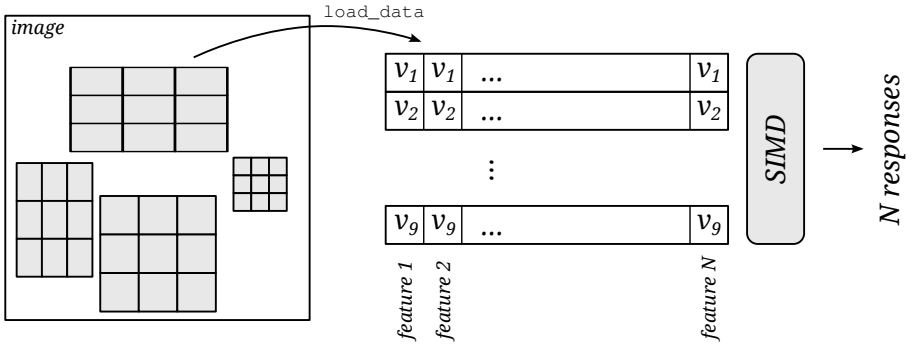


Figure 5.10: Data for multiple features can be loaded as SIMD vectors and all of them can be evaluated simultaneously.

```
void eval_lbp_simd_bunch(int x, int y, TStage s[16], float * h)
{
    const int order[8] = {0, 1, 2, 5, 8, 7, 6, 3};
    vector v[9] = load_data(img, s);
    vector w = {1};
    vector response = {0};

    for (int i = 0; i < 8; ++i)
    {
        if (v[order[i]] > v[4])
            response |= w;
        w <<= 1;
    }

    for (int i = 0; i < 16; ++i)
        h[i] = s[i].alpha[response[i]];
}
```

Listing 5.3: The code for "bunch" evaluation of 16 LBP features (128 bit (16×8 bits) where SIMD is assumed). The `load_data` function in this case loads data for all features to 9 variables which are then processed by a data-parallel code. Sixteen responses of weak hypotheses are returned as a result (`h`).

The evaluation of responses of a bunch of features can be efficiently rewritten to use SIMD instructions to evaluate *all* features by a short sequence of instructions. The Listing 5.3 shows the algorithm which calculates a bunch of 16 LBP features. Codes for LRP and LRD could be made analogically. It should be noted that the code is very similar to the sequential one (see Listing 5.1), but in this case, one sample is replaced by a bunch of 16 samples.

5.3.3 Pre-processing the Image

The above methods extracted a feature response during the classifier evaluation. Then the feature response is pre-calculated (i.e. when the extraction is done during the pre-processing stage), during the detection only one memory reference has to be made to get the response.

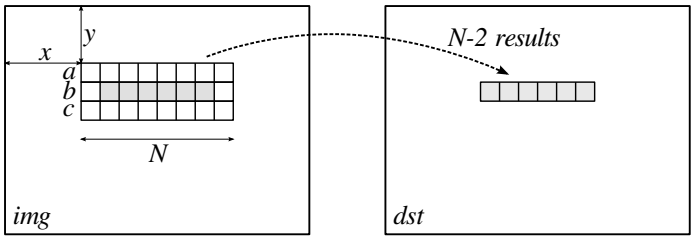


Figure 5.11: In SIMD, multiple spatially close feature responses can be calculated by a simple code.

In principle, illustrated in Fig. 5.11, when using N-wide SIMD, N-2 responses of spatially close features can be calculated by a simple code (see Listing 5.4). The code on the given position x, y in the image loads N pixels from 3 consecutive rows. The responses are then produced by using differently shifted versions of the data, producing N results. Only $N - 2$ of the results are, however, valid.

```
void eval_lbp_simd_bunch(Image img, Image dst, int x, int y)
{
    vector a, b, c;
    vector result = zeros;
    vector w = {1};

    a = load_data(img, x, y);
    b = load_data(img, x, y+1);
    c = load_data(img, x, y+2);

    result |= (w & ((a >> 8) > b));  w <<= 1;
    result |= (w & ((a) > b));       w <<= 1;
    result |= (w & ((a << 8) > b));  w <<= 1;
    result |= (w & ((b << 8) > b));  w <<= 1;
    result |= (w & ((c << 8) > b));  w <<= 1;
    result |= (w & ((c) > b));       w <<= 1;
    result |= (w & ((c >> 8) > b));  w <<= 1;
    result |= (w & ((b >> 8) > b));

    // write masked result to the dst
 }

void preprocess_image_lbp(Image src, Image dst)
{
```

```
    for (int y = 0; y < src.height-2; ++y)
        for (int x = 0; x < src.width-16; x += 14)
            eval_lbp_simd_bunch(Image img, Image dst, int x, int y);
}
```

Listing 5.4: Pre-Calculation of a LBP feature responses on an SIMD.

The function `pre-process_image_lbp` in Listing 5.4 processes the whole image. It should by noted that when calling `eval_lbp_simd_bunch` for some y and $y+1$, two lines of data are the same in both cases and it is not necessary to load them. The code can be simply modified to address this issue – process N pixels wide vertical strip of image.

The evaluation of a feature during the classifier response calculation is then only a matter of *one* memory access (see Section 5.2.4).

5.4 Implementation with Intel SSE

The principles described in Section 5.2 and 5.3 were used for implementation of a software library for object detection. The library uses Intel's Streaming SIMD Extension (SSE) instruction set. The SSE is supported by modern compilers (such as GNU Compiler Collection or Microsoft Visual Studio) in the assembly language and it is also supported as a library of intrinsic functions (i.e. a function call is translated during compilation to a single instruction). The library is published as free software[1] and it can be used in other applications that use object detection with classifiers.

As classifiers, the library supports XML files produced by the experimental framework for research on detection classifiers [38] developed at Brno University of Technology. The framework is also available on-line[2] for public use [47] where a user can upload data and learn his own classifiers. The classifiers, however, do not need to be stored in XML and the library supports classifiers in the form of a static structure in C source code. Such a representation is suitable for embedded devices or applications with built-in detectors.

5.4.1 Pre-processing

The library implements pre-processing methods described in Section 5.2. The input is an intensity image which is transformed to a pre-processed image (see Fig 5.12) which is an abstract image that can hold any form of image (e.g. a convolution with block rearrangement).

It should be noted that the pre-processing is *not* accelerated with the exception of pre-calculation of the LBP operator. Implementing the functions more efficiently

[1] <http://medusa.fit.vutbr.cz/libabr>
[2] <http://medusa.fit.vutbr.cz/detect>

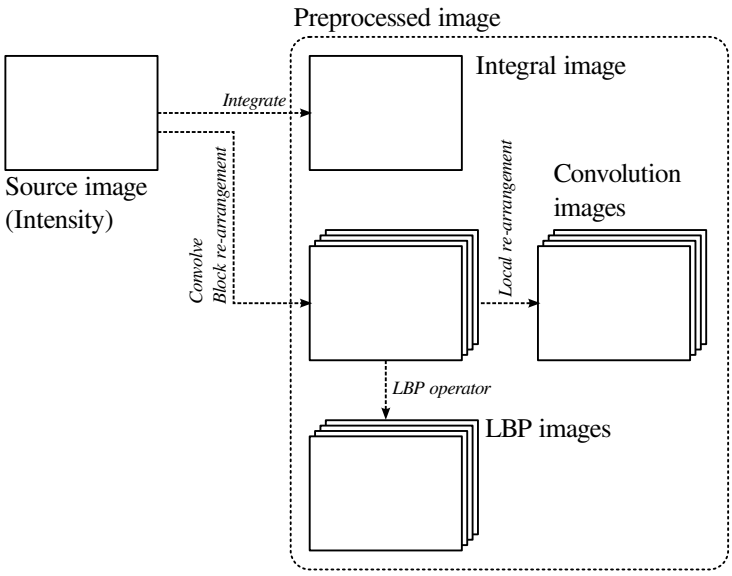


Figure 5.12: Scheme of pre-processing used in the library. The programmer can select which representations have to be calculated during the pre-processing.

would result in an improved performance. For the purposes of this thesis, however, the speed of the pre-processing is of little importance.

5.4.2 Object Detection

The pre-processed image is an input of the object detection. The other input is a structure with classifier parameters. The library implements five main methods (see Table 5.2) of detection which differs by the image structure on which they are evaluated.

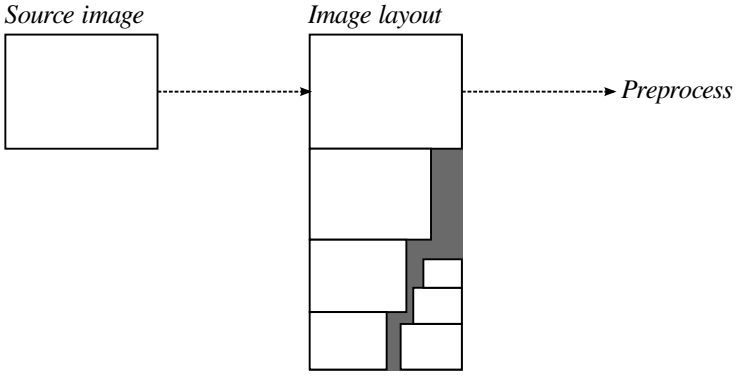


Figure 5.13: Scaled versions of the input image are placed on a larger image which is pre-processed. Object detection is then executed on this image in order to detect multi-scale objects.

Every method scans the input image *in a single scale* (i.e. it returns positions

of objects which has the size of the classification window). Multi-scale detection is achieved by putting scaled versions of the input on the larger image – image layout (see Fig. 5.13) similar to [32]. The detection of differently sized objects can then be performed in a single scale. The results after detection have to be transformed back to a normal image space. The main advantage of image layouting is that only one image is used and the image data are more *compact* in the memory. The drawback is that the image contains unused areas which are processed uselessly. However, the fraction of such areas is very low.

Run-time	Input	Bunch eval.	Ops. per feature	Mem. accesses per feature
Intensity	Intensity image	no	33 — 60	9 — 36
Integral	Integral image	no	40	16
SSE-A	Convolved image	no	7	2
SSE-B	Convolved image	yes (16)	11	9
SSE-C	Pre-calculated LBP	no	1	1

Table 5.2: Summary of properties of different run-times implemented in the library.

Table 5.2 summarizes properties of the implementation in the library. The column *input* describes what pre-processing is needed for the run-time. The column *bunch eval.* says whether the run-time evaluates weak classifiers sequentially or in a bunch. The *Ops. per feature* is the number of operations related to the evaluation of the feature (the response evaluation without addressing, data access and other overhead), and *Mem. accesses per feature* is the number of accesses to memory related to the loading of feature data (without addressing and other calculations). From the values, one can get a notion of complexity of the evaluation of features. The values, however, do not include addressing of data and pre-processing. The effeciency of the feature extraction is the subject of measurements in Section 7.1.

5.4.3 Benchmark

A widely used software implementation of the object detection is the OpenCV Haar Cascade which can be considered as a baseline. The main properties of this implementation are the following: the usage of AdaBoost Cascade classifier with Haar features and LBP features, support of multiple cores via the TBB interface (Intel Thread Building Blocks), and canny pruning which allows for skipping image areas with a low probability of target object occurrence.

