Master of Science Thesis

An Analysis and Efficient Implementation of Felzenszwalb's Object Detection System

Dan Zecha

Universität Augsburg
Fakultät für Angewandte Informatik
Multimedia Computing and Computer Vision Lab
Supervisor: Prof. Dr. Rainer Lienhart

Augsburg, March 2013

Revised Version (28.03.2013)

Abstract

In this thesis we introduce an implementation of discriminatively trained, deformable part models, a reliable and award winning object detection system. The detector relies on a deformable structure of multiple detection filters made from histograms of oriented gradients to enable the detection of objects that appear in a wide variety of configurations. We develop a clean C++ implementation that is faster than the code provided by Felzenszwalb et al, the creators of the system, without loosing any detection performance. We exploit the convolution theorem to speed up time critical parts and show that it is possible to significantly decrease computation time.

Zusammenfassung

In dieser Abschlussarbeit stellen wir die Implementierung eines zuverlässigen Objekterkennungsverfahrens vor. Das System baut auf einer verformbaren Anordnung von mehreren Erkennungsfiltern (auch Teilefilter genannt) auf, welche aus Gradientenhistogrammen zusammengesetzt sind, die in der Lage sind, Objekte mit einer großen Vielfalt an Erscheinungsmöglichkeiten in Bildern zu erkennen. Wir entwickeln eine saubere Implementierung, die schneller ist als der von den Erfindern des Verfahrens veröffentlichte Code, ohne dabei Erkennungsleistung einzubüßen. Dazu nutzen wir das Faltungstheorem, um zeitkritische Bereiche im Code zu beschleunigen und zeigen, dass die Berechnungszeit deutlich verkürzt werden kann.

Contents

Chapter 1

Introduction

The field of computer vision has gained a lot of attention within the last years, as increasing computational power allows for very sophisticated algorithms in everyday devices. Beyond the task of enhancing images in creative and design fields, the industry discovered the professional use of image processing algorithms for monitoring processes first. Through the last years, cameras have become an important and often "invisible" part of our lives: smartphones automatically detect faces and facial expressions, cars assess their immediate environment and assist their drivers in automatic braking actions and remind them of speed limits, and satellites monitor the weather around the world in real-time by observing the planet from space.

One of the probably most important and well researched tasks in the image processing domain is the unsupervised detection of objects in images. Exponentially growing disk space lets image databases all around the world grow very fast. Automatic analysis and interpretation of images is not only interesting for secret services, but also for image hosting platforms like Flickr and imgur or companies like Google, for very different reasons like improving the search for specific images or analyzing an image in order to display appropriate advertising near it.

This Master of Science thesis analyzes and reverse engineers one of the most exiting state-of-the art object detection systems. The desciminatively trained, deformable part based models introduced by Felzenszwalb et al. in 2006 [8] won multiple awards and are one of the most noted and researched systems in the object detection community in the last years. They use the histograms of oriented gradients framework [2] and build filters to allow for computing heatmaps that indicate the position and size of object instances in images. A deformable configuration is defined for multiple filters that cover and detect parts of the desired object class and are therefore much more adaptable to wide variations in appearance of objects in images.

The creators published a research implementation of the system that works excellent with only one flaw: it is very slow. This thesis focuses on implementing a clean version of the object detection code while trying to optimize certain slow

parts and therefore increase the execution speed.

We will at no point discuss the steps necessary to train the system and refer the reader to [8, 5, 9] for more details.

Overview

This thesis has the following structure: we give insight into a few general techniques used by the algorithm in the chapter 2. Chapter 3 introduces the basic concept and functionally of the object detection system, followed by a deep analysis of the research code. In chapter 4, we present an own implementation and discuss changes in methodology. In the last chapter, we compare our code versions against the original implementation and discuss the memory footprints and time requirements as well as their average precision.

Chapter 2

Preliminaries

In the following chapter, we will give some insight into fundamental knowledge needed in order to fully grasp Felzenszwalb's object detection approach. In the first section, we discuss histograms of oriented gradients as the feature of choice. Section two gives insight into the basic notion of filtering as a general image processing procedure, but also in context of scoring filters made of dense grids of features. The last section shortly explains a general distance transform, an efficient way of computing all shortest pairwise distances between points in a discrete array of arbitrary function values.

2.1 Histograms of Oriented Gradients

In order to get an idea about histograms of oriented gradients, we discuss the concept and application of gradients first. Let $f(x_1, \ldots, x_n)$ be an arbitrary function with n parameters. A **gradient** ∇ of this function is defined as

$$\nabla f = \begin{pmatrix} \frac{\partial}{\partial x_1} f \\ \vdots \\ \frac{\partial}{\partial x_n} f \end{pmatrix}. \tag{2.1}$$

Thus, a gradient is a vector with the partial derivatives of a function f as entries. We like to point out that the term gradient in general does not describe just one vector, but a field of vectors, one for each combination of input parameters in the original function. We will only discuss 2-dimensional functions and thereby 2-dimensional gradients, as a digital image is basically a function with two input parameters x and y denoting a pixel position on a discrete grid and an intensity value/triple as the output.

A 2-dimensional gradient vector in a plane has two important properties, namely a **magnitude** or length and a **direction**. The magnitude of a 2-dimensional intensity gradient $\mathbf{v}$ at (x, y) is given by

$$r(x, y) = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle} = \sqrt{x_1^2 + x_2^2}, \tag{2.2}$$

while the direction of the vector can be computed by

$$\Theta(x, y) = atan2(x_2, x_1). \tag{2.3}$$

Considering an arbitrary 2-dimensional function, each gradient vector points to the greatest rate of increase in a region, and the length of a gradient vector is an indicator for the steepness of the slope.

As the notion of using gradients in image processing is fundamental for understanding Felzenszwalb's approach, we like to discuss gradients in this context more vividly. An image can be seen as a discrete 2-dimensional function. The two finite input parameters x and y describe the coordinates of every pixel in that function, and the intensity value at a pixel position is the output of the function.

The size of an image is finite, so we can compute a finite number of gradient vectors for it. The gradient is defined through the partial derivatives of a function, so the question arises how a discrete function like an image may be derived. In image processing, the derivation of an image can be approximated by means of differential filtering. A very popular smoothed differential filter often used for edge detection is the Sobel-operator. It is defined as

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.4}$$

and

$$S_y = S_x^T \tag{2.5}$$

for the derivation of the image in x and y direction, respectively. Such filters are applied to an image via discrete convolution, given by

$$(I * F)(x, y) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I(i, j) \cdot F(x - i, y - j), \tag{2.6}$$

whereas $I(x, y)$ denotes the image an $F(x, y)$ an arbitrary filter. Note that the filter response may be smaller than the original image as border pixels don't have enough neighbor pixels to compute a correct response. It depends on the application whether either the missing neighbor pixels are interpolated or the gradient computation of border pixels is ignored completely.

In order to compute the gradient field of an image, two gradient filters are applied to the image, one for each dimension. A gradient vector for a pixel (x, y) is then formed combing the entries of both filter response maps at the position (x, y) to one vector.

The basic notion behind using intensity gradients for object detection is to describe an object through its edge distribution. The magnitude of a gradient vector of a homogeneous area in an image is zero. A gradient vector at an edge though does not only have a high magnitude indicating the intensity of that

edge, it also represents the edge's orientation as it always has a perpendicular orientation towards that edge.

Considering every gradient vector of an image for object detection is not useful for two reasons: it would be computational infeasible and, more importantly, a model that considers all gradient information of an object would not be robust against affine transformations like rotation, scaling, displacement and distortion of the object in different images. This is why groups of gradient vectors are combined to a feature called histograms of oriented gradients.

The term **histogram** denotes a representation of the distribution of data. It joins values of a given continuous variable together in non-overlapping **bins**, i.e. ranges of values, in order to give a density estimation of that data. The very common application of this statistical approach in image processing are color or intensity histograms. An intensity histogram depicts the distribution of intensity values in an image by simply counting the occurrences of intensities and visualizing them in a bar graph. To count every intensity individually leads to an unnecessary fine-grained histogram. Usually, the bin size is increased which leads to a coarser histogram resolution. Increasing the bin size to e.g. 10 means that all pixels with a value between 0 and 9 are pooled in the first bin, values 10 to 19 are summarized in the second bin and so forth.

The basic notion of histograms is applicable to a gradient as well. Recall the orientation $\Theta(x, y)$ and the magnitude $r(x, y)$ of a gradient vector at a position (x, y) in an image. The gradient orientation can be discretized into one of p bins. While the size of a bin in an intensity or color histogram is a range of intensity values, it is a range of angle values in the context of gradient histograms.

There are two different definitions for discretization of gradients, given by

$$B_1(x, y) = \text{round}\left(\frac{p \cdot \Theta(x, y)}{2\pi}\right) \bmod p \tag{2.7}$$

and

$$B_2(x, y) = \text{round}\left(\frac{p \cdot \Theta(x, y)}{\pi}\right) \bmod p \tag{2.8}$$

where B_1 denotes a contrast sensitive feature while B_2 represents a contrast insensitive definition. Note that the only difference between both is that the orientation is divided by 2π or π, respectively. The division by π leads to vectors pointing in opposite directions (more precisely: opposite bins) being joined in one and the same bin for the insensitive case. Hence, the insensitive definition does not consider whether the gradient is computed in an area where the pixel intensity along its orientation increases or decreases; like a straight line g which is rotated with π at any point $P \in g$, mapping the line on itself, it just captures the presents and orientation of an edge.

With either one of the definitions we might define a feature vector at (x, y)

$$F(x,y)_b = \begin{cases} r(x,y) & , b = B_{1/2}(x,y) \\ 0 & , otherwise \end{cases} \tag{2.9}$$

with $b \in \{0, \cdots, p-1\}$. For each pixel (x, y), the gradient vector is discretized to one of p bins.

Histograms of oriented gradients (HOG), initially proposed by Dalal and Triggs in 2006 [2], is formed by spatially aggregating edge histograms in cells $C(i, j)$. A cell is a square region within the feature map of a $w \cdot h$ image. Let $k > 0$ be a parameter specifying the side length of a cell, then all feature vectors $F(x, y)_b$ in each cell are summed up element-wise for a dense grid of $i \cdot j$ cells with $0 \leqq i \leqq \lfloor (w-1)/k \rfloor$ and $0 \leqq j \leqq \lfloor (h-1)/k \rfloor$. It is common to use a bin size of $p = 9$ and a cell size of $k = 8$. This leads to a p-dimensional feature vector for each HOG-cell. Dalal and Triggs proposed **normalization** and **truncation** of HOG-features in order to gain invariance to changes in bias.