The OpenCV Cascade implementation was compared to the SSE-A implementation described in Section 5. Detectors supplied with the OpenCV package were used. A competing classifier contains 1000 LBP based weak classifiers learned by WaldBoost with a *false negative rate* set to $\alpha = 0.2$. All classifiers were tested on the CMU dataset (130 images) in order to obtain ROC points. OpenCV, however, does not offer a setting of the detection threshold and only one point for a classifier can

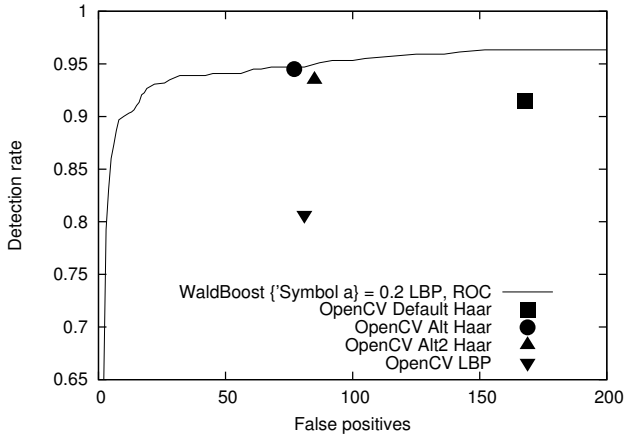


Figure 5.14: Comparison of classifiers supplied with the OpenCV 2.3 to the WaldBoost classifier. The Haar Cascade Alt has a similar error rate to the WaldBoost. The LBP classifier using the same features, however, has a larger error rate and a false positive rate.

be obtained. Comparison of the classifiers is shown in Fig. 5.14. The closest match to the WaldBoost classifier in terms of error rate is Haar Cascade Alt.

The benchmark was executed on PC (Intel Core i5 with 4 cores, 3.3GHz. 4GB RAM, Debian Linux 32bit) and it tested the multi-scale detection performance on the CMU dataset. Fig. 5.15 shows the detection time dependency on the input image resolution for all tested classifiers. The time includes image pre-processing and multiscale detection of objects in *all* image positions. The SSE-A implementation is approximately 1.2 times faster than the OpenCV Haar Cascade when using a classifier with similar error rates (Haar Alt Cascade). Compared to the OpenCV LBP Cascade, the WaldBoost runtime is slower by approximately 2 times. Such a comparison, however, is unfair as the OpenCV classifier is less accurate, which makes it inherently faster.

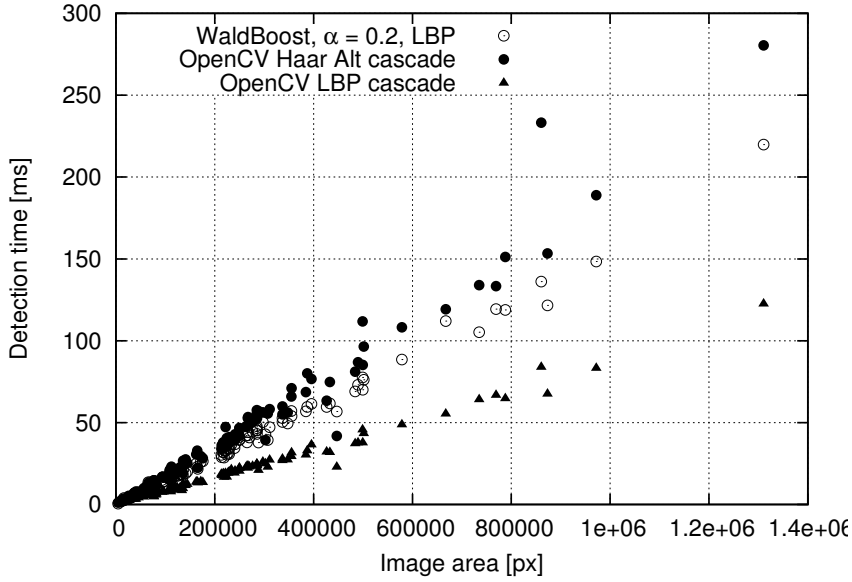


Figure 5.15: The plot shows the dependence of detection time on the resolution of the image (image area in pixels).

CHAPTER 6

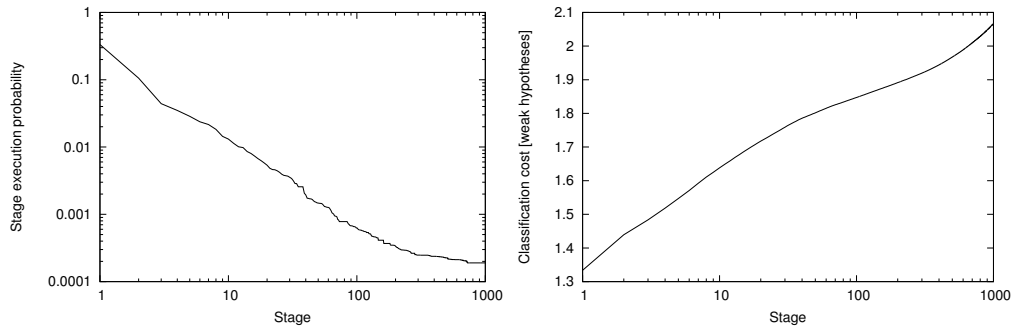
Classification Cost and its Minimization

This chapter presents the contribution of the thesis – *minimization of classification cost of WaldBoost classifiers through a combination of different run-time implementations of object detection*. Sources of the classification cost are twofold. Firstly, the classifier has its inherent cost given by the learning process. When executing the classification on a set of images, there is an average number of weak classifiers that has to be evaluated in order to reach a decision. Classifiers executing a lower number of weak classifiers are faster and therefore have a lower cost. Secondly, the cost is a property of the classification engine in which the object detection is executed. The engine can be implemented by various methods – parallel evaluation of weak classifiers, parallel evaluation of image sub-windows, etc.; and on various platforms offering different means of classifier evaluation – SIMD, FPGA, etc. The knowledge of the classifier properties and properties of the detection engines can be used for reduction of computational effort. The reduction can be performed by combining two or more detection engines, each executing a different part of the classifier [48] (e.g. a hardware pre-processing unit connected to a post-processing unit on traditional CPU). The reduction can be applied to various types of cost (computations, memory, hardware price, etc.) as its formulation is general. In this thesis, the interest is in the minimization of computational effort and the relative cost thus roughly corresponds to the computational time (except where otherwise noted).

6.1 Classifier Properties

The main property of WaldBoost classifiers is the probability of evaluating a weak classifier, reflecting on how often a weak classifier is executed during detection. This

value p can be calculated for every stage i from statistics obtained on a dataset of images. Due to rejection nature of WaldBoost classifiers, the sequence of p_i is decreasing. The first stage is evaluated always (i.e. the $p_0 = 1$). An example of such statistics is shown in Fig. 6.1 (left). The evaluation probability captures *intrinsic* computational complexity of the classifier.



(a) (b)

Figure 6.1: Example of classifier statistics. On the left, stage execution probability. On the right, the number of evaluated weak classifiers on average for a particular length of the classifier.

The classifier statistics depends mainly on the classifier rejection rate – on how rapidly are negaive samples rejected by early weak classifiers. A classifier can have different statistics when using different implementations of the evaluation. For example, consider an implementation which evaluates four weak classifiers in one step and applying the thresholds after this "bunch" evaluation. The first four weak classifiers would have probability of evaluation equal to 1, even though execution of all of them is not necessary in most cases. The next four would have a probability of execution equal to each other, and so on. Therefore, the classifier statistics is a property of the classifier *and* the implementation of its evaluation.

6.2 Cost Evaluation

In the case of the AdaBoost and WaldBoost classifiers the total cost C is proportional to the sum of individual costs of executed weak classifiers which can be calculated by (6.1). The T is the length of classifier, k is the overall classifier cost which symbolizes evaluation cost on a particular platform on which the classification is implemented. The p is the probability of execution of particular weak classifier (see Section 6.1). The c is relative cost of the weak classifier evaluation which addresses the possibility that the weak classifiers have different costs (due to the use of different features, for example).

$$C = k \sum_{i=1}^{T} p_i c_i \tag{6.1}$$

When analyzing real classifiers, p can be obtained from the statistics on input images and can be obtained c by time measurement or other cost estimation and k can be set to a constant value ($k = 1$). In Fig 6.1, the plot on the left shows a value of p_i and the plot on the right shows the area under the p_i curve which is proportional to the amount of computational resources needed for the evaluation of the classifier.

In object detection, homogeneous classifiers are most common (i.e. those with all weak classifiers of the same type and with the same type of features). In such cases, the cost of a weak classifier is constant $c_i = c$. Additionally in AdaBoost, all weak classifiers are executed every time and the probability of executing all weak classifiers is equal to $p_i = 1$. The C from (6.1) can be thus simplified to $C^{(AB)}$ (for AdaBoost) and $C^{(WB)}$ (for WaldBoost) in (6.2).

$$C^{(AB)} = knc \qquad C^{(WB)} = kc \sum_{i=1}^{n} p_i \tag{6.2}$$

Considering a classifier of length T, the value C gives us an expected cost of evaluation of the classifier. The measure is abstract and it can express different facts about the analyzed classifier. For example, when the c is set to 1, the cost expresses the number of weak classifiers executed in average; or when set according to time needed to evaluate a particular classifier, the C expresses the average time needed to evaluate the classifier.

6.3 Cost Minimization

Besides the properties of a classifier, the properties of run-time implementation also contributes to the total cost. Implementations with different properties exist – differences can be in the design of the feature extraction, image scan, multi-scale detection, etc. Imagine, for example, an implementation A which can very efficiently evaluate $K > 1$ weak classifiers in a row, but it always evaluates *all* of them no matter how many weak classifiers are actually needed for the evaluation. It could be the pre-processing unit implemented in hardware which rejects areas without the occurrence of target object. The other implementation B in software can evaluate the classifier in a standard way. The computational cost for one feature in A is much lower than in B, but implementing the whole classifier in the hardware is hard to achieve due to limited resources. Moreover it could be uneconomic to do so as the hardware resources are relatively expensive.

$$C = \arg\min_{0 \le u \le T} \left(k_1 \sum_{i=0}^{u-1} p_{1,i} c_{1,i} + k_2 \sum_{i=u}^{T-1} p_{2,i} c_{2,i} \right) \tag{6.3}$$

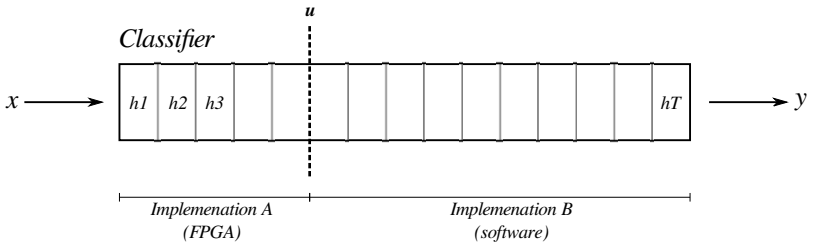


Figure 6.2: Composition of two implementations of classification. First u classifires are evaluated in FPGA and the rest in software.

Both implementations can be put together in a composed implementation as illustrated in Fig. 6.2. The problem here is on how many weak classifiers should be put into a hardware unit and on how many are left for the software. The cost of both parts can be measured and sum of the individual costs of both parts gives us a total cost C. The composition with the minimal total cost can be found using Equation (6.3).

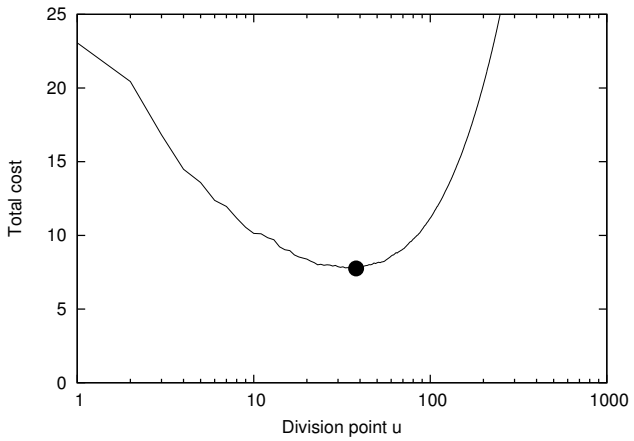


Figure 6.3: An example of minimization of total cost for a two-phase classifier. The horizontal axis corresponds to the division point of the classifier. The vertical axis is the cost of the composition. In this case, the first phase always evaluates all weak classifiers and the second phase evaluates weak classifiers one by one. The black dot marks the division with a minimal cost.

The composition of two phases can be fine tuned by one parameter – division point u. Equation 6.3 shows the minimization problem. The C is the total minimal cost of the evaluation; u is point of classifier division; and k, c and p correspond to the parameters of the cost computation from Equation 6.1. Fig. 6.3 shows values of C for different settings of u. It should be noted that although the properties p of

classifier are the same for both parts, the p can be in general different for each part. This is due to the structure of the evaluation in the particular implementation which can *force* different probabilities of feature evaluation for example by evaluating more features in one step (see Section 6.1).

When going beyond the example given above, more than two phases of evaluation can be used. And the minimization problem is thus multi-dimensional. In general cases described by (6.4), the classifier division is defined by vector $\mathbf{u}$ whose values are searched for in order to find the best composition of parts with different properties. It should be noted that u_i can be equal to u_{i+1} and some parts (evaluation implementations) could be in fact skipped when they do not contribute to the minimal cost in the composition.

$$\begin{aligned} & C = \arg\min_{\mathbf{u}} \sum_{m=1}^{M} \left(k_m \sum_{i=u_{m-1}}^{u_m - 1} p_{m,i} c_{m,i} \right) \\ & s.t. \\ & u_0 = 0 \\ & u_M = T \\ & u_{i-1} \leq u_i, \quad 0 \leq i \leq M \end{aligned} \tag{6.4}$$

In practical applications, it is easy to get the stage execution probabilities p – it reflects classifier behavior on images. On the other hand, it could be tricky to identify values of c and k. It has to be done by a careful examination of the performance of the particular implementation of the detection on the target application (e.g. by precise measurement of time needed for executing the weak classifier).

CHAPTER 7

Experiments and Results

The objective of the experiments is to test the hypothesis about minimization of total cost using a composition of classifier run-time implementations presented in Chapter 6. Firstly, the implementations used in this thesis are described and their performance is measured. This measurement serves as a cost estimation in subsequent experiment. Secondly, the evaluation metric used in the thesis is the improvement of performance of object detection.

7.1 Classification Cost Measurements

7.1.1 Experimental Setup

Classifiers used in the experiments were face detectors trained with the WaldBoost algorithm in an experimental framework developed at the Faculty of Information Technology [38]. Twelve classifiers with different properties were used – for each feature type (LBP, LRD, LRP) classifiers with $\alpha \in \{0.02, 0.05, 0.1, 0.2\}$ were used (see Section 2.3.3). Features with maximum 2×2 pixel blocks were used to meet requirements of all run-time implementations (see Section 5).

As a baseline, a software implementation working on an integral image was selected, as it is a standard way of implementation of the detection. The other used implementations in software were implementations that use an SSE intruction either for pre-processing or evaluation of features. Properties of run-time implementations are summarized in Table 5.2.

Additionally, an FPGA pre-processing unit (FPGA) is used in the experiments. This unit can contain up to N classifiers and it always evaluates *all* weak classifiers in parallel and thus works as a simple AdaBoost unit which applies WaldBoost

thresholds after the evaluation of all weak hypotheses.

	PC1	PC2
CPU	Intel Core i5	Intel Core2
Cores	4	2
Frequency	3.3 GHz	800 MHz
Memory	4 GB	1 GB
OS	Debian 32 bit	Ubuntu 11.04 32bit

Table 7.1: Specifications of the two machines used in the experiments.

The cost of software implementations correspond to the time needed for the evaluation of a weak classifier. The time was measured on a dataset of 130 images (CMU dataset) with different resolution and content. Each image was processed ten times and the times were averaged. In total, the cost was evaluated from around 26 million classifications which is enough to get an accurate estimate. The measurement was performed on two different computers (Table 7.1) with CPU frequency set to a constant value (i.e. no automatic frequency scaling allowed) and a single-thread code was used.

The cost of the classifier in the hardware implementation was set as an area needed for the classifier in the FPGA circuit, reflecting the cost of the chip. In these experiments, the cost is set constantly to $c_i = \frac{1}{N}$ where N corresponds to a number of weak classifiers which can be efficiently stored in a typical low cost FPGA. The value depends on a feature type and the hardware used; $N = 50$ is assumed in experiments. Certainly, better cost functions could be found. For example, cost incorporating properties of a real divice – such as feature evaluation speed or or price of the device. The cost selected in this work is selected in order to illustrate the principle of cost minimization. In general, by setting the cots to a low value, we simply say that the cost of the hardware unit is not of much interest to us, and conversely, by setting the cost to a large value, we say that the cost of the hardware is very important.

For each classifier, stage execution probability (see Section 6.1) was obtained from detection results on the CMU dataset.

7.1.2 Cost Measurements

Table 7.2 shows measurements of classification cost c_i for different types of features for different implementations. The cost value reflects time needed for the evaluation of a weak classifier and thus it depends only on the feature type used. The costs for all classifiers (with the same feature type) were thus averaged in order to get more precise estimate.

On the left side, Fig. 7.1 shows stage execution probabilities p_i for the classifiers in the experiments. It should be noted that p_i for classifiers with higher false negative

	PC1			PC2		
	LBP	LRD	LRP	LBP	LRD	LRP
INTEGRAL (ref.)	0.0421	0.0466	0.0464	0.2244	0.2465	0.2403
SSE-A	0.0236	0.0269	0.0306	0.1211	0.1000	0.1020
SSE-B	0.0114	0.0129	0.0117	0.0659	0.0765	0.0704
SSE-C	0.015	–	–	0.100	–	–

Table 7.2: Costs c_i of weak hypotheses evaluation in different implementations of detection run-time on two different computers used in the experiments. Note that the SSE-C implementation can evaluate only LBP features.

rate α decrease faster. This results in a lower number of weak classifiers evaluated on average, and ultimately, to higher classification speed compared to the more accurate classifiers with a lower false negative rate. The right column of Fig. 7.1 shows the speed comparison of the classifiers – the average number of weak classifiers evaluated per window.

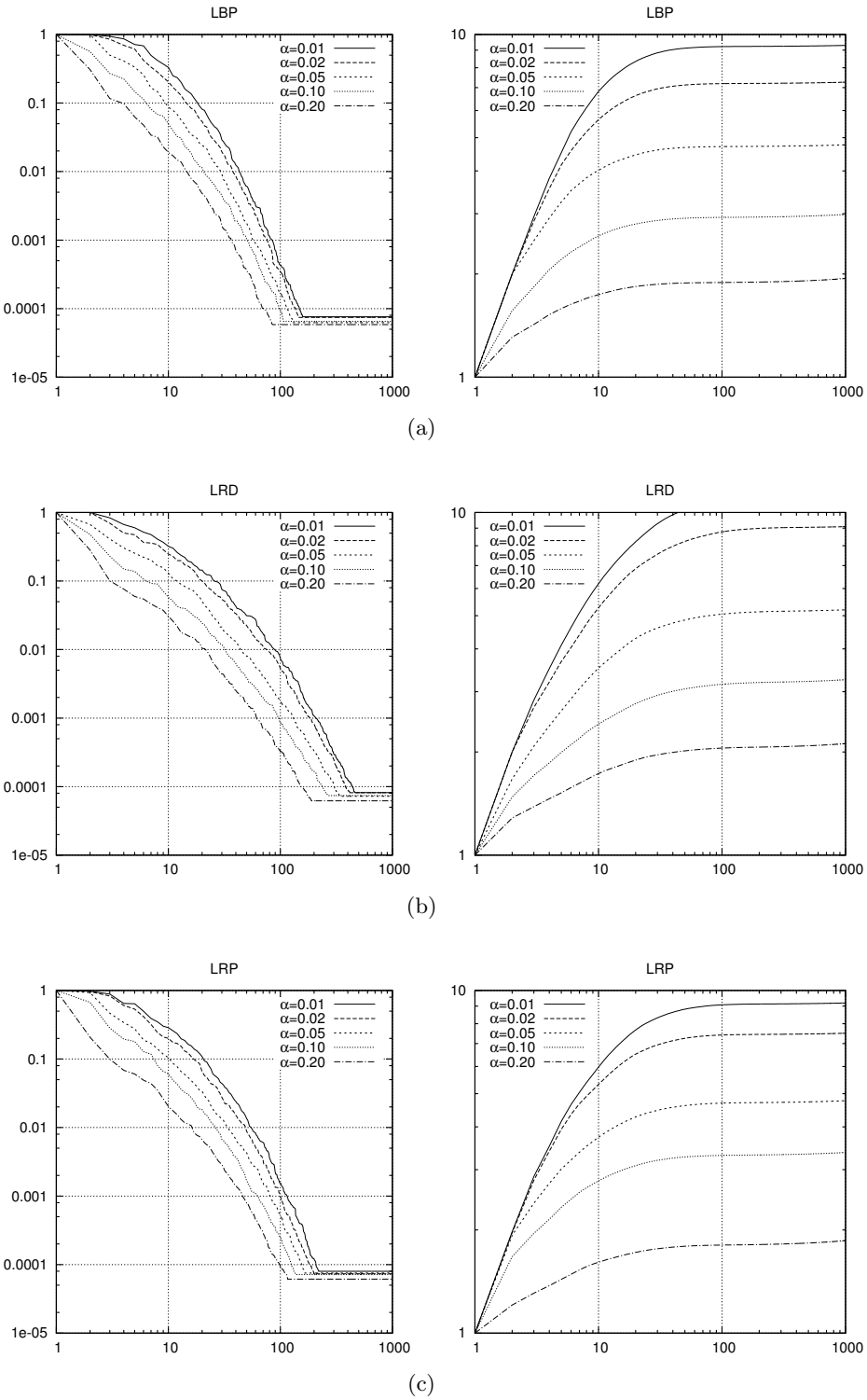


Figure 7.1: Left column, stage execution probabilities for classifiers used in the experiments. Right column shows how the classifier cost (measured in number of weak classifiers) grows with length of the classifier. Note that all axes are shown in logarithmic scale.

7.2 Cost Minimization

This section gives results of classification cost minimization based on results described in the previous section. First, compositions of classifier implementations used in the experiments are described. Then the results are presented and compared to real measurements.