Normalization of a feature $C(i, j)$ is achieved via normalization factors $N_{\delta,\gamma}(i, j)$ with $\delta, \gamma \in \{-1, 1\}$, given by

$$\begin{aligned} N_{\delta,\gamma}(i,j) = (\|C(i,j)\|^2 + \|C(i+\delta,j)\|^2 + \\ \|C(i,j+\gamma)\|^2 + \|C(i+\delta,j+\gamma)\|^2)^{0.5}. \end{aligned} \tag{2.10}$$

Hence, we obtain four normalization factors for all tuples of (δ, γ). Normalization takes 4-tuples of HOG-cells according to equation (2.10) and sums up their overall gradient energy. Component-wise truncation $T_\alpha(C(i, j))$ denotes the process of replacing an element c_b in the feature vector $C(i, j)$ with $\min(\alpha, c_b)$. The final HOG-feature is then defined as a 9×4 matrix

$$H(i,j) = \begin{pmatrix} (T_\alpha(C(i,j))/N_{-1,-1}(i,j)) \\ (T_\alpha(C(i,j))/N_{+1,-1}(i,j)) \\ (T_\alpha(C(i,j))/N_{+1,+1}(i,j)) \\ (T_\alpha(C(i,j))/N_{-1,+1}(i,j)) \end{pmatrix}. \tag{2.11}$$

As a result of normalization, the features at the border of the cell based feature map should not be considered for further computation due to the fact that the complete normalization takes the gradient energy of all eight neighbors of a cell into account, so border cells can't be normalized properly. Commonly, these features are discarded.

Felzenszwalb et al. proposed to use smaller features in order to minimize the number of parameters in their models and speed up the detection and learning processes. They analyzed the principal components [10] of a collection of HOG-features and derived a 13-dimensional alternative feature that obtained the same performance as the original 36-dimensional one. The new feature is computed by summing over all four normalizations and and by summing over the nine

orientation bins in the feature matrix 2.11. This leads to a $9 + 4$ - dimensional new contrast insensitive feature. Also, their findings indicated that some object classes benefit from contrast insensitive features while the detection performance improved for other classes using contrast sensitive features.

The final system implements features that combine both contrast sensitive and insensitive features. Recall that the original insensitive feature, without normalization and truncation, has $p = 9$ dimensions. We can easily define a contrast sensitive feature through equations (2.9) with (2.7) and $p = 9$ and concatenate both to a $(9 + 18)$-dimensional feature vector. Equation (2.11) tells us that the normalized version is a matrix with $(9 + 18) \cdot 4 = 108$ entries. The improved version of this feature vector is then obtained by summing over the $9 + 18 = 27$ columns of the matrix and concatenating it with 4 sums over the orientation of the 9 contrast insensitive orientations. This leads to a final feature vector length of $27 + 4 = 31$ dimensions. Felzenszwalb et al. add another entry at the end of each feature, called a truncation feature, that is usually set to 0 except for artificially generated features for padding, where it is set to 1 in order to indicate the difference between a padded border cell and a "real" feature.

The whole process of compressing the features has a very simple reason: in further computation, extensive amounts of dot products of different HOG-features are computed. Hence, it is very desirable to keep the feature length short as this massively shortens computation time. The improved features are much shorter without a significant loss in performance, according to [8].

Apart from normalization and truncation, **soft-binning** is applied as one further optimization of HOG-features on cell level that improves robustness of the whole system. The idea behind soft-binning is that the gradient at a pixel (x, y) should not only give its whole energy, i.e. magnitude, to just one cell, but also to adjacent cells as well. We examine one gradient in one of the four quadrants of a hog cell. Every quadrant is neighbor to three adjacent HOG-cells. Soft-binning computes the distance of the gradient's position to the center of its own cell and to its quadrant's cell borders. If its position is very close to the center of its own cell, almost the whole gradient magnitude is added to its own histogram. The closer it is to the neighbor cells though, the more of its energy is added to the corresponding bins of the adjacent histograms. This makes a complete HOG-map much more robust against small displacement of objects in images.

2.2 Linear filtering

Filtering is a very basic concept in image processing. It is for example used for finding edges (differential or high- pass filters) or smoothing (low-pass filters) images. The concept of filtering can also be applied for scoring specific filters on feature maps in order to find objects in images. We briefly discuss filtering

methods in this section and point out peculiarities we take advantage of in our own implementation of the object detection system.

2.2.1 Filtering in Image Processing

In section 2.1, we discussed how a low-pass filter like the Sobel-operator is applied to an image via discrete convolution 2.6. A **filter** F, sometimes also referred to as kernel, is a matrix that in most cases of classical filtering is very small compared to the image I it is applied to. The convolution $F * I$ takes this filter, mirrors it at its center, and computes a convolution result for every subwindow in an image through the dot product of filter and subwindow. It is difficult to obtain the complete convolution of a filter with an image for all border pixels, as these do not have enough neighbors. In image processing, missing pixels are often interpolated, e.g. by mirroring the image onto the missing regions. Besides interpolation, the convolution in border regions can also be rejected, especially for very small filter kernels.

To avoid confusion in the following course of this thesis, we like to point out a small fineness in terminology: In their original paper [8], Felzenszwalb et al. talk about taking the dot product of a filter and the subwindow of a feature map, while comments in the code talk about convolution for this process. Both dot product and convolution are two related mathematical operations, hence we can say that a convolution is a series of dot products with a mirrored filter. Also, a dot product is a convolution were we mirror the filter first and then restrict the convolution parameters i, j in equation (2.6) to just one position, discarding the sum.

2.2.2 Score Filtering

Apart from computing gradients of an image, there is a second application using the concept of filtering in Felzenszwalb's object detection system: scoring a filter on a feature map.

Therefore, we use special filters, consisting of concatenated HOG-features, although the general idea does not depend on a specific feature framework. A **HOG-filter** can simply be obtained by computing a feature map as described in section 2.1 and cutting out an arbitrary subwindow. In general though, these filters are trained with a discriminative procedure from large sets of labeled images.

We can easily observe the notion of using gradients and specifically HOG-features for object detection in figure 2.1. The HOG-filter on the right side was trained for detecting people in images. We can slightly recognize the abstract contour of a person in that filter. Recall that the HOG-features are computed from gradients which indicate the presence of edges in an image. The outline of a person in the derivate of an image leads to very clear edges, which ultimately appear very distinctive in the feature map.

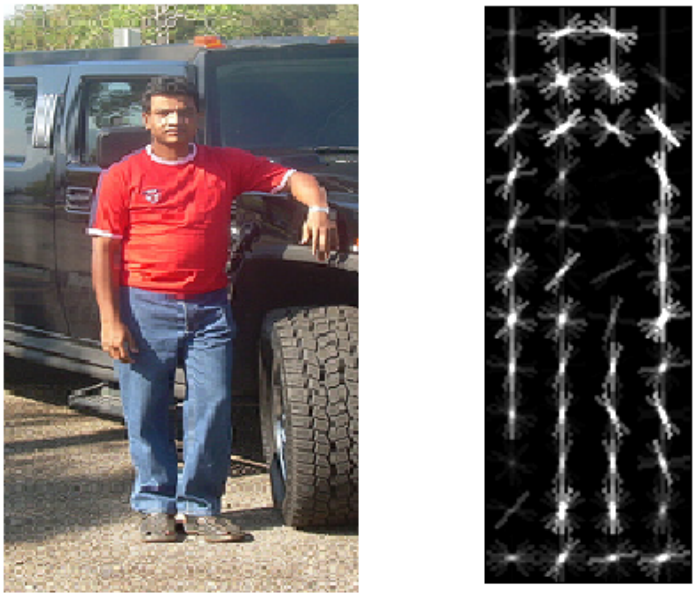

Figure 2.1: A HOG-filter (left) trained from a set of images depicting persons (right) (source: [8]).

In 2006, Dalal and Triggs proposed a methodology [2] for using HOG-filters for object detection. They trained a filter for detecting persons and applied it by computing the HOG-feature map of an image and thresholding the dot product of the filter and every subwindow in the feature map. If there was a person in the image, then certain subwindows of the feature map obviously would be very similar to the feature map of the filter, and the dot product between both would lead to a very high value, or **score**. If the filter covered a subwindow depicting a scene from the background though, then the score would be significantly smaller. As this approach would only work for people that have the same size as the trained HOG-filter in the image, they rescaled the image multiple times, repeating the computation for each scale. The set of all filter scores will be denotes a **response map** from this point on.

The whole procedure closely resembles the convolution of a normal filter kernel with an image with one important difference: the dot product between a HOG-filter and the feature map intuitively makes sense at positions where the filter completely covers a subwindow of the feature map and the features completely overlap. The dot products at all other positions don't yield any useful information, which means that we can simply restrict the number of filter positions to meaningful positions. In the following, we will refer to this "restricted" way of filtering as **block-wise** convolution.

2.2.3 Filtering in Frequency Domain

We can implement a fast convolution between a filter and an image by taking advantage of properties of frequency space to minimize the number of arithmetic operations.

We stated earlier that an image (and also a filter) is a special case of a 2-dimensional function. The 2d discrete fourier transform (DFT, [11]) is a powerful tool in digital signal processing and expresses a discrete time-domain signal in frequency domain as a sum of weighted sine and cosine basis-functions. Once in frequency domain, the **convolution theorem** allows for finding a computational less expensive implementation of the convolution in time domain.

Let $f(x_t, y_t)$ and $g(x_t, y_t)$ be two discrete, 2-dimensional functions, which can be transformed by a 2-dimensional discrete fourier transform through

$$\mathcal{F}(f) = F(u,v) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) \cdot \exp\left(-j2\pi \left(\frac{um}{M} + \frac{vn}{N}\right)\right), \tag{2.12}$$

with j being the imaginary unit and $M \times N$ the size of the DFT. The back-transformation is further

$$\mathcal{F}^{-1}(F) = f(m,n) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(u,v) \cdot \exp\left(j2\pi \left(\frac{um}{M} + \frac{vn}{N}\right)\right). \tag{2.13}$$

Then the convolution theorem states:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \Leftrightarrow f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)). \tag{2.14}$$

In other words, we can express the convolution in time domain by transforming both functions to frequency domain, computing the element wise product and transforming the result back. This would be much faster, as the number of element-wise operations is far smaller than the number of multiplications and summations in the conventional convolution. This of course needs a very fast transformation to and from frequency domain. The (inverse) fast fourier transformation ((i-)**FFT**) offers a very fast implementation for both directions under certain conditions.