7.2.1 Experimental Setup

In this experiment, classifiers were divided into two parts. The first part is evaluated in the hardware pre-processing unit. The second part is left for evaluation in the software using one of the above described run-time implementations. The classifier cost estimation method and cost minimization described in Section 6 is used to estimate the optimal length of the pre-processing unit according to the cost measure defined in Section 7.1. The optimization objective is thus minimization of the circuit area and, at the same time, minimization of the amount of computations in the software. By combination of such diverse cost measures the result (total cost C) given by the cost evaluation can be viewed as a 'relative cost' but the interpretation of the value might be somewhat problematical. This does not, however, matter too much as we do not care about the value of the cost, but instead care about the position of the minima.

7.2.2 Minimization and Measurements

Figures 7.2 to 7.4 shows results of the minimization of total cost on PC1 for different feature types used. Figures 7.5 to Fig. 7.7 shows the same for PC2. The plots show dependence of total cost on the setting of a classifier divison point. The division with minimal cost is marked by a circle. It should be noted that slower classifiers (low false negative rates, α) result in a longer part in the hardware unit, meaning that the hardware should take care of the majority of computations and the "post-processing" is left for the software. Another notable fact is that when using slower implementation in the software (or slower computer), the larger part of computations is left for the hardware unit. This means that faster computers/implementations tend to compute more weak classifiers in the software.

Comparison of the optimization and real measurement in a few selected cases is shown in Figures 7.8 to 7.11. It should be noted that although the scale of curves is different, the position of the minima is approximately in the place predicted by the optimization. The difference is mainly caused by the overhead introduced by the switching of run-time during the detection and other effects not included in the predictions (caching, memory bandwidth, etc.). The optimization thus predicts a good point of classifier division with respect to the user defined cost function.

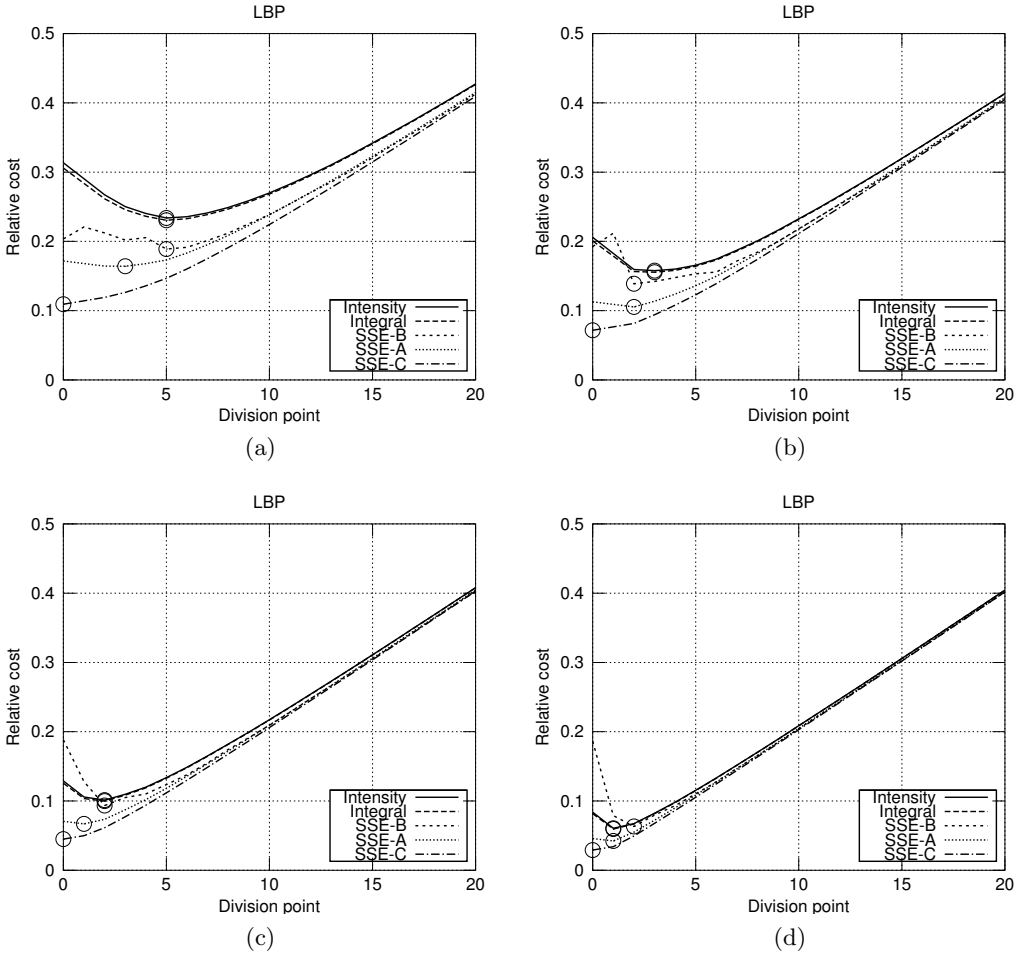


Figure 7.2: Search for an optimal division point on PC1. Classifiers were divided into two parts, the first one is executed in FPGA and the second in PC1 (different run-time implementations are shown as curves in plots). Plots (a), (b), (c) and (d) shows the cost evaluation for four different classifiers with α of 0.02, 0.05, 0.1 and 0.2 respectively. LBP features were used. The position with minimal cost is marked by a point.

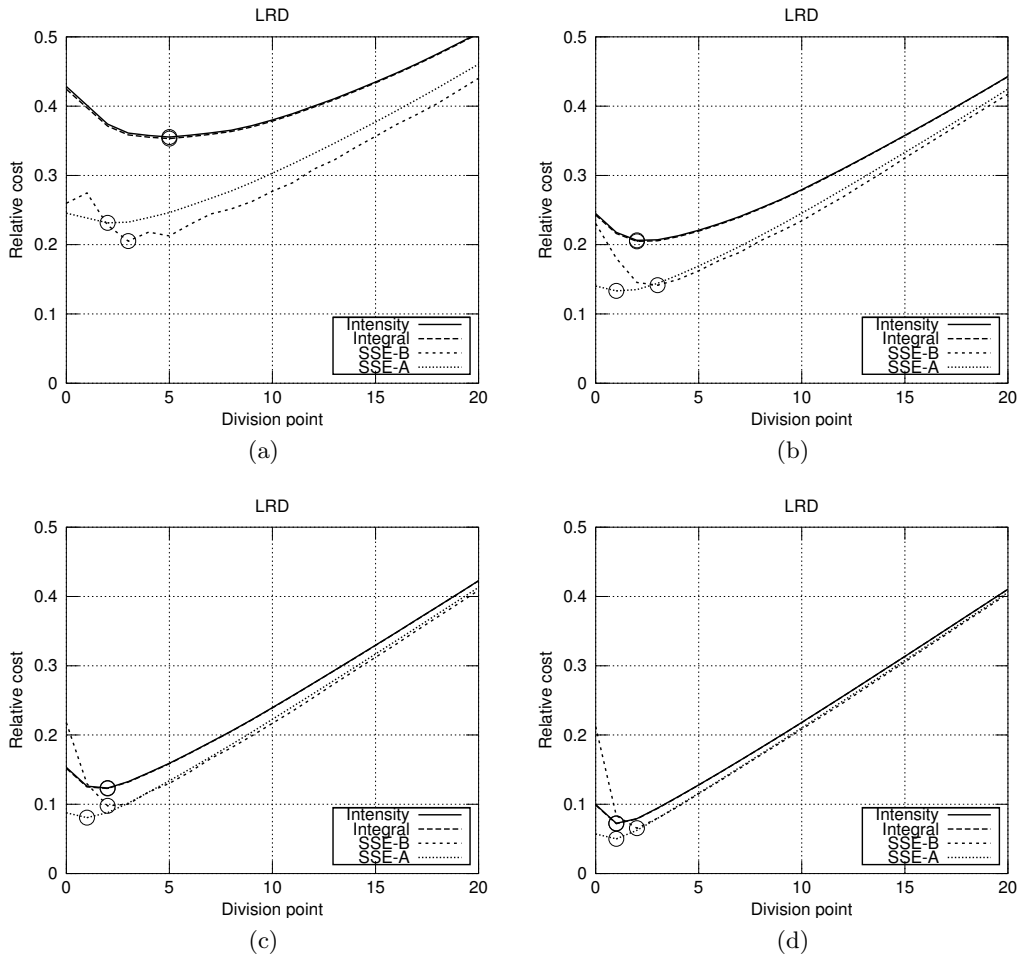


Figure 7.3: Search for optimal division point on PC1. In this case, classifiers with LRD features were used.

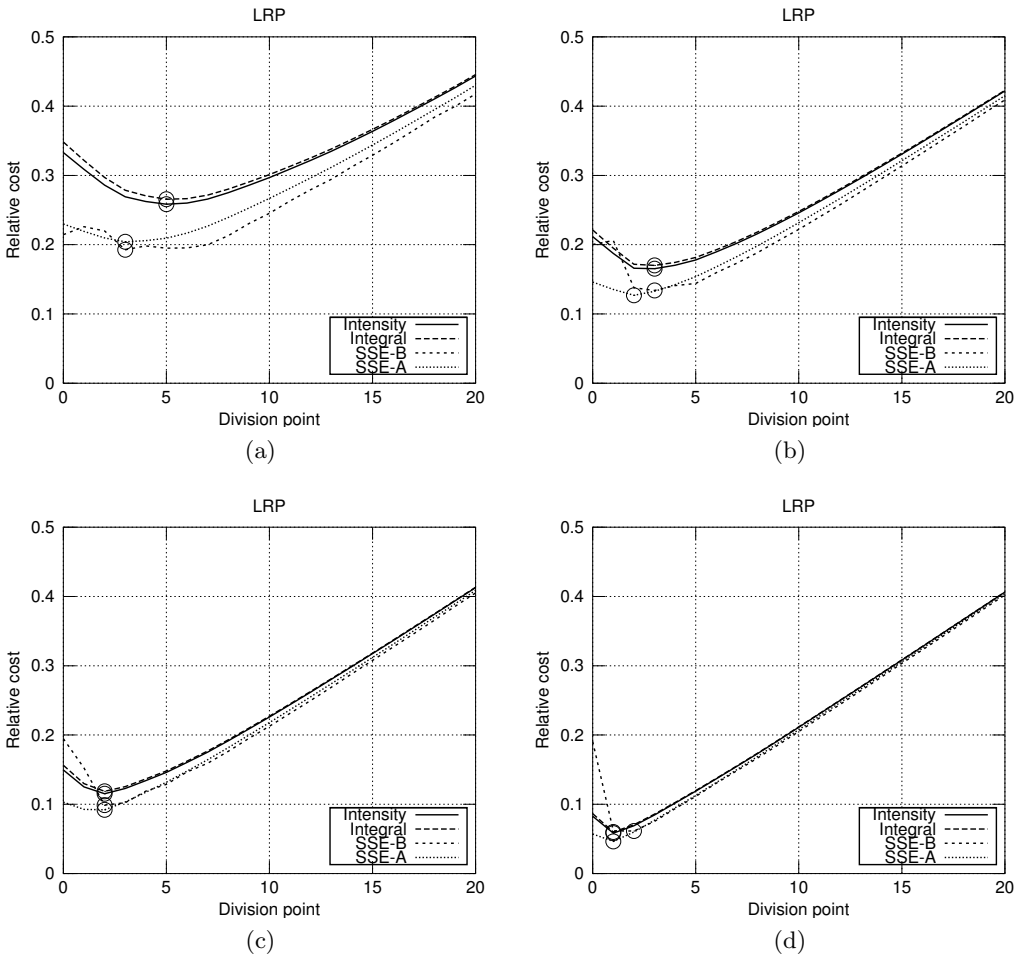


Figure 7.4: Search for optimal division point on PC1. In this case, classifiers with LRP features were used.

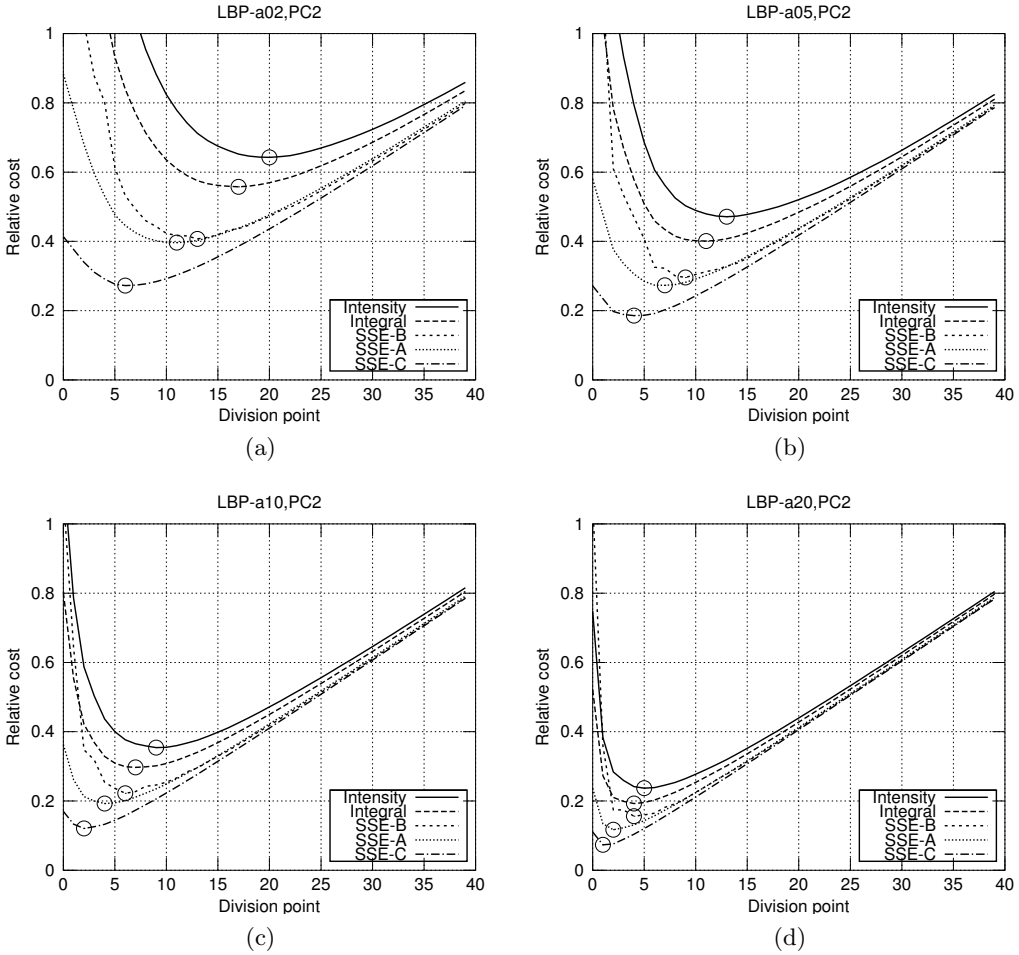


Figure 7.5: Search for an optimal division point on PC2. Classifiers were divided into two parts: the first one was executed in FPGA and the second in PC2 (different run-time implementations are shown as curves in plots). Plots (a), (b), (c) and (d) shows the cost evaluation for four different classifiers with α of 0.02, 0.05, 0.1 and 0.2 respectively. LBP features were used. The position with minimal cost is marked by a point.

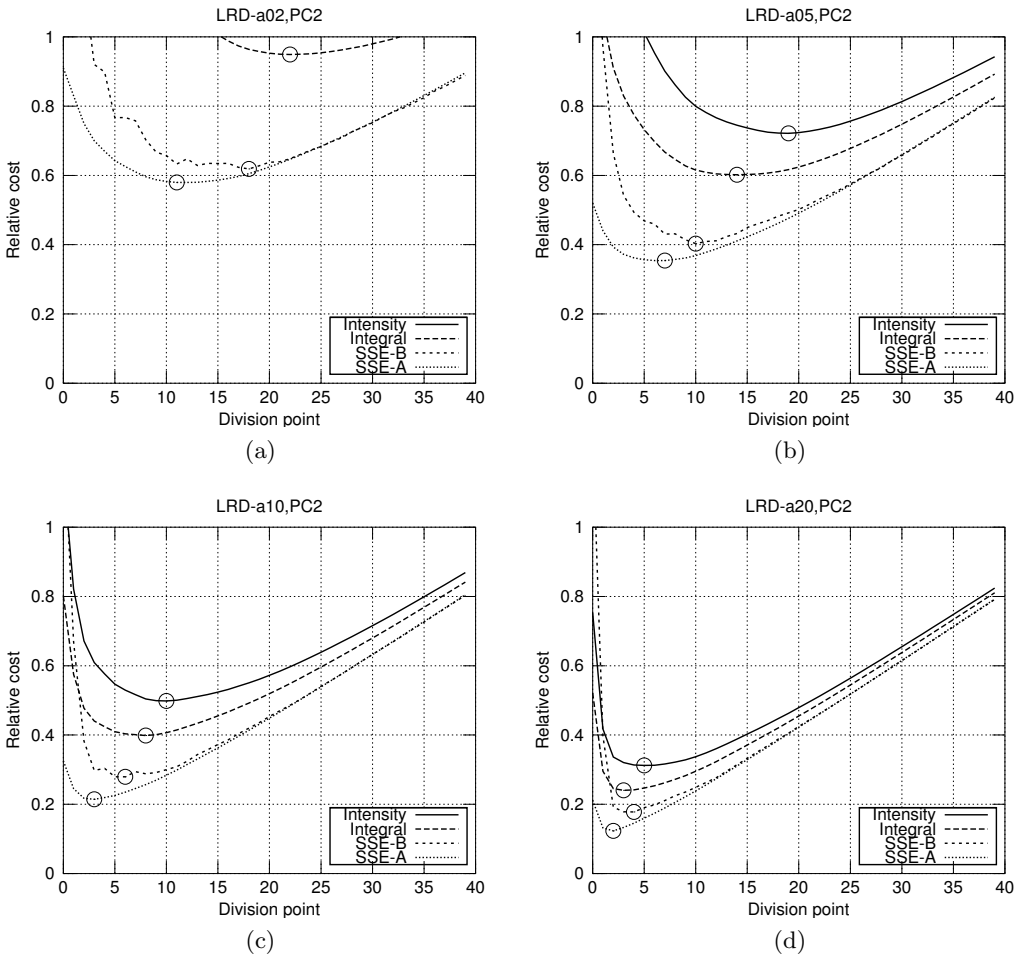


Figure 7.6: Search for optimal division point on PC2. In this case, classifiers with LRD features were used.

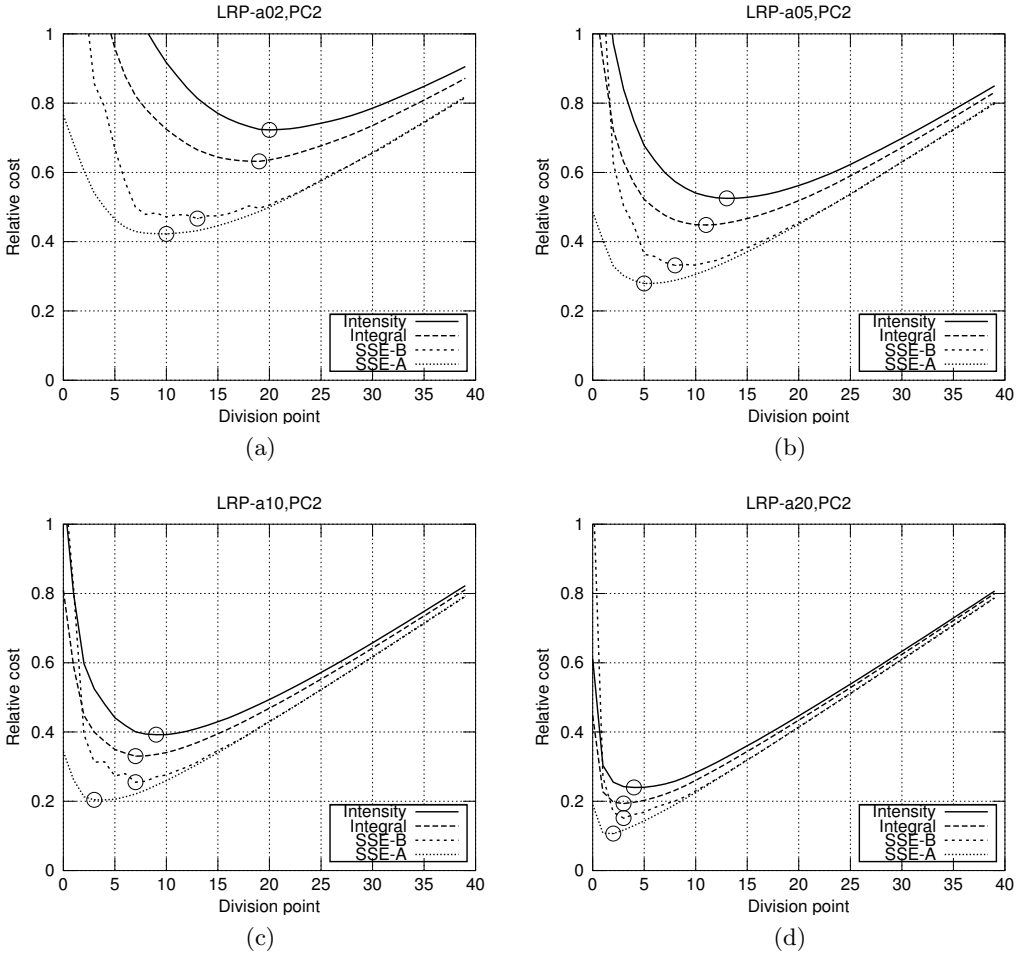
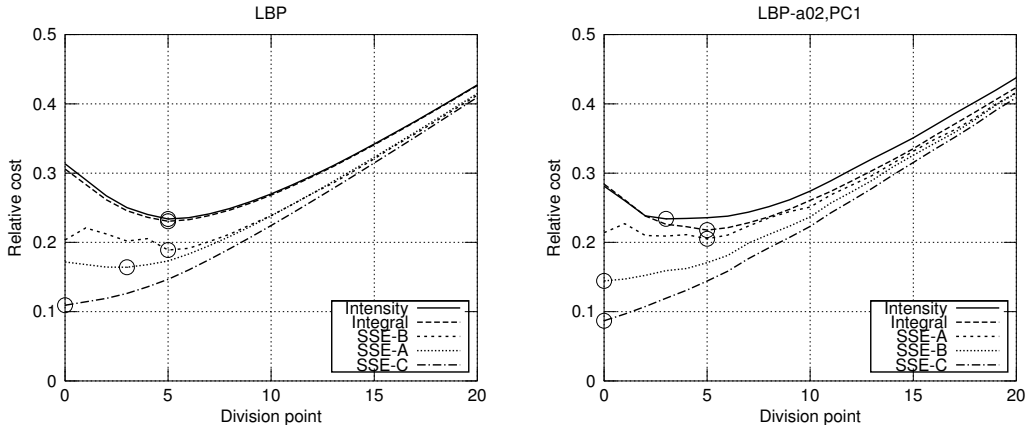
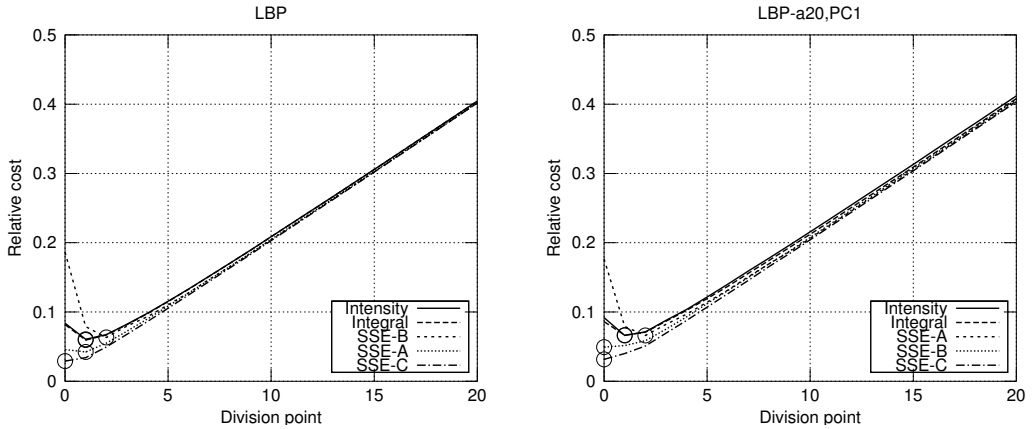