The convolution theorem also applies to image filtering, where both functions are replaced with the image and the filter (or featuremap and HOG-filter), respectively. If we like to implement this procedure, there are several tripwires we have to avoid. Both functions have to be zero-padded to have the same size. The size of the discrete fourier transform is always the same as the size of the discrete function in time domain. Unfortunately, to retrieve the correct convolution result, both maps have to be padded even further. The reason for this is that the convolution theorem performs a circular convolution which wraps the filter around the image borders if parts of it do not completely overlap with the image. The solution to this problem is just enough padding. For one dimension, let L be

the side length of the image and P the side length of the filter, then the ideal size N for the FFT is

$$N = L + P - 1. \tag{2.15}$$

This rule also applies to the second dimension.

We mentioned that the FFT is a very fast implementation of the DFT under the condition that the size of every dimension of the FFT has to be a power of 2. In most cases this means that both image and filter have to be padded once more in order for the FFT to work fast. Even with this much padding, experience shows that in combination with a good FFT library, convolution can be implemented much faster in fourier-space. Block-wise convolution though cannot be replaced as the convolution theorem needs all values of the functions in order to work correctly. Nevertheless, we present a workaround in chapter 4.

2.2.4 Overlap-Add and Overlap-Save

In the previous section, we listed three padding steps necessary to compute the correct convolution result in frequency space. We like to briefly introduce algorithms that minimize this padding by disassembling the convolution theorem into blocks which can be processed successively.

The general steps are the same as explained before. The difference is that the image, which should be larger that the filter, is cut into pieces/blocks and the convolution theorem is applied in the same way. The only difficulty that arises is that the response maps have to be stitched together in the correct way.

For the sake of simplicity, we explain the algorithms just for 1-dimensional functions. All steps can simultaneously be applied to the second dimension without limiting the functionality. Recall that N is the size of the FFT and L and P denote the sizes of image (here: image pieces) and filter, respectively.

The **Overlap-Add** algorithm implements the convolution theorem by cutting the image into non-overlapping blocks with the length L. A FFT with the length $N = L + P - 1$ is computed for this block and, as the size of a block is known beforehand, was also computed for the filter. After the element-wise multiplication, the inverse transform of the result is computed. The result blocks *overlap* at $P - 1$ values. The overlapping areas are simply *added* together in order to form the complete, correct convolution response.

This approach has two advantages. First, the length N of a FFT is much smaller for one block as one block is much smaller than the complete image. Smaller FFT sizes should be preferred as their transformation is usually much faster. Second, we have to compute the FFT of the filter only once, as the size of the FFT stays the same for all blocks. Also, two smaller FFTs are much more likely to fit in the cache of a processor, which speeds up computation time as page misses are avoided. In fact, we try to exploit this circumstance in our implementation in chapter 4.

The second approach for computing a block-wise convolution theorem is called **Overlap-Save**. This method cuts the image into blocks that overlap at $P-1$ values. It computes a FFT with the size $N = P$. According to equation 2.15, this FFT is to short and produces false values, i.e. the first $P-1$ values are affected. This is not a problem as the blocks overlapped in the first place. The last L values are the correct ones, and in contrary to Overlap-Add, the single blocks just have to be concatenated in order to reproduce to correct convolution result.

2.3 Distance Transform

We discuss general distance transforms of sampled functions [4] in the last section of this chapter, yet another important basic tool in the field of computer vision. A distance transform of an image specifies the distance between a pixel and its closest non-zero pixel, for all pixels in the image.

The **traditional distance transform** D_P is defined by

$$D_P(p) = \min_{q \in P \setminus \{p\}} d(p, q). \tag{2.16}$$

$d(p, q)$ is a distance measure between a set of points q and a point p, and the distance transform searches for the closest point q that has a non-zero value.

Felzenszwalb and Huttenlocher proposed [4] a generalized version of this classical transform. Let $\mathcal{G}$ therefore be a regular grid and $f : \mathcal{G} \to \mathbb{R}$ a function assigning a real number to each grid cell. This of course is a generalization of an image definition. They defined a **general distance transform** to be

$$D_f(p) = \min_{q \in \mathcal{G}}(d(p, q) + f(q)). \tag{2.17}$$

Additional to the simple transform, each point in the grid now has a function value that influences the distance in a certain way, where larger function values of course lead to a larger distance transform.

The question of interest is how to compute such a transform for the whole grid efficiently. The original work proposes an algorithm which solves the transform in linear time. They use the squared Euclidean distance, which leads to

$$D_f(p) = \min_{q \in \mathcal{G}}((p - q)^2 + f(q)). \tag{2.18}$$

For a one dimensional grid, the distance transform for each point $q \in \mathcal{G}$ is bounded by a parabola rooted at $(q, f(q))$. The set of all parabolas then forms a lower **envelope** (figure 2.2), which exactly defines $D_f(p)$. The computation of this envelope is the essential step in finding the distance transform.

In order to grasp the basic notion of envelope computation, we examine three directly adjacent parabolas a, b and c on a 1-dimensional grid, where a is a neighbor to b and b is a neighbor to c. The algorithm proposed in [4] studies the

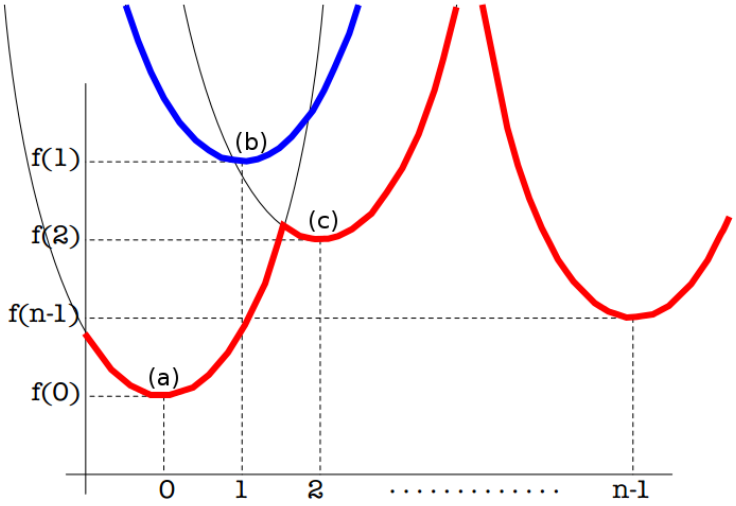

Figure 2.2: The set of n parabolas defining constrains for the distance transform, which is given by their lower envelope (red). The algorithm excludes parabolas (blue) if they do not contribute to the envelope. (source: [4])

intersections between the parabolas: if the position of the intersection between a and b is to the right of the position of the intersection between a and c, then b obviously does not contribute to the envelope. Figure 2.2 depicts this situation clearly. The algorithm can iteratively extend the envelope to the right, always checking the intersections and discarding parabolas that do not contribute.

We can easily extend this approach to the two dimensional case. Therefore, we compute the envelope for each row of the grid first and discard all unnecessary parabolas. Similarly, on the basis of all horizontal envelope, we can iteratively scan through all columns and reject more parabolas that do not support the envelope in this dimension.

We do not discuss the implementation of this procedure in detail here and refer the reader to [4] for further details and a pseudocode implementation. Felzenswalb et al. modify the general distance transform in order to do the exact opposite, i.e. maximize grid values, which allows for high values to spread across the grid. This of course does not change the general approach.

Chapter 3

Algorithm Analysis

Within the last years, research in object detection has focused on recognizing generic classes of objects rather than specific instances. Felzenszwalb et al. [7] introduced a pictorial structure representation, where an object is modeled by a collection of parts that are somehow arranged in a deformable configuration. Therefore, each part represents a visual property of an object class, and the deformable configuration is represented by a graph structure with spring-like edges. Such a model is applied to an image by minimizing an energy function, taking the deformation costs of parts into account. Imagine a pictorial structure for persons: One part is a detector for the whole body, while there are several parts specifically trained to detect parts of a person, e.g. the head, arms, legs and the torso. As the appearance of a person is more dynamic, the deformable configuration of such a model allows to catch a wide variety of appearances in images.

3.1 Object Detection with Pictorial Structures

In the next section, we introduce deformable part models as a specific implementation of pictorial structures. The main focus of this work is the analysis and efficient implementation of the detector, which is why we will ignore the training of the models. We will give an insight into the structure of a deformable part model and focus on the question how it is applied to an image.

3.1.1 Deformable Part Models

Discriminatively trained, deformable part models take the notion of pictorial structures and model an object through a star-structure with one part defining a coarse root filter and several higher resolution filters modeling parts of an object class. Filters, root and parts, are represented by linear HOG-feature maps introduced in section (2.1) which are matched with feature maps as explained in section (2.2).

A deformable part model is formally defined by a $(n+2)$-tuple

$$M = (F_0, P_1, \cdots, P_n, b) \tag{3.1}$$

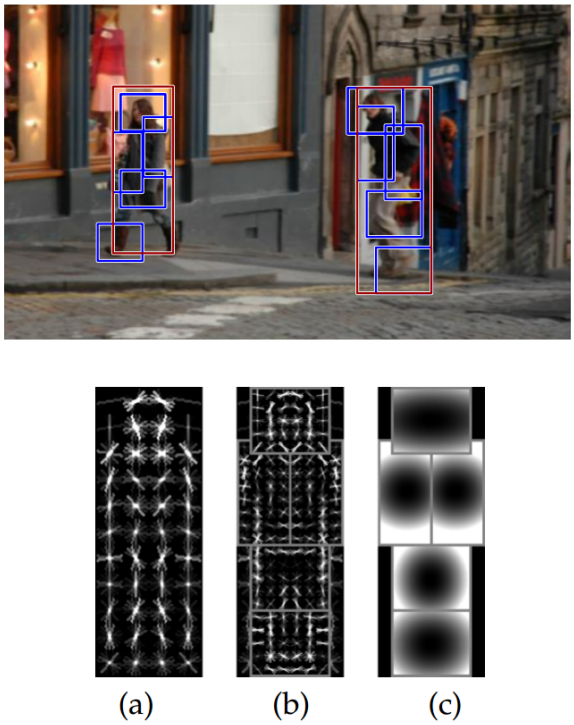

Figure 3.1: A simple deformable part model for persons. Feature map (a) defines a coarse root filter covering the whole object, while (b) is a collection of part filters for smaller parts if the object. (c) depict deformation costs for the parts. (source: [8])

where F_0 is the root filter, P_i is a template for a part i and b is a bias value for that model. A part P_i is furthermore a 3-tuple

$$P_i = (F_i, v_i, d_i) \tag{3.2}$$

with F_i being a filter for a part of the object with an anchor position v_i relative to the root filter F_0 and deformation cost d_i which allow for this part to have a certain displacement from its intended position v_i. Figure 3.1 depics a simple deformable part model for persons.