Figure 7.7: Search for optimal division point on PC2. In this case, classifiers with LRP features were used.

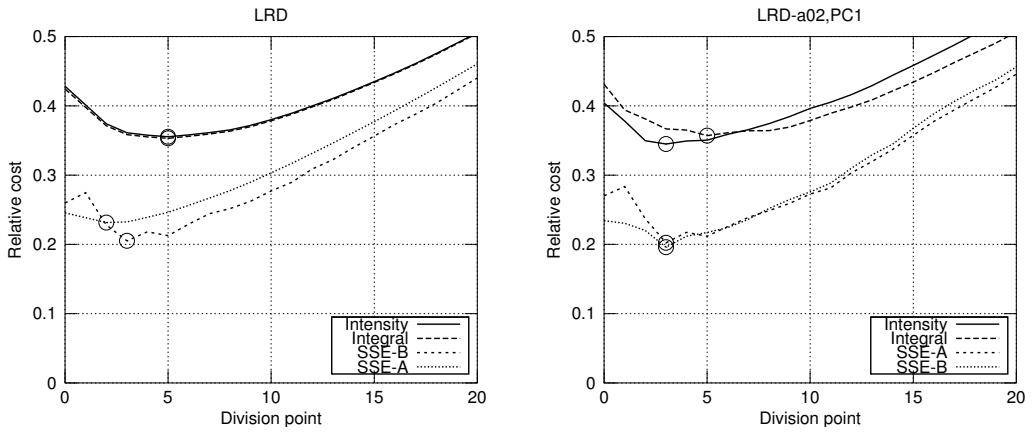


(a)

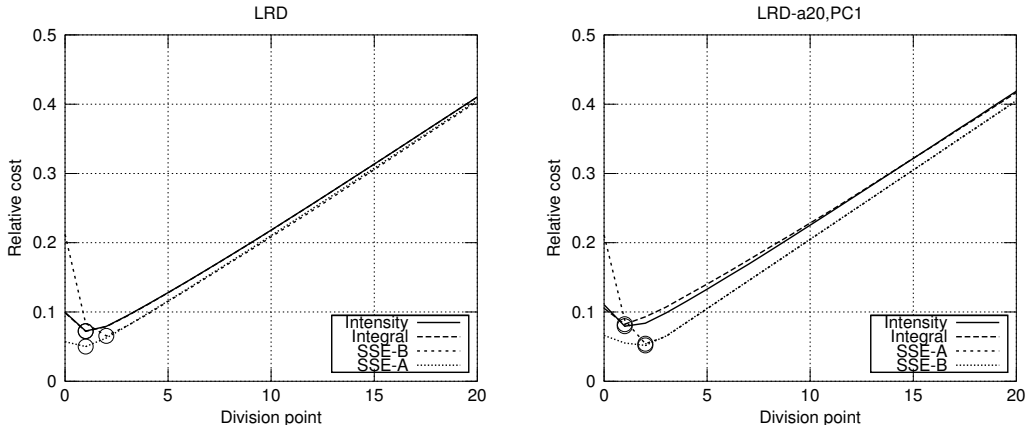


(b)

Figure 7.8: Comparison of optimization results with measurements on PC1. Each couple of plots shows (left) result of optimization and (right) result of measurement.

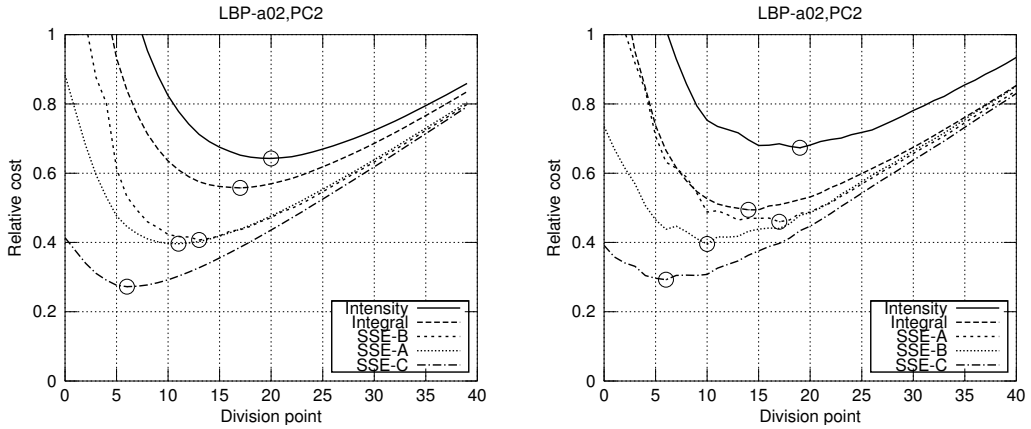


(a)

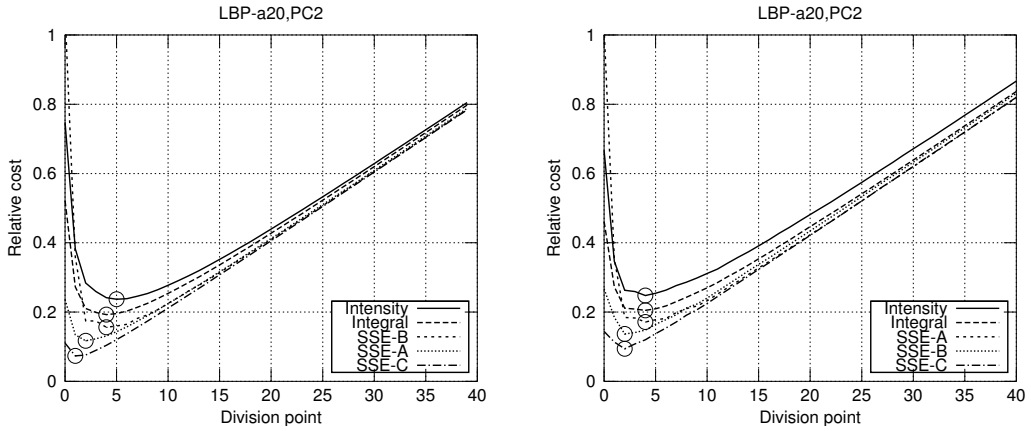


(b)

Figure 7.9: Comparison of optimization results with measurements on PC1. Each couple of plots shows (left) result of optimization and (right) result of measurement.

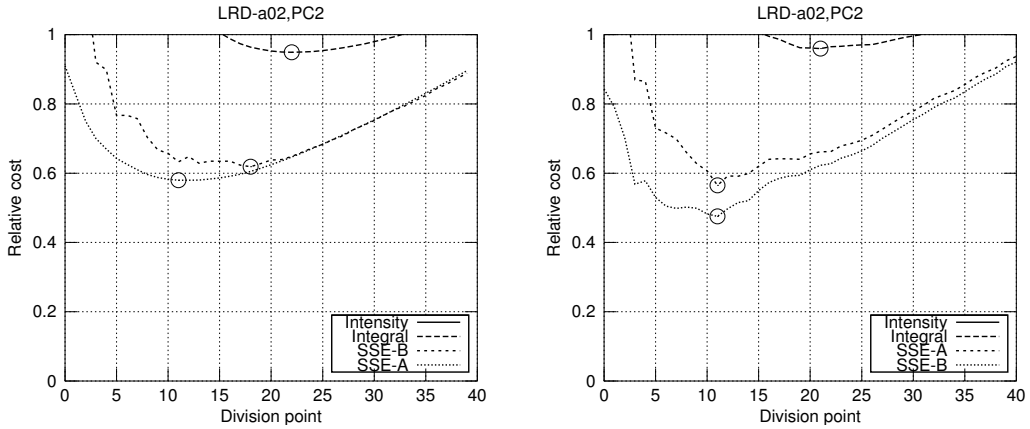


(a)

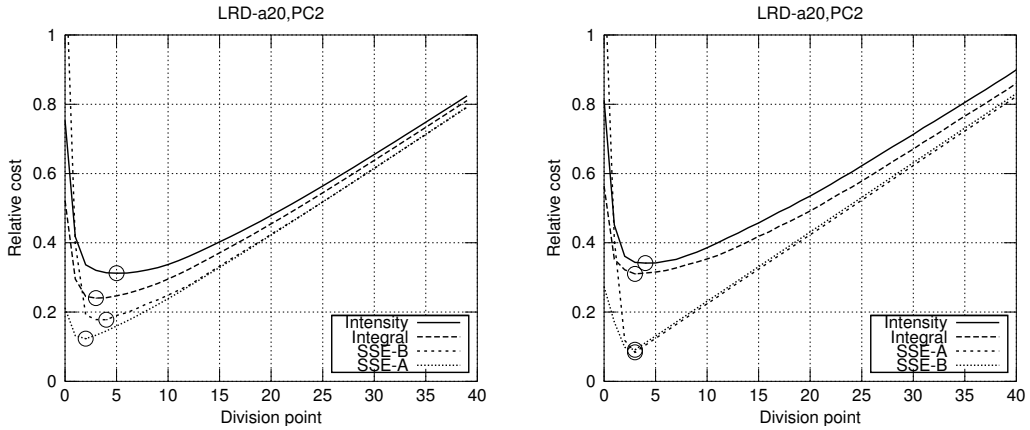


(b)

Figure 7.10: Comparison of optimization results with measurements on PC2.



(a)



(b)

Figure 7.11: Comparison of optimization results with measurements on PC2.

7.3 Results Discussion

7.3.1 Classification Costs

In this thesis five software implementations with different properties were used for the experiments.

The evaluation of weak classifiers in the SSE implementations is about two or three times faster than in the reference implementation due to the use of SSE instructions. The need of preprocessing, however, slightly lowers the computational efficiency. The SSE-A can be used in general as it evaluates classifiers in a standard one-by-one manner and has no restrictions (except for the size of the features). The SSE-B implementation comes with the fastest evaluation of weak classifiers due to the 'bunch' evaluation. The reference implementation cost is more than three times slower. The SSE-B on the whole, can be ineffectient to use for the complete detection since it evaluates weaker classifiers due to the 'bunch' manner of the evaluation. The average number of hypotheses (Fig. 7.1, left) is in most cases lower than 10 and some classifiers need only about two weak hypotheses on average to make the decision. The SSE-B thus evaluates many weak classifiers in vain. The SSE-B is beneficial in later phases of the evaluation as experiments in Section 7.2 shows. The benefit is, however, very small as the computations in the end of the classifier takes only a small fraction of the total amount of computations. The SSE-C implementation benefits from pre-calculation of the LBP operator which makes the evaluation of weak classifiers very easy. In the proprocessing, it is necessary to create a convolved image and then preprocess this image in order to calculate the LBP operator, which can be inefficient.

The implementations used in the experiments are pure WaldBoost with no additional improvements. It can be efficient to add, for example, *Neighborhood suppression* (see Section 4.4) to the evaluation. The price would be a larger overhead during the classifier evaluation, because each feature response needs to be transformed by multiple look-up tables predicting labels for neighboring sub-windows. The benefit would be large improvement of computational time as a large fraction of sub-window classifications would never happen. The Neighborhood suppression would alter the stage execution probabilities such that $p_1 < 1$, thus affecting the total cost C. The costs c_i of the weak classifiers in Table 7.2 would not be changed as the features are extracted in the same way. Using such an improvement would not, however, effectient when feature pre-calculation is used due to the fact that the pre-processing calculates all the features from an input image. The Neighborhood Suppression rejects a position even before the classification of it is executed (using prediction from other position). This would lead to many unnecessary computations during the pre-processing.

7.3.2 Cost Minimization and Measurements

The cost function can be evaluated for a classifier with the knowledge of the properties of the classifier and the costs of implementations in which the classifier is executed. The costs can be measured (when possible) or estimated in some way. The *cost* is a quantitative measure which allows us to rate different compositions of run-time implementations and select the one with the lowest cost – reduce computational effort, memory demands, power consumption or other user-defined parameters.

In the presented case, the objective was to estimate the cost of different compositions of a hypothetical hardware unit with different software implementations. The cost of the hardware unit was chosen as a linear function in order to illustrate the principle of the cost evaluation and its minimization. The costs of the software implementations were measured for a set of classifiers (with different features and false negative rates) on two different computers. The results show that the cost can be minimized by a combination of a short hardware unit and software post-processing. The length of pre-processing depends on many factors, notably properties of the particular classifier, the implementation of the detection run-time and the computer executing the detection. The results sugests that it is reasonable to use longer pre-processing in cases when a slow classifier (low false negative rate) or a slow computer is used – for such an example see Fig. 7.2a or Fig. 7.5a. And conversely, when a fast classifier or fast computer is used, the pre-processing unit can be shorter or can be even rendered useless – see Fig. 7.2d. The theoretical estimates are supported by measurements on the two computers. For this comparison, see Figures 7.8 to 7.11. In the figures, the left plot shows the theoretical evaluation of the cost, and the right plot shows the value of the measured cost. The differences between theoretical estimates and measurements are results of effects that were not included in the estimates. These are, for example, caching, operating system switching of other tasks, measurement methods, etc.

7.3.3 Application Field

The minimization of the classification cost can be applied on problems where the classification run-time can be divided into two (or more) parts with different properties. There can be pure hardware applications dividing the detection to parallel and sequential parts. The parallel part typically takes more resources, but it is very fast. The sequential part is typically smaller and slower due to the need of sequential evaluation of the weak classifiers. The classification cost minimization can predict the best lengths (number of weak classifiers) for each part in order to get the fastest solution or a solution which fits the particular hardware platform in terms of hardware resources. Alternatively, there can be software implementations combined together in order to get the fastest detection. And of course, combined applications where

the detection pre-processing is done by the fast hardware unit with the results being sent for post-processing to the software. The minimizations of the cost can balance computations between the hardware and software module.

Application of such minimization can be, for example, in surveillance or traffic control where many types of objects have to be detected. The minimization can be used for the design of a low-cost and low-power pre-processing unit. Similar applications can be found in the design of consumer digital still cameras which can detect different types of objects.

CHAPTER 8

Conclusion

Implementation of object detection is a complex task. When done without care, the process can be ineffecient, which could mean that performance is low or power consumed is too high. When high frame rates are desired, or when many types of objects need to be detected, or high resolution images have to be processed, powerful hardware and/or some acceleration techniques have to be used.

This thesis in Sections 2 and 3 reviewed detection of objects through classification and focused on WaldBoost-based classifiers with Local Rank Functions and Local Binary Patterns as image features. Section 4 summarized techiques usable for the acceleration of object detection with a focus on implementational acceleration through parallelism and algorithmic accelerations through prediction of labels of several close positions of sub-windows and early suppression of non-maximal responses. The use of data parallelism is discussed in more detail in Section 5. The thesis focused on the acceleration of detection through a combination of several implementations of object detection with known properties into a coherent unit. The contribution of the thesis is a method for evaluation of the cost of such combinations and the minimization of the cost presented in Section 6. In the experiments presented in Section 7, tested compositions of a hypothetical hardware unit with few selected implementations in the software. It turns out that it is indeed beneficial to divide the classifier evaluation into two (or even more) parts. This division allows for moving a significant portion of computations into the implementation which can efficiently reject most of background sub-windows and leave the remaining computations for a potentially slower software unit. These results are supported by measurements The optimization of the division can be tuned for the particular classifier and hardware on which the detection is executed.

The principles described in this thesis can be used to, for example, to automatically tune object detection for the best performance with respect to the properties of the classifier and the machine on which the detection is executed, by combination of more implementations of the classifier evaluation. Other example can be a design of smart cameras which can produce, along with standard image, an image pre-processed by a classifier or directly parameters of detected objects. The cost minimization allows for the composition of parallel and sequential hardware units and the cost criterion used for minimization can be, for example, power consumption, chip area, etc.

Further research includes implementation of the object detection in a hardware unit, combination of this unit with software implementation in an embedded platform and fine tuning of this combination by using the cost evaluation and minimization. The possible criteria for the minimization includes power consumption of the whole system and processing speed.

Bibliography

[1] Shivani Agarwal and Dan Roth. Learning a Sparse Representation for Object Detection. In Heyden A. et al., editor, *Computer Vision — ECCV 2002*, volume 2353 of *Lecture Notes in Computer Science*, chapter 8, pages 97–101. Springer Berlin / Heidelberg, Berlin, Heidelberg, April 2006.

[2] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face recognition with local binary patterns. In *European Conference on Computer Vision*, pages 469–481, 2004.

[3] S. Avidan. Ensemble tracking. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(2):261–271, 2007.

[4] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, 2008.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] L. Bourdev and J. Brandt. Robust object detection via soft cascade. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 236–243, 2005.

[7] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008.

[8] S. Charles Brubaker, Matthew D. Mullin, and James M. Rehg. Towards optimal training of cascaded detectors. In *In ECCV06*, pages 325–337, 2006.

[9] S. Charles Brubaker, Jianxin Wu, Jie Sun, Matthew D. Mullin, and James M. Rehg. On the design of cascades of boosted ensembles for face detection. *Int. J. Comput. Vision*, 77(1-3):65–86, 2008.

[10] Roberto Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley Publishing, 2009.

[11] Chi-ho Chan, Josef Kittler, and Kieron Messer. Multi-scale local binary pattern histograms for face recognition. *Advances in Biometrics*, 4642:809–818, 2007.

[12] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, 1. edition, 2001.

[13] Jie Chen, Shiguang Shan, Peng Yang, Shengye Yan, Xilin Chen, and Wen Gao. Novel face detection method based on gabor features. In *Sinobiometrics 2004*, Lecture Notes in Computer Science, pages 90–99. Springer Berlin / Heidelberg, November 2004.

[14] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 207–212, New York, NY, USA, 1984. ACM.

[15] Xinyi Cui, Yazhou Liu, Shiguang Shan, Xilin Chen, and Wen Gao. 3d haar-like features for pedestrian detection. In *International Conference on Multimedia Computing and Systems/International Conference on Multimedia and Expo*, pages 1263–1266, 2007.

[16] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *IEEE Computer Society Conferenceon Computer Vision and Pattern Recognition*, volume 1, pages 886–893, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Ayhan Demiriz, Kristin P. Bennett, and John S. Taylor. Linear programming boosting via column generation. *Machine Learning*, 46(1-3):225–254, 2002.

[18] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, November 2000.

[19] P. Felzenszwalb, D. Mcallester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR) Anchorage, Alaska, June 2008.*, June 2008.

[20] S. Fidler and A. Leonardis. Towards scalable representations of object categories: Learning a hierarchy of parts. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1–8, 2007.

[21] W. T. Freeman and E. H. Adelson. The design and use of steerable filters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 13(9):891–906, September 1991.

[22] Yoav Freund. Boosting a weak learning algorithm by majority. *Inf. Comput.*, 121(2):256–285, 1995.

[23] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[24] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 2000.

[25] Jiří Granát, Adam Herout, Michal Hradiš, and Pavel Zemčík. Hardware acceleration of adaboost classifier. In *Workshop on Multimodal Interaction and Related Machine Learning Algorithms (MLMI)*, pages 1–12, 2007.

[26] Adam J. Grove and Dale Schuurmans. Boosting in the limit: Maximizing the margin of learned ensembles. In *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 692–699, 1998.

[27] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69:331–371, September 1910.

[28] D. Haussler. Over view of the probably approximately correct (pac) learning framework. 1995.

[29] Jiří Havel. Accelerated object detection. In *Proceedings of the 15th Conference Student EEICT 2009*, Volume 4, pages 456–460. Faculty of Electrical Engineering and Communication BUT, 2009.

[30] Marko Heikkilä, Matti Pietikäinen, and Cordelia Schmid. Description of interest regions with local binary patterns. *Pattern Recogn.*, 42:425–436, March 2009.

[31] Adam Herout, Michal Hradiš, Roman Juránek, and Pavel Zemčík. Implementation of the „local rank differences“ image feature using simd instructions of cpu. In *Proceedings of Sixth Indian Conference on Computer Vision, Graphics and Image Processing*, page 9, 2008.

[32] Adam Herout, Radovan Jošth, Roman Juránek, Jiří Havel, Michal Hradiš, and Pavel Zemčík. Real-time object detection on cuda. *Journal of Real-Time Image Processing*, 2011(3):159–170, 2011.