Recall that the object is represented by HOG-feature maps. Even though histograms of oriented gradients are slightly invariant to very small differences in size of an object, they are not suited to cover all scales of objects. In order to define the score of a model for different object sizes, we specify a feature pyramid.

3.1.2 Feature Pyramid Scoring

A **feature pyramid** is a set of feature maps for a finite number of scales. Practically, a feature pyramid is computed by iteravely smoothing and subsampling an original image and computing the features for each scale. Therefore, the pa-

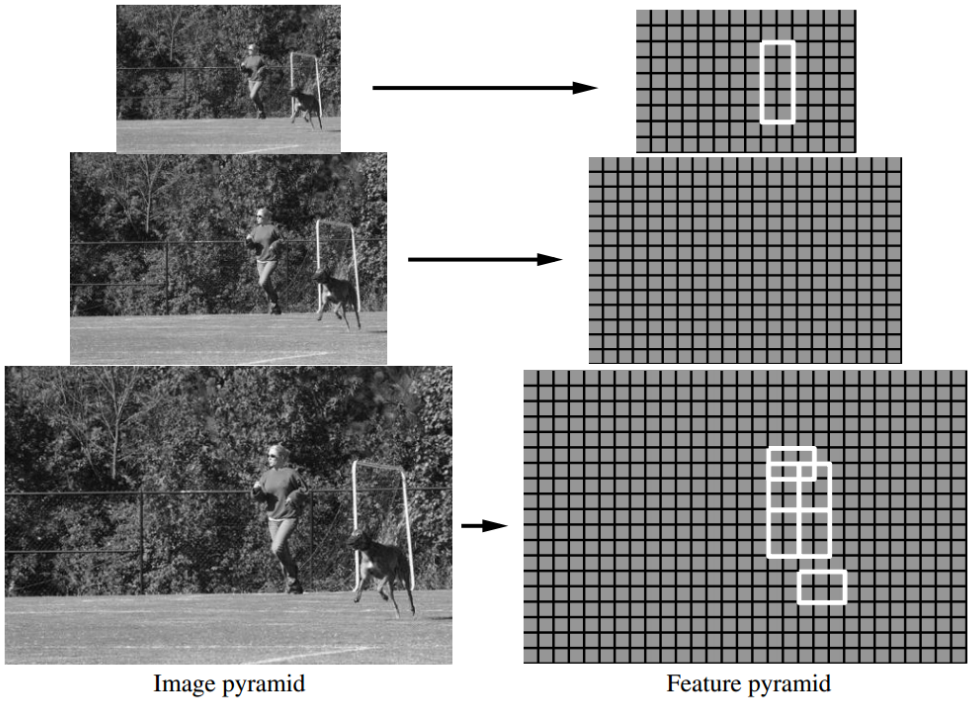

Figure 3.2: A feature pyramid is created by iteratively resizing the image and computing a feature map for each level. (source: [8])

rameter λ defines the number of levels in an octave as the distance between two layers in the feature pyramid, where the upper layer has twice the resolution of the lower one. Figure 3.2 depics the construction.

In practice, λ is chose to be $\lambda = 5$ for model training and $\lambda = 10$ for testing. This means that feature pyramids are much larger in test cases, which is important for obtaining a high performance with a model.

Contrary to figure 3.2, we define the feature pyramid in reverse order with the largest layer on the top, getting smaller as we move down the scales. This might seem unnecessary, but this definition is much more related to the existing MATLAB implementation.

With the term feature pyramid specified, we would like to define a score of a feature map. Let H be a feature pyramid and $p = (x, y, l)$ a position (x, y) within the l-th layer of the pyramid. Also, let F denote a $w \times h$ filter and $\Phi(H, p, w, h)$ a concatenation of all feature vectors in a $w \times h$ subwindow of H with p being the root of the subwindow. The score of F is then defined via

$$\text{score}(F) = \langle F', \Phi(H, p, w, h) \rangle = \langle F', \Phi(H, p) \rangle \tag{3.3}$$

with F' is the vector obtained by concatenating all feature vectors in F. Since the size of $\Phi(H, p, w, h)$ is implicitly defined by the size of F, we may write $\Phi(H, p)$ instead.

3.1.3 Deformable Part Model Scoring

Let $p_i = (x_i, y_i, l_i)$ be the placement of a filter f_i in the l-th layer of a feature pyramid. Since all part filters have twice the resolution of the root filter, they are placed one octave away from the root filter layer, with $l_i = l_0 - \lambda$ for $i > 0$.

While the score of a single filter at a specified position can be computed very easily, scoring a complete deformable part model is more complicated as the deformable configuration of the parts has to be taken into account. The score of a deformable part model is given by

$$\text{score}(p_0, \cdots, p_n) = \sum_{i=0}^{n} \langle F_i', \Phi(H, p_i) \rangle - \sum_{i=1}^{n} \langle d_i, \Phi_d(dx_i, dy_i) \rangle + b. \tag{3.4}$$

The first dot product computes the score of every filter with a subwindow as explained in section 2.2 and sums them up to an overall score. The second dot product takes the part-filter displacement into account with

$$(dx_i, dy_i) = (x_i, y_i) - (2(x_0, y_0) + v_i) \tag{3.5}$$

being the displacement of the filter relative to the root. The root position (x_0, y_0) is multiplied with the factor 2 to adjust it to the double resolution of the part's layer. The displacement v_i is added to the root filters position, setting the nominal position of the filter. Subtracting it from the actual position yields the displacement. The dot product of d_i with the deformation feature

$$\Phi_d(dx, dy) = (dx, dy, dx^2, dy^2) \tag{3.6}$$

yields a smaller value for the second sum in equation 3.4 for small displacements and therefore a higher overall score. The term b in equation (3.4) is a bias which makes different models comparable when combined to a bigger mixture model.

Given a feature pyramid and a deformable part model, we can detect an object within an image. The object detection process is an optimization problem, given by

$$\text{score}(p_0) = \max_{p_1, \cdots, p_n} (p_0, \cdots, p_n), \tag{3.7}$$

where the score of the root filter is extended with responses of the part filters in order to form a full model hypothesis.

To compute the best locations of the parts, the approach in [8] uses the generalized distance transform introduced in section (2.3). Let

$$R_{i,l} = \langle F_i', \Phi(H, (x, y, l)) \rangle \tag{3.8}$$

be an array storing the complete response of the i-th filter with the l-th layer of the the feature pyramid. A response is computed via a sliding window approach, introduced in section (2.2). It is then transformed in order to allow for spatial uncertainty,

$$D_{i,l} = \max_{dx,dy} (R_{i,l}(x + dx, y + dy) - \langle d_i, \Phi_d(dx, dy) \rangle). \tag{3.9}$$

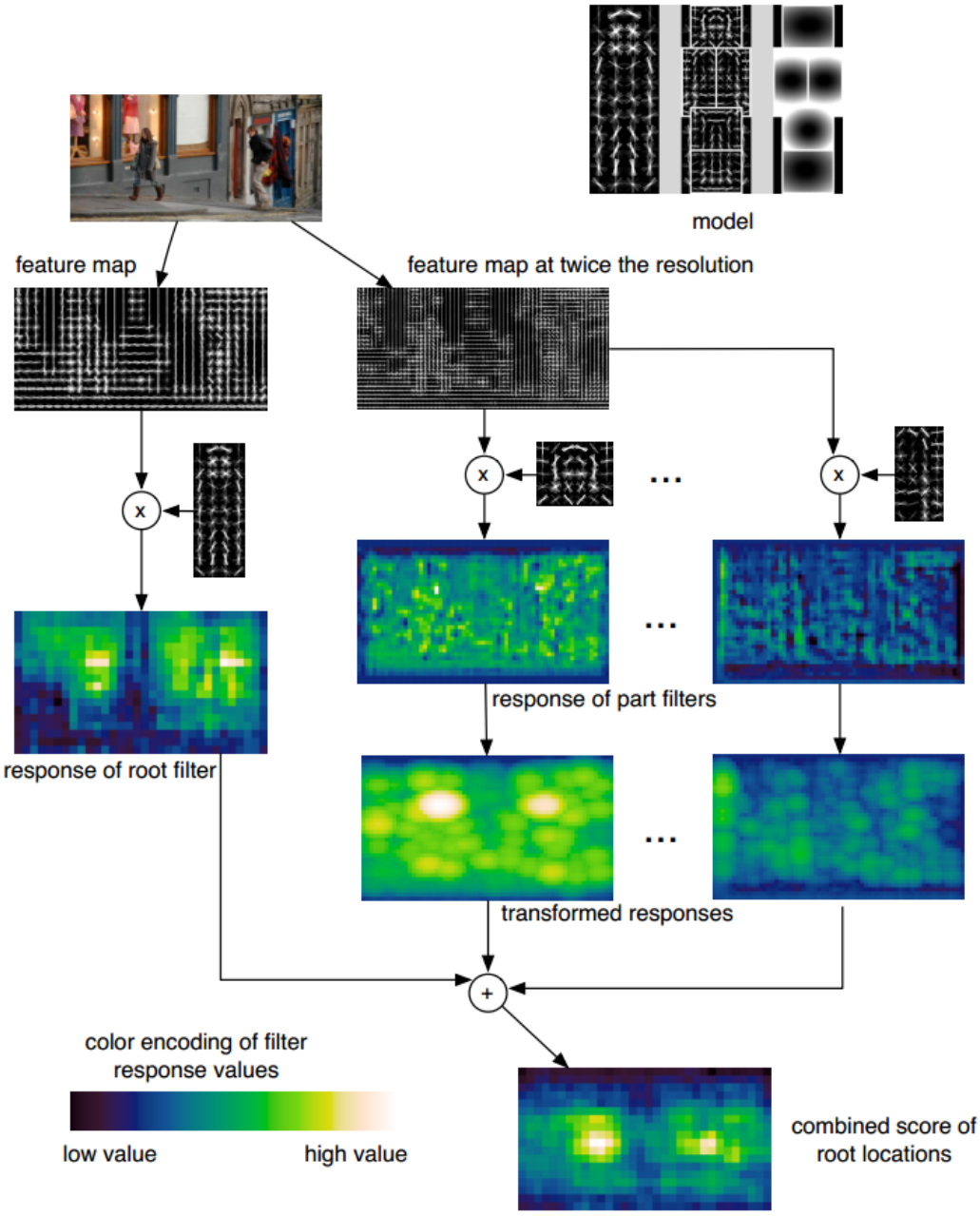

Figure 3.3: The matching process of a deformable part model at one scale. Filters for the root and the parts are convolved with feature maps from different scales in the feature pyramid. The responses are transformed to allow for spatial uncertainty. Summing up the transformed responses with the root response yields the overall score for the scale. (source: [8])

The transformation takes high scoring detections (of any filter) and spreads it around nearby locations. Figure 3.3 illustrates this process.

The form of equation 3.9 closely resembles the general distance transform 2.3 where $\langle d_i, \Phi_d(dx, dy)\rangle$ is a distance measure and $R_{i,l}(x + dx, y + dy)$ a function value.

The overall score of a deformable part model at a layer in the feature pyramid adds the response array of the root filter to the transformed and subsampled responses of the part filters, computed at one octave above the root layer, through

$$\text{score}(x_0, y_0, l_0) = R_{0,l_0} + \sum_{i=1}^{n} D_{i,l_0-\lambda}(2(x_0, y_0) + v_i) + b. \tag{3.10}$$

Figure 3.3 depicts the whole matching process at one specific scale.

3.1.4 Mixture Models

A problem that arises when using only one deformable part model (one root and associated part filters) for object detection is that the richness of object categories as well as viewpoint change and partial truncation cannot be captured by a single model. Imagine the object class of cars: they are built in different shapes and sizes and thereby vary greatly in appearance. Also, a model trained for detecting cars from the side would perform poorly on an image where a car is shown from a frontal viewpoint. We face the same problem if the camera viewpoint is not ideal or too close to the object; in both cases, the object will only be partially visible in an image (truncation). While heavy occlusion might not be a problem for most object classes, it is very desirable to deal with this problem for other classes. For instance, we want to detect the whole body of a person as well as any person on a passport photo where most of the body is truncated. Figure 3.4 illustrates this problem.

One way to deal with all problems mentioned above are **mixture models** as an extension to a single deformable part model. A mixture model is a m-tuple $M = (M_1, \cdots, M_m)$ where each M_i is a single deformable part model. We will refer to each of these models as **submodels** in the following. Combining different submodels to one bigger mixture model allows for dealing with abundance of object categories as well as viewpoint problems. In case of the car class, different models can be trained on different types of cars and on different viewpoints, i.e. for a frontal view on a class versus a side view. In case of a person class, one model might be trained to detect the whole body of a person while a second one recognizes truncated persons.

3.1.5 Post Processing

Matching a complete mixture model with a feature pyramid often yields multiple high scores for the same instance of an object and thereby a lot of bounding boxes framing that object. The original code uses a greedy **non-maximum suppression** procedure in order to eliminate double detections. All detections

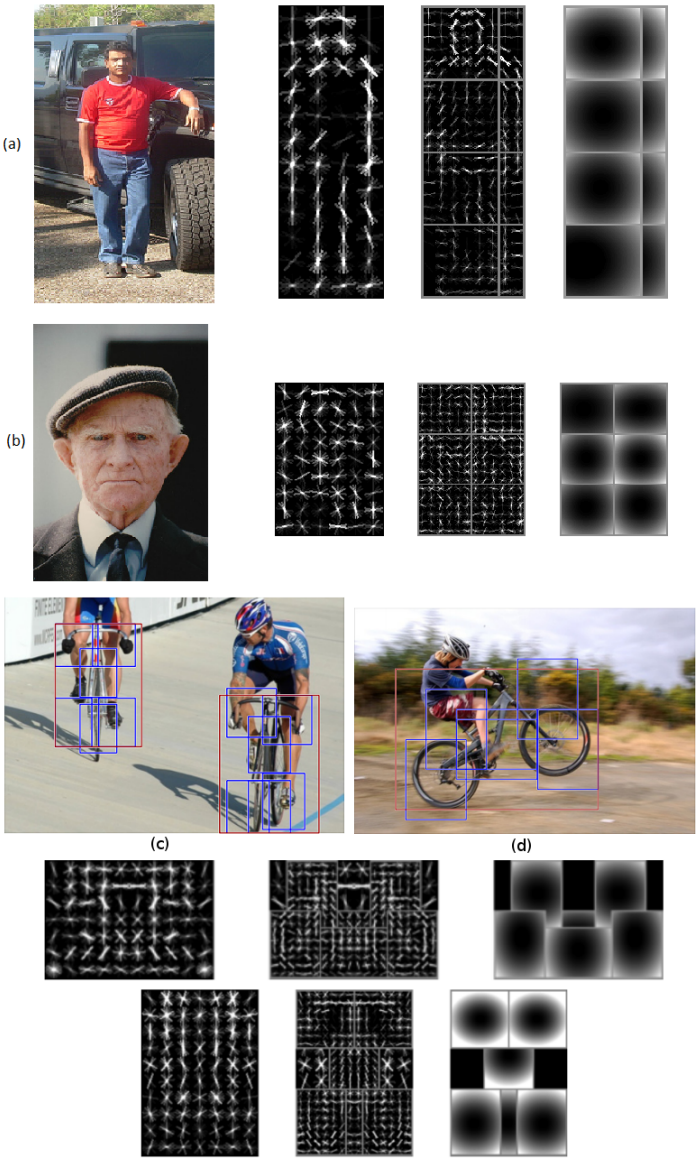

Figure 3.4: Different mixture models. Figures (a) and (b) depict a typical case of truncated objects, and their deformable part submodels. Figures (c) and (d) illustrate different perspectives on the same object, the second reason for combining multiple submodels to a mixture model. (source: [8, 3])

in an image are sorted by score, and the highest scoring ones are selected while detections that are at least 50% covered by a previous one are skipped.

The desired output of an object detection system is not clearly defined. The VOC-challenge [3] assesses the performance of an object detector though the bounding box of an object. Felzenszwalb et al. improve the bounding boxes with a **bounding box predictor**. The notion is that the best scoring positions of the

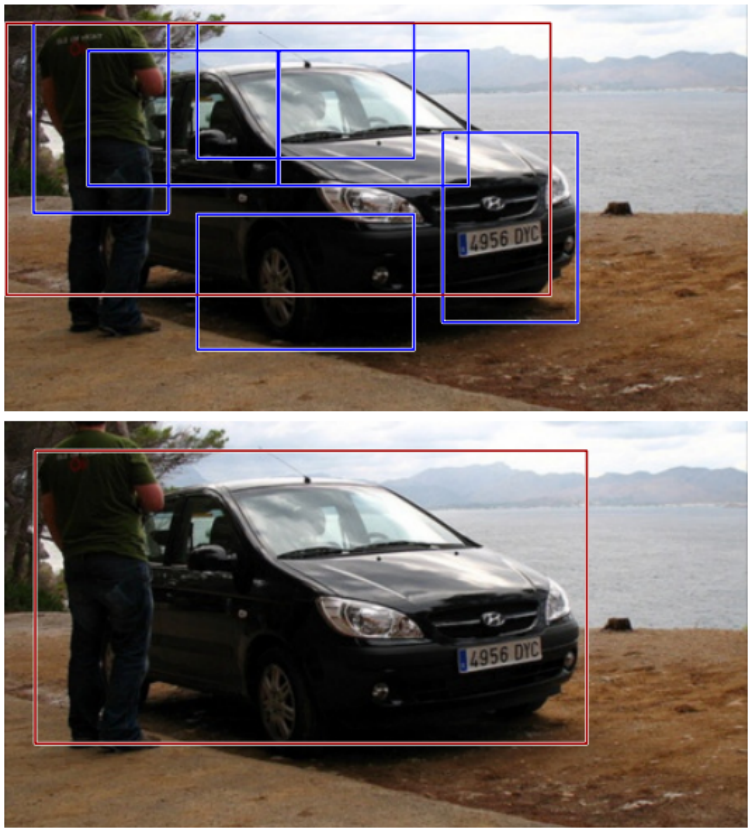

Figure 3.5: Bounding box prediction resizes a bounding box by taking the best scoring positions of the parts into account. (source: [8])

part filters within a detection describe the dimensions of an object instance far better than a single, coarse root filter. Hence, the bounding box predictor resizes a bounding box according to the positions of the part filters. Figure 3.5 depicts such an optimization.

3.2 Original Implementation Analysis

In the previous sections, we introduced the preliminaries for detecting objects in images with multiscale, deformable part models. In this section, we take an original implementation apart and analyze the different stages necessary to perform object detection.

3.2.1 Technical Overview

The algorithm we analyze here is the original implementation by Felzenszwalb et al [6]. Although the latest version number is 5, we work with version 4.1, as we have far more experience with it through previous work. Also, version 5

implements optimizations like the star-cascade detector which we did not want to consider in our implementation.

The aforementioned implementation was used in the Pascal Visual Object Classes Challenge (VOC-Challenge, [3]), where it won a lot of prices in different detection categories over the years. Hence, it works with the evaluation framework (VOCdevkit) provided by the challenge, which includes a vast amount of annotated images and algorithms for determining precision and recall as well as average precision computation. For all our tests, we used the VOCdevkit from the year 2010. This development kit includes images from 2007 to 2010. As a result, the models we used for evaluation, are trained on the 2010 database with the code version 4.1.

Version 4.1 of the code is generally implemented in Matlab, with time-critical section implemented in ANSI C. It runs on Windows and Linux. The 'compile.m' script compiles the C code into the Matlab friendly format '*.mexw64', which can directly be executed from the Matlab command shell.

3.2.2 Model Type

We give a short overview of the structure of a version 4.1 model for documentation purposes mainly. A model is saved in a binary format which can not be used in C++ straight away, which makes a translation of the models into an exchange format necessary. Each model saves more information than we actually need for a simple object detection application. The additional information is either needed for the training of the models, which we don't cover in this work, or for documentation and debugging. We confine the model description to the necessary elements in the following.

We address elements within the model type in an object oriented way with the member access operator. A model is a struct that has the following members:

- model.class: the object class name the model initially was trained for, e.g. person, car, cat, etc.
- model.year: the year of the VOC database this model was trained on
- model.filters: an array of filter data of this object model. Each array entry is a structure where model.filters.w hold the weights of each filter and model.filters.size the size of a single filter. As a filter is basically a patch of HOG-features, the weights matrix is a 3-dimensional matrix where the first two dimensions address the first element of every feature vector while the third dimension holds the actual feature data. Felzenszwalb et al. use HOG-features with a dimension of 32 (see section 2.1), which means that the third dimension of the weights matrix is always 32-dimensional. The size of each filter is therefore a two dimensional vector, holding the number

of cells of each filter. The filter array does not hold any information about the structure of the model, i.e. the filters are not sorted in any fashion.