[33] Adam Herout, Radovan Jošth, Pavel Zemčík, and Michal Hradiš. Gp-gpu implementation of the „local rank differences“ image feature. In *Proceedings of International Conference on Computer Vision and Graphics 2008*, Lecture Notes in Computer Science, pages 1–11. Springer Verlag, 2008.

[34] Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, Jiří Havel, Radovan Jošth, and Martin Žádník. *Low-Level Image Features for Real-Time Object Detection*, page 25. IN-TECH Education and Publishing, 2009.

[35] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Pascal Fua, and Nassir Navab. Dominant orientation templates for real-time detection of texture-less objects. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2257–2264, 2010.

[36] M. Hiromoto, K. Nakahara, H. Sugano, Y. Nakamura, and R. Miyamoto. A specialized processor suitable for adaboost-based detection with haar-like features. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2007.

[37] David W. Hosmer and Stanley Lemeshow. *Applied logistic regression (Wiley Series in probability and statistics)*. Wiley-Interscience Publication, September 2000.

[38] Michal Hradiš. Framework for research on detection classifiers. In *Proceedings of Spring Conference on Computer Graphics*, pages 171–177. Comenius University in Bratislava, 2008.

[39] Michal Hradiš, Adam Herout, and Pavel Zemčík. Local rank patterns - novel features for rapid object detection. In *Proceedings of International Conference on Computer Vision and Graphics 2008*, Lecture Notes in Computer Science, pages 1–2, 2008.

[40] C. Huang, H.Z. Ai, Y. Li, and S.H. Lao. High-performance rotation invariant multiview face detection. *Pattern Analysis and Machine Intelligence*, 29(4):671–686, April 2007.

[41] Chang Huang, Haizhou Ai, Bo Wu, and Shihong Lao. Boosting nested cascade detector for multi-view face detection. In *Proceedings of the Internation Conference on Pattern Recognition*, volume 2, pages 415–418, Washington, DC, USA, 2004. IEEE Computer Society.

[42] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *J Physiol*, 195(1):215–243, March 1968.

[43] Seunghun Jin, Dongkyun Kim, Thuy Tuong Nguyen, Bongjin Jun, Daijin Kim, and Jae Wook Jeon. An fpga-based parallel hardware architecture for real-time face detection using a face certainty map. *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 0:61–66, 2009.

[44] Michael Jones, Paul Viola, and Daniel Snow. Detecting pedestrians using patterns of motion and appearance. In *In ICCV*, pages 734–741, 2003.

[45] Michael J. Jones and Paul Viola. Face recognition using boosted local features. Technical report, MERL, 2003.

[46] Roman Juránek, Adam Herout, and Pavel Zemčík. Impelementing local binary patterns with simd instructions of cpu. In *Proceedings of Winter Seminar on Computer Graphics*, page 5. West Bohemian University, 2010.

[47] Roman Juránek, Michal Hradiš, and Pavel Zemčík. Platform for teaching detection classifiers. In *Proceedings of Digital Technologies 2010*, page 5. Zilina University Publisher, 2010.

[48] Roman Juránek, Michal Hradiš, and Pavel Zemčík. *Real-Time Systems*, chapter Real-Time Object Detection with Classifiers, page 21. InTech Education and Publishing, 2012.

[49] Z. Kalal, J. Matas, and K. Mikolajczyk. Online learning of robust object detectors during unstable tracking. *On-line Learning for Computer Vision Workshop*, 2009.

[50] Hung-Chih Lai, M. Savvides, and Tsuhan Chen. Proposed fpga hardware architecture for high frame rate face detection using feature cascade classifiers. In *First IEEE International Conference on Biometrics: Theory, Applications, and Systems*, pages 1–6, 2007.

[51] Tai Sing Lee. Image representation using 2d gabor wavelets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(10):959–971, 1996.

[52] Bastian Leibe, Aleš Leonardis, and Bernt Schiele. Robust object detection with interleaved categorization and segmentation. *Int. J. Comput. Vision*, 77(1-3):259–289, 2008.

[53] S. Li, Z. Zhang, H. Shum, and H. Zhang. Floatboost learning for classification. In *The Conference on Advances in Neural Information Processing Systems (NIPS)*, 2002.

[54] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *IEEE ICIP 2002*, pages 900–903, 2002.

[55] Ce Liu and Hueng-Yeung Shum. Kullback-leibler boosting. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 1, pages 587–594, 2003.

[56] David G. Lowe. Object recognition from local scale-invariant features. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1150, Washington, DC, USA, 1999. IEEE Computer Society.

[57] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[58] David Marr. *Vision: Computational Investigation to Human Representation.* Freeman, 1982.

[59] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision (IJCV)*, 80(1):45–57, October 2008.

[60] Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. Gray scale and rotation invariant texture classification with local binary patterns. In *ECCV '00: Proceedings of the 6th European Conference on Computer Vision-Part I*, pages 404–420, London, UK, 2000. Springer-Verlag.

[61] Constantine Papageorgiou and Tomaso Poggio. A trainable system for object detection. *Int. J. Comput. Vision*, 38:15–33, June 2000.

[62] Constantine P. Papageorgiou, Michael Oren, and Tomaso Poggio. A general framework for object detection. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 555, Washington, DC, USA, 1998. IEEE Computer Society.

[63] M. Pietikainen. Image analysis with local binary patterns. In *SCIA*, pages 115–118, 2005.

[64] Lukáš Polok, Adam Herout, Pavel Zemčík, Michal Hradiš, Roman Juránek, and Radovan Jošth. "local rank differences" image feature implemented on gpu. In *Proceedings of the 10th International Conference on Advanced Concepts for Intelligent Vision Systems*, Lecture Notes In Computer Science; Vol. 5259, pages 170–181. Springer Verlag, 2008.

[65] P. Pudil, F. J. Ferri, J. Novovicova, and J. Kittler. Floating search methods for feature selection with nonmonotonic criterion functions. In *Pattern Recognition, 1994. Vol. 2 - Conference B: Computer Vision & Image Processing., Proceedings of the 12th IAPR International. Conference on*, volume 2, pages 279–283 vol.2, 1994.

[66] Gunnar Ratsch. *Robust Boosting via Convex Optimization: Theory and Applications*. PhD thesis, Mathematisch-Naturwissenschaftichen Fakultat der Universitat Potsdam, October 2001.

[67] James Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[68] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2(11):1019–1025, November 1999.

[69] R Rigamonti, Matthew A. Brown, and V Lepetit. Are sparse representations really relevant for image classification. In *IEEE Computer Vision and Pattern Recognition (CVPR) 2011*, pages 1545–1552, June 2011.

[70] Henry A. Rowley, Shumeet Baluja, and Takeo Kanade. Neural network-based face detection. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, 20:23–38, 1998.

[71] Cynthia Rudin, Robert E. Schapire, and Ingrid Daubechies. Boosting based on a smooth margin. In *Learning Theory*, volume 3120/2004 of *Lecture Notes in Computer Science*, pages 502–517. Springer, June 2004.

[72] Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2):197–227, 1990.

[73] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee S. Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

[74] Robert E. Schapire and Yoram Singer. Improved boosting algorithms using confidence-rated predictions. *Mach. Learn.*, 37(3):297–336, 1999.

[75] Henry Schneiderman and Takeo Kanade. A statistical method for 3d object detection applied to faces and cars. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 1:1746+, 2000.

[76] Thomas Serre, Minjoon Kouh, Charles Cadieu, Ulf Knoblich, Gabriel Kreiman, and Tomaso Poggio. A theory of object recognition: Computations and circuits in the feedforward path of the ventral stream in primate visual cortex. In *AI Memo*, 2005.

[77] Thomas Serre, Aude Oliva, and Tomaso Poggio. A feedforward architecture accounts for rapid categorization. *PNAS*, 104(15):6424–6429, April 2007.

[78] Thomas Serre and Maxmilian Reisenhuber. Realistic modeling of simple and complex cell tuning in the hmax model, and implications for invariant object recognition in cortex. Technical report, MIT CSAIL, 2004.

[79] Thomas Serre, Lior Wolf, and Thomasso Poggio. A new biologically motivated framework for robust object recognition. Technical report, MIT CSAIL, 2004.

[80] Thomas Serre, Lior Wolf, and Tomaso Poggio. Object recognition with features inspired by visual cortex. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 994–1000, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Jan Sochman and Jiri Matas. Adaboost with totally corrective updates for fast face detection. In *FGR*, pages 445–450, 2004.

[82] Jan Sochman and Jiri Matas. Inter-stage feature propagation in cascade building with adaboost. In *Proceedings of the 17th International Conference on Pattern Recognition*, pages 236–239, Washington, DC, USA, 2004. IEEE Computer Society.

[83] Jan Sochman and Jiri Matas. Waldboost - learning for time constrained sequential detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 150–156, Washington, DC, USA, 2005. IEEE Computer Society.

[84] Jan Sochman and Jiri Matas. Learning fast emulators of binary decision processes. *International Journal of Computer Vision*, 83(2):149–163, June 2009.

[85] T. Theocharides, N. Vijaykrishnan, and M.J. Irwin. A parallel architecture for hardware face detection. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, March 2006.

[86] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, pages 1134–1142, 1984.

[87] Vladimir N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[88] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1:51–518 vol.1, 2001.

[89] Paul Viola and Michael Jones. Fast and robust classification using asymmetric AdaBoost and a detector cascade. In *Advances in Neural Information Processing System 14*, volume 14, pages 1311–1318, 2002.

[90] A. Wald. *Sequential Analysis.* John Wiley and Sons, Inc., 1947.

[91] Yu Wei, Xiong Bing, and C. Chareonsak. Fpga implementation of adaboost algorithm for detection of face biometrics. In *IEEE International Workshop on Biomedical Circuits and Systems*, pages 17–20, Dec. 2004.

[92] B. Wu, H. Z. Ai, and C. Huang. LUT-based AdaBoost for gender classification. In *Audio- and Video-Based Biometric Person Authentication*, pages 104–110, 2003.

[93] Rong Xiao, Long Zhu, and Hong-Jiang Zhang. Boosting chain learning for object detection. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 709, Washington, DC, USA, 2003. IEEE Computer Society.

[94] Pavel Zemcik, Michal Hradis, and Adam Herout. Local rank differences - novel features for image. In *Proceedings of SCCG 2007*, pages 1–12, 2007.

[95] Pavel Zemčík, Michal Hradiš, and Adam Herout. Exploiting neighbors for faster scanning window detection in images. In *ACIVS 2010*, LNCS 6475, page 12. Springer Verlag, 2010.

[96] Pavel Zemčík and Martin Žádník. Adaboost engine. In *Proceedings of FPL 2007*, page 5. IEEE Computer Society, 2007.

[97] Cha Zhang and Paul Viola. Multiple-instance pruning for learning efficient cascade detectors. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1681–1688. MIT Press, Cambridge, MA, 2008.

[98] Lun Zhang, Rufeng Chu, Shiming Xiang, ShengCai Liao, and Stan Z. Li. Face detection based on multi-block lbp representation. In *ICB*, pages 11–18, 2007.

[99] Guoying Zhao and M. Pietikainen. Dynamic texture recognition using local binary patterns with an application to facial expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):915–928, June 2007.

[100] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *Proceedings of the IEEE Computer Society Conferenceon Computer Vision and Pattern Recognition*, pages 1491–1498, Washington, DC, USA, 2006. IEEE Computer Society.