- model.symbols: holds information whether a filter is a terminal 'T' or a non-terminal 'N'. We will give a short insight into the notion of representing objects in images with grammars in section (3.2.3).
- model.rules: represents rules in an object grammar. There are two kinds of rules for each model: deformation rules and structural rules. Structural rules hold the structural information of the model, i.e. which filters are roots, which are parts, how they form single submodels with their biases in model.rules.offset and what the ideal positions of a part relative to its root is (in model.rules.anchor). Deformation rules save the deformation costs for each filter in model.rules.def.
- numfilters: the overall number of filters in a model
- maxsize: the size of a filter that could cover all filters in the model, as its dimensions are the maxims over the dimensions of each filter.
- interval: size of one octave in the feature pyramid.
- sbin: length of one side of a quadratic cell on which a HOG-feature is computed, usually equals 8.
- thres: detection threshold for the whole model.
- bboxpred: a bounding box predictor. Resizes bounding boxes in order to improve model performance.

3.2.3 Algorithm Analysis

Before we start looking into details of the implementation, we give a short overview of the basic algorithm. We boiled the object detection process down to a very simple version:

```
process(image, model){
   feat = compute_feature_pyramid(image, model);
   responses = filter_pyramid(feat, model);
   apply_deformation_rules(responses, model);
   apply_structural_rules(responses, model);
   detections = threshold_detections(responses, model);
   bbox = compute_bounding_boxes(detections, model);
   bbox_im = improve_detection(bbox, model);
}
```

While the original code is of course much more convoluted, this flattened version captures all essential steps. A feature pyramid is computed from the

original image and every layer in the pyramid is convolved with every filter in the model (lines 2-3). To each layer, a distance transform is applied in order to allow for spatial uncertainty of parts (line 4). The transformed responses of part filters are then subsampled, shifted and added to the corresponding response of their root filter (line 5). The resulting response pyramids, one for each submodel, are then searched for maxima greater than a threshold (line 6). All these maxima correspond to positive detections, and from the position of a maximum and the size of the root filter of the submodel that created this maximum, we can derive the bounding box of a detection (line 7). Each bounding box may be improved using a bounding box predictor (line 8). With the position and size of a bounding box, we yielded the location and size of an object instance in the image.

We take a deeper look at the single steps in the following sections.

Feature Pyramid Computation

The first step before applying a deformable part model is to compute a feature pyramid. The filters in the model are of course rigid and can not be changed in size, so we have to adjust the size of the image in order find object instances with different sizes, as explained in section (3.1.2). The height of the feature pyramid depends on different factors. For one, the size of one octave o, as saved in model.interval and usually set to $o = 10$ for object detection, sets a scaling factor sc, where

$$sc = 2^{o^{-1}}. \tag{3.11}$$

Second, the smaller edge length of the image also contributes to the formula that computes the size maxscale of the feature pyramid,

$$\text{maxscale} = 1 + \left\lfloor \log\left(\frac{\min(\text{size}(\text{image}))}{5 \cdot model.sbin}\right) / \log(sc) \right\rfloor. \tag{3.12}$$

The factor '5' in equation 3.12 is a system-specific parameter set by the developers in the original code. Hence, an image with the size $800x600$ and an octave size of 10 would result in a feature pyramid with overall 40 layers. This is actually not the correct size of the final pyramid, as equation (3.12) only accounts for shrinking the image. In reality, one extra interval with larger feature maps is computed, which would lead to a pyramid size of 50 for our example.

Before an image is rescaled, it is converted to double. Computing the pyramid by iterative subsampling is done pretty efficient in the original code:

```
1 for i = 1:interval
2    // rescale image in octave
3     scaled = resize(im, 1/sc^(i-1));
4    // compute features for first interval
5     pyra.feat{i} = features(scaled, sbin/2);
6    // scale for second interval
7     pyra.scales(i) = 2/sc^(i-1);
```

```
8     // features for second interval
9      pyra.feat{i+interval} = features(scaled, sbin);
10    // all other intervals
11     for j = i+interval:interval:max_scale
12         scaled = resize(scaled, 0.5);
13         pyra.feat{j+interval} = features(scaled, sbin);
14     end
15 end
```

For the first two intervals of the pyramid, the code rescales the image only one time with the appropriate factor (line 3). The trick is to compute the feature map for the top interval from half the usual cell size ($sbin/2$ in line 5), i.e. using $4x4$ sized cells instead of $8x8$. This leads to a feature map with twice the resolution without having to rescale the image a second time. The rest of the pyramid is computed the common way though (lines 11-14) by consecutively rescaling the image with the factor 0.5 and computing the feature maps. This process creates the single layers in a distributed but fast way.

After resizing the image, a HOG feature map is computed for each layer in the pyramid. Computation of features is an expensive task which was outsources to C code by the developers.

We described in section (2.1) the steps we have to take in order to compute the features: for each pixel, compute the gradient with magnitude and orientation, snap the orientation to one of 9 (insensitive) or 18 (sensitive) bins, add the magnitude to the according bin and normalize the features. The first difference here is that color images have 3 channels, one for the color blue, green and red, respectively. This is simply solved by computing the gradients of all three channels separately and choosing the one with the highest magnitude, discarding the two remaining vectors. Instead of convolving each color channel with the Sobel-filter, gradient computation is done on the fly by simply subtracting the left pixel value from the right one for the $x-$derivative and and the top one from the bottom one for the $y-$derivative at an arbitrary pixel position. This procedure strongly corresponds to partially deriving an image via Sobel-filtering. In order to find the correct bin for each gradient, Felzenszwalb et al. use 9 normalized sample-gradients placed at the center of each bin. They compute the dot product between their gradient and each of the sample-gradients. The dot product leads to a very high value for the correct bin, and gets smaller for gradients that are further away from their correct bin. This is a brute-force method of course as they always have to compute 9 dot products for every gradient in the image. For a sensitive gradient they just have to negate the result of each dot product. This seems very inefficient though and we will introduce an optimization for this step in chapter 4.

The last steps in HOG-computation are soft binning and normalization. Soft binning uses linear interpolation to divide the magnitude of a gradient between

neighboring cells. The closer a pixel that corresponds to a gradient is to the border of a HOG-cell, the more of its gradient magnitude is added to the correct bin in the three directly adjacent cells of the cell quadrant. Normalization of groups of 4 adjacent HOG-cells is achieved as describes in section 2.1. Each entry in a feature vector is truncated with a truncation factor of 0.2, and the feature is divided by a normalization factor derived from the complete neighborhood of a cell. As the cells at the border of the feature map do not have to necessary 8 neighbors needed for normalization, they are discarded, leading to a feature map that lacks two rows and two columns of features. A feature vector is created with 9 contrast insensitive, 18 contrast sensitive, 4 "texture" and 1 truncation feature, which is always set to 0.0 here.

Padding completes the generation of each feature map in the feature pyramid. An object can be truncated in an image, which is why each feature map is padded with additional HOG-features at each side. The size of the padding is given though the maximal size of the filters (in model.maxsize) to assure that every filter in the model fits the padding zone. Furthermore, one extra line of feature is added to each side of the map to compensate for the HOG-features that fell victim to normalization. All entries but the truncation feature in the padding vectors have the value 0.0. The truncation component is set to 1.0. Padding is always done after the feature map is computed already, which is expensive due to memory reallocation.

This completes feature pyramid generation. We would like to note that the original implementation always saves all interim results. Additionally to the whole feature map, the scale of each map is also saved.

Response Maps

The most expensive operation in the whole algorithm is the computation of response maps by filtering every level of the feature pyramid with every filter in the model. The score of one filter at a position (x, y) on one layer of the pyramid, covering a $w \times h$ subwindow (section 3.1.2) is only one filter response value. As we are finally interested in all response values a filter produces in one layer, the filter is moved over the whole image in a scan line fashion, consecutively computing a response value at each position where it completely covers a subwindow.

Filtering is the most expensive computation step in the algorithm. The feature pyramid of an image can easily have 45 levels with the dimensions of 130×400 HOG-cells for the biggest and 30×92 for the smallest one. A complete deformable part model can easily hold more than 50 filters with an average size of 7×7 HOG-cells. Recall that each HOG-cell has the 32 dimensions, applying each filter to each level of the feature pyramid leads to a vast number of multiplications/additions, even when filtering only at every 32^{nd} position.

Felzenszwalb et al. offer multiple implementations for filtering (or feature-

convolution). All implementations are written in C, do the same and differ only in how long the convolutions needs. Execution speed is gained by using fast implementations of BLAS (Basic Linear Algebra Subprograms), or simple parallelization via multithreading (e.g. pthreads).

Apart from differences in execution time, the convolution process is the same in all implementations: the code takes a pair of pointers, one to the feature map and one the the first element of every filter, and computes the dot product of the filter and the correct subwindow by multiplying all entries, summing them and shifting the feature map pointer the proper amount of entries further for the next correct overlay. Note that the code actually computes the dot product of both instead of the convolution we would expect through the term of filtering. We pointed out the terminology differences with dot product and convolution earlier, and it is easy to replace the dot product with the convolution by simply mirroring the filters of the model prior to convolution.

The convolution function takes a feature map from the feature pyramid and convolves it with all n filters in one function call, leading to n filter responses. All filter responses of one pyramid level are then padded with $-\infty$ to the same size at the bottom and right border of each response map. The responses are then saved together with an additional 0-initialized response map with the same size which is needed in order to compute the overall score of the model later.

At this point, the memory footprint is huge due to saving all interim results. The fact that filtering has the longest runtime by far within the whole matching process gives reason to search for a way to make it much faster in our own implementation.

In the next step, we allow for spatial deformation of the parts of a model by applying a distance transform to the response maps.

Deformation and Structure

Deformable part models where only the first step in the development of object detection grammars according to Felzenszwalb et al. Although this notion of representing a visual object is mentioned only very briefly in [8], the original code implements a slight portion of object detection grammars, mainly to represent the structure (i.e. submodels, relation between roots and parts) of an object.

Grammar based models as a generalization of deformable part models represent objects using a hierarchical structure. An object can be defined directly or in terms of parts, which allows for modeling structural variations and sharing information between different models, e.g. different parts. As the usage of grammars in the context of deformable part models is poorly documented, an analysis of this part of the code is particularly difficult.

Noam Chomsky introduced a formalization of generative grammars in the 1950s [1], consisting of four basic elements: a finite set of non-terminal sym-

bols N, a finite set of terminal symbols T, a finite set of production rules and a distinguished start symbol $S \notin N$. Starting with S, production rules state recursive replacements within the set of all symbols.

We find this structure in the object detection code, too. The interesting members of the model that the object detection grammar operates with is 'model.symbols', 'model.rules' and 'model. filters'. The code currently implements two kinds of symbols, non-terminals 'N' and terminals 'T', which we find in the 'model. symbols' array. Each terminal symbol corresponds to exactly one filter, filed under 'model.symbols.filter'. Non-terminals don't correspond to any filter.

The third important field, 'model.rules', offers rules on how the symbols are arranged. We identified two different sorts of rules: structural and deformation rules. Structural rules describe the structure of the model, i.e. which terminals (and thereby filters) define a submodel. All important structural variables, like which one of the filters is the root and all anchor positions of the parts relative to the root, are also associated with a rule. Deformation rules link each filter to deformation costs and therefore give the parts of a model their deformable configuration. Although the name at first sight does not imply it, a deformation rule just holds information affecting the structure of the model, following the general notion of grammar based object detection.

We found that there is no particular order within all three fields. As the grammar is scattered over different data array as well as code fragments, it was particularly difficult to grasp its meaning. Our analysis induces the following interpretation: the underlying structure can be interpreted as a undirected graph, with a non-terminal root for the whole model. The root can be replaced by an arbitrary number of non-terminal symbols representing submodels, and each submodel symbol may be replaced by a variable number of terminals. Terminal rules hold a binary sub-tree which itself holds a specific filter response in one branch and a transformed one in the other. Each branch is created during the runtime of the algorithm.

The field 'model.start' points to a rule within the rules-array that holds n structural rules for n submodels. A lot of rules are empty and all rules that are not empty or hold a set of structural rules are deformation rules. The order at which each rule is applied is hidden within the code. The function 'model_sort.m' performs a topological sort of the non-terminal symbols in the model grammar. The result is a field with rule identifiers. The code calls each rule according to the order in this field. We found that all deformation rules are executed first and the structural rules are applied after each transformation.

In sections 2.3 and 3.1.3 we introduced the generalized distance transform and one application in the context of deformable part models. A deformation rule calls the distance transform on each response map with the correct deformation parameters for the filter that produced this map in order to spread high scoring

detections of parts. The implementation of the distance transform follows [4] and applies a horizontal 1-dimensional transform before repeating the procedure in vertical direction. Figure 3.3 illustrates the result of the algorithm.

As always, the creators save all interim results, in this case the transformed response maps, in the grammar tree. Also, the envelopes for both x and y-direction are saved separately and later used in order to detect optimal locations of part filters.

After all deformation rules are executed for all response maps, structural rules total them up to n score pyramids, one for each of the n submodels. We stated in section 3.1.3 that part filters are trained one pyramid levels with twice the resolution of the levels the roots where trained on, defining one octave within a pyramid. As a result, we have to "downsample" the resolution of transformed part responses.

Each structural rule takes the corresponding submodel and all response maps created by its filters. The convolution step created a separate response pyramid for each submodel with a suitable size. All elements are initialized to an offset value we introduced in section 3.1.3. This bias is a value learned during model training that makes the overall score of different submodels comparable against each other.

To each layer of the score pyramid, the response of the submodel's root is added. For the same level level, the part filter responses from on octave up the pyramid are considered for sub-sampling. The parts have a certain position relative to the root, which together with a displacement of the filters due to multiple padding steps results in a sub-sampling region at a level. Within that region, the original algorithm takes every second value in x and y direction to sample a new map, thereby discarding 75% of the original response. The sampled map is then added (with a the appropriate offset) to the transferred root filter response one octave down.

The application of structural rules finally gives us one score pyramid for each submodel. In the next step, all these pyramids are combined to form an overall score from which we can derive the position and also the size of an object, given some additional information like the scale of each level and interim results.

Overall Score and Object Location

To find the position and dimension of an object instance in an image, the scores of all submodels are combined to one single score pyramid. This is fairly simple: by scanning over every level of every submodel score pyramid, the original implementation takes the highest scores and copies them to a separate score pyramid.

This is an important stage in the detection algorithm. We now have one pyramid with detection scores of the whole model for different object sizes. We can now scan this pyramid and save only high scoring results. A high scoring

result is defined to have a score higher than a certain threshold. All high scoring results are written to a special array together with the number of the level they were found in and the position within that level. Position and level number directly lead to the position of the object in the original image, given the scale of the level. However, if we define the position of an object by its bounding box, we only obtain the top left corner of that bounding box with the available information.

Felzenszwalb et al. follow their object detection grammar approach in order to derive the size of the bounding box. The separate C-function 'getdetections.cc' performs a backward trace for each detection. As they have all interim results available, they can look up which model computed a certain score. Taking the size of the root filter in HOG-cells, the scale of a level with a high detection and the knowledge that each HOG-cell originally corresponds to a rectangular 8×8 pixel region into account, it is possible to rescale the size of the root filter with respect to the size of the original image. The resulting rectangle, together with the top left position, equals the sought bounding box.

At this point, the objective of detecting object instances within an image could be seen as achieved. In order to improve their detection results, Felzenszwalb et al. proposed a bounding box prediction step, for which they need the locations of the parts for a successfully determined bounding box. The highest scoring location of a part with respect to a high scoring detection and its size can again be determined by using interim results from the distance transform, namely the envelope and transformed response maps.

The final structure returned by the object detector is a sorted array with a row for each detection. In each row, we find the bounding boxes for all parts and roots as well as the overall score. A bounding box is defined through 4 coordinates, two for the top-left and two for the bottom-right corner of the box.

The final step is post processing to improve the detections though non-maximum-suppression in order to suppress double detections of object instances. All detection are sorted by their score. Detections that overlap with more than 50% with a higher scoring box are discarded. The implemented overlap criterion divides the intersection of both boxes through the size of the box with the minor score. If the coefficient is greater than 0.5, the box is suppressed.

Chapter 4

Implementation

In the previous chapter, we discussed discriminatively trained, deformable part models for object detection as well as the implementation of the detection system published by the creators. This chapter follows the structure of section 3.2.3 by describing our own implementation of the system. Most of the code does not have to be reinvented, of course, which is why we will focus on optimizing particularly time complex parts of the code. We will disclose a general implementation strategy and background on used technology in the first section and explain our own implementation in the following sections. We discuss the efficiency of our implementation in the next chapter.

4.1 Implementation Strategy

To realize a fast version of the algorithm, we set two main objectives:

1. Memory footprint minimization. This includes avoiding unnecessary allocation/deallocation as well as keeping data in memory longer than necessary. The general approach is to try to process data as fast as possible and discard it if no longer needed. The smaller the memory footprint, the higher the chances of fitting the problem into the cache of the processor. Less cache misses result in faster execution.

2. Try to find faster implementations for time critical code. The convolution of filters is the slowest part of the algorithm. We introduced algorithms for speeding up convolution in section 2.2 and will try to apply them to our problem.

As we would like to create a very clean implementation, we strip the original code from all unnecessary functionality, too. This includes not implementing the first implementation of object detection grammars in code version 4.1 and the star-cascade optimization which is included (but not activated) in version 5.

The original implementation is a mixture of Matlab and C code. We choose C++ as the implementation language. The interface for I/O of images and data-

management will be provided by the Open Computer Vision library (OpenCV 4.1). This interface offers functionality for reading and writing images and drawing bounding boxes. In order to exploit all resources, we use the Intel Integrated Performance Primitives (IPP 7.1), which offer highly optimized image processing and linear filtering algorithms for Intel CPUs.

4.2 Model Casting and Datatypes

The first thing that comes to mind when implementing Felzenszwalb's approach is that we cannot use the mixture models trained in Matlab for our implementation directly. We did not examine and implement the training code, which is why we are not able to train own models. Hence, we need to transfer the Matlab models to a C++ readable format.

With the structure of models analyzed in section 3.2.2, we created a function that writes all necessary data to external files. These are in detail:

- *model.model*: a simple text file that contains the **name** of the object class the model was trained to detect, the side length of a HOG-cell **sbin**, the size **interval** of an octave in the feature pyramid, the size **maxsize** of a HOG-patch with height and width being the maximum over the side-lengths of all filters (which can be used for creating a generic filter template that covers all filters in the model), the number of filters **numfilters** and the individual **model threshold**. All values are commented.

- *filter_ *.csv, filter_ m_ *.csv*: every filter is saved to its own *.csv file. Also, every model contains a center-mirrored version of each filter, indicated by the initial '_m_' in the file name. The asterisk is a placeholder for the number of the filter. We give a description of the exact filter structure below.

- *model_ anchors.csv*: a '*.csv' file that contains 4 numbers in each line. The first one is the number of the filter, the second and third are anchor values, indicating the initial position of a part relative to its root filter, and the fourth value is an indicator for whether the associated filter is a root or a part.

- *model_ dt.csv*: holds 4 distance transform values of a filter and bias value.

- *model_ submodels.csv*: instead of programming the structure of the model into an object detection grammar, we extracted it from the initial model and the code and derived a single file that contains all necessary information. There is one line for each submodel, and each line holds the bias value for the submodel, followed by the overall number of filters and the id numbers of each filter in the submodel. The filter ids are sorted so that the first value identifies the root and the following ids point to the part filters.

All files are collected in one folder on disk. The loading routine of the model loads all necessary files and performs pre-computations if necessary, e.g. FFTs for filters.

Matlab generally works with matrix structures that have a column-major order, while C++ works on row-major data. Both methods describe how a multidimensional array is saved in linear memory. In Matlab, the entries of a 2-dimensional matrix are addressed from the top to the bottom and then from left to right, i.e. they are picked column-wise. In C++ though, the order is row-wise, i.e. from left to right and then from top to bottom.

The memory model of HOG-filters in Matlab can be geometrically interpreted as a cube, where the front-side is a matrix that holds the first entry of every HOG-feature, and every k-th layer behind it holds all k-th entries of the feature vectors. As a result, if we want to address the 5-th entry of the first feature in the top-left corner of the filter, we have to scan through all first values of every feature in the whole filter before we find the desired value. We find that this kind of memory-organization of filters is not very intuitive.

We define a new memory layout for filters in our implementation. First, we save them in a row-major order. Second, we specify that all values of a feature vector are saved in a consecutive row, much like a 32-dimensional image. An 8×8 filter thereby has the size of $8 \times 8 \cdot 32$ and is also saved like this in a 'filter_*.csv' file. This allows for easy access of a feature, as all values lie directly behind each other in memory.

We define a few general data-types in our program. The first one is obviously a class representing a model. We can load a model by passing the path to a folder holding all model files. A model generally consists of all variables mentioned above, but also has some additionally fields. When loading a model, we compute an ideal padding size out of all filter sizes. This padding size is later used to generate response maps with an ideal size, which makes all additional padding operations performed in the original code obsolete. Hence, we minimize the number of unnecessary memory allocations by incorporating them into a computation that has to be done anyways.

The smallest data-type is a Map_32f, a very simple matrix classes that is equally used for feature-map and response-array representation. It has a height, width and number of channels and holds floating point data as entries.

The Filter class has a Map_32f member saving all filter values, and additional members for filter id, submodel affiliation and offset, filter size in blocks, anchor position and deformation costs.

The structures FFT_Framework and FFT_Framework_ Overlap_Add hold all informations necessary to compute FFTs of filters and blocks. If specified in the code, the FFTs of filters are precomputed when the model is loaded.

4.3 Implementation in Detail

We will discuss a few implementation details in the following section. As we did not reinvent most parts of the code, we focus especially on the aspects were we tried to improve the existing implementation.

4.3.1 Feature Pyramid

The first big change is implemented in feature pyramid computation. We stated that we want to keep the memory footprint as small as possible. We do not have to compute the whole feature pyramid in advance. Equation 3.12 tells us exactly how many levels a pyramid has. Also, we can compute how the size of a pyramid level develops through the whole computation as multiple padding steps are applied to interim results. This allows for creating a score-pyramid for every submodel in advance, create just one level of the feature pyramid at a time and process this level to the point where the transformed filter responses are added onto the corresponding score pyramids. The pyramid level has then fulfilled its purpose and can be deallocated again.

The score-pyramids for each submodel are the smallest units we have to keep in memory all the time. We can derive the locations and sizes of object instances from them. After computing the height of the pyramids and the optimal size of each level, we generate them with every value already initialized to the correct bias of the submodel.

We then iteratively generate a level in the feature pyramid by subsampling the image and computing the features by taking the same approach as explained in section 3.2.3. We experimented with different methods of interpolation used for resizing the image and found that the original implementation correctly rounds the size of the image, while some common implementations of resize functions, e.g. in the OpenCV library, always floor the size the of the scaled image. An image with a height of 91 pixels, which is rescaled with a factor of 0.5, has a height of 46 pixels versus 45 pixels afterwards, depending on the method of handling floating-point numbers. We implemented both versions and compare different rescaling techniques in chapter 5.

The second step after rescaling a level is feature computation. We described how this is done in the original code, and as this method is pretty straight forward, we simply adopt it in our code with some slight changes. First, the original code takes an image of type double as an input and also returns a double feature map. We change input and output to adapt to floating point images. Also, in order to avoid padding in a later step, we pad each map when allocating memory for it with $model.maxsize + 1$ on each side of the map and also set the truncation feature in the padded area. All other entries are set to 0. The features are then filled in the map with the proper offset.

The computation of a HOG-feature finds the optimal bin for each gradient by computing the dot-product between the gradient and a set of normalized orientation vectors. This is a brute force approach which can be prevented under certain conditions. If the input of the function is neither a double nor a float, but a uchar image, then we can compute the bin and magnitude of every combination of uchar gradients in advance and save them in a lookup table. This leads to two 511×511 sized accumulators, one for the bins and one for the magnitudes. The advantage of this is that due to the limited number of values the uchar data-type offers, we can look up the bin and magnitude of every gradient instead of trying every possible combination for each gradient again and again. This gives us a small, but noticeable speed up when computing multiple feature maps. Our implementation handles a uchar image through a lookup table and a floating-point input with the traditional implementation. Both generate ideally padded feature map of type float. Every feature map is then passed on to the response computation stage and deallocated afterwards.

4.3.2 Feature Map Convolution

We implemented multiple forms of convolution which we all discuss in this section.

The first convolution is the most straight forward one. It performs a block-wise convolution as introduced in section 2.2.2. Block-wise means that it takes every filter, looks for the perfect feature-vector overlay of the filter with a subwindow and computes a response map for all positions with a **complete overlap** of filter and subwindow. This version is implemented with only a few lines of code, is very fast as we can use highly efficient functions from the Intel-IPP library and also, compared to other approaches, uses and compute only necessary values with no overhead.

We tried to implement different variations of this version by hand and without using IPP. None of our alternatives was faster than the first implementation, which is why there are not other straight forward convolutions in the code.

Nevertheless, there are different possibilities of expressing a convolution, which is defined in time domain, in frequency domain, as stated in sections 2.2.3 and 2.2.4. We implemented one version of the convolution theorem for both.

The first implementation takes a feature map and a filter, pads both with enough zeros to fulfill the condition for a successful convolution in frequency domain using the fast fourier transform algorithm, again from the IPP library. Both map and filter are then transformed, multiplied element-wise in fourier space, and the result is transformed back with an inverse FFT. Unfortunately, this approach has two drawbacks.

First, we have to compute the fourier transform of every filter for each stage of the feature pyramid while a feature map has to be transformed only once for one set of filters. As all filters have to be padded to a power of 2 in each dimension,

there is some space for minimizing the number of transformations for a couple of levels in the pyramid where the overall size of the padded filters does not change. We optimized the algorithm in a way that the FFTs of filters are only computed if the complete padding changes while browsing through the pyramid. This allowed for recomputing the FFTs for all filters only 5 times instead of 45 for a sample image.

The second drawback lies in the nature of the convolution theorem: the complete convolution is computed for the filter and the feature map. We are interested only in the block-wise convolution, so we have to search for the correct convolution values and copy them to our final (much smaller) response map. For the sake of simplicity, we only consider one dimension of filter and feature-map. Let L be the length of the feature map and P the length of the filter. Both are padded to $N = L + P - 1$ and once more in order to set N to a power of 2. The first value of our correct response map is located at position $P - 1$ of the convolution result. The following correct values are now located at each multiple of 32. This is not continued through the whole map, as we come into the padded regions after a few steps. We locate the last correct entry by computing the size of the correct response map in advance and counting the correct number of multiples of 32.

We can easily extend this 1-dimensional explanation to the second dimension. The nature of the convolution theorem always forces us to search for the desired response values and subsample them to an extra response array, which is far from ideal.

We implemented an overlap-add convolution, too. This solution at least does not have the problem of computing the FFTs of filters multiple times. Instead, we set a block-size parameter which gives us the size of the FFT for each filter and each block we want to convolve in frequency domain. Note that the block-size parameter is not the same as the block-size in HOG-filtering. It denotes the size of a portion of the feature map for which we want to compute the convolution theorem. For a fixed block size, we can now compute the FFTs of all (ideally padded) filters once, as the size of the FFT does not change over the execution time of the algorithm. A feature map is cut into non-overlapping blocks, each block is padded to the same size of the filters and the convolution theorem can be applied. After the inverse FFT of the block response, the overlapping areas are added together. The resulting overall map suffers from the same disadvantage as the first FFT implementation: we have to search and subsample the desired true response values. We will compare all three implementations in the last chapter.

After computing the response between one level of the feature pyramid and all filters, the original code again pads all responses to the same size. This allows for applying the distance transform, but more importantly to subsample the results correctly. Padding requires additional allocation and copying of data, which we like to avoid. We pad the size of every response map before the first allocation

and set all values in the padded region to $-\infty$. We found that the padding size in this step is directly influenced by the sizes of all filters. While we load a model, we therefore compute an ideal padding size from the filter sizes, and apply this optimal padding at this point in computation.

We can now discard the resized image and the corresponding feature map as they served their purpose. The next step transforms the responses and adds them to the score pyramids.

4.3.3 Distance Transform and Model Structure

At this point in implementation, we choose to ignore the framework of the original implementation for the first time. Felzenszwalb et al. used the notion of object detection grammars to execute rules and and rearrange symbols. We analyzed to code an came to the conclusion that we can implement the same semantics without relying on the surrounding structure. Hence, we exported structural information about the object to a separate data structure in order to be able reproduce the results of the original code.

Our simplified code now takes the responses of the filters and transforms them using the general distance transform in order to allow for spatial uncertainty of the part filters. We use the original implementation which is already very efficient. It also uses Matlab column-major notation for accessing the elements of the response arrays which we change to row-major addressing. Also, it computes the envelope of the distance transform which we do not need in further computations and thereby exclude from the code. Apart from these small changes, the original implementation is adopted as is.

The transformed response maps are then added to the score pyramids with appropriate shifting and subsampling. This process also closely resembles the original implementation apart from addressing issues which we adapt. Transformed root responses are simply added to the level with the same size in the respective score pyramid, and the part responses are subsampled by discarding every second value from the response map and adding them with an appropriate shift (which depends on how much the map was previously padded) one octave down in the score pyramid.

The responses can be deallocated and removed from memory at this point. After all feature pyramid levels have been convolved with all filters and the filter responses have been transformed and integrated in the score pyramids, the next step joins to pyramids in order to locate high scoring instances and determine their sizes.

4.3.4 Object Location and Post Processing

Finding high scoring locations is as simple as in the original implementation. We first join all submodel score pyramids to an overall score by maximizing the score

over all levels. In addition, we always save the number of the submodel that yielded the highest score in an extra field.

In the next step, we detect high scoring instances by scanning the final score pyramid and saving all scores that are higher than a threshold. Once again, we take a different approach compared to the original code. It uses object detection grammars in this step. The underlying semantics essentially take a high scoring instance and browser through all interim data in order to find out which submodel produced the score. The model-number then identifies a root-filter which size they use to determine the final bounding box.

Our implementation looses the overhead from object detection grammars and focuses on the main task of determining the correct bounding box. With the id of the submodel that produced a highest score, we can compute the box taking into account the size of a HOG-cell and the level of the detection. Note that for detections in the first octave of the pyramid, the cell size was initially halved. Is is very unlikely to detect high scoring instances in this octave though as its main purpose is to provide part filter responses for subsampling to the second octave.

One advantage of saving all intermediate results is the opportunity to look up the highest scoring positions for each part filter of a detection. This is the second part the original code implements in order to apply bounding box prediction for improved results. As we do not save any interim results, we are not able to improve our system with a predictor; a step which we are willing to take in order to minimize the memory footprint.

Non-maximum suppression is a necessary step to delete multiple detections of the same object instance. We implemented the same procedure from the original code with the same overlapping criterion, as described in section 3.2.3.

This completes our implementation. We found that it detects objects well with a noticeable increase in speed for some configurations. The next chapter will extensively analyze runtime and memory footprints and compare the precision obtained on a big test-set.

4.4 Code Usage

The code appended to this work simply takes an image, loads a model and calls the 'process' method. It is a bit convoluted in order to allow for using all implemented methods. The 'entry.cpp' contains three flags, USE_FFT_CONVOLUTION, USE_FFT_ OVERLAP_ADD and USE_HOG_LOOKUP, which are self-explanatory. The overlap-add method needs to include both FFT flags.

The code itself needs to be linked against the Intel Integrated Performance Primitives 7.1 for image processing and the OpenCV 4.1 (or later) library in order to work correctly. For testing, we compiled the code with the all optimizations.

Chapter 5

Evaluation

In the previous chapter, we described our implementation of the object detection system of [8]. We tried to minimize the memory footprint wherever possible and realized different optimization approaches. The following chapter analyzes and assesses their quality by comparing memory consumption over time, runtime and average precisions of our code in contrast to the original implementation.

5.1 Test Setup

Although the object detection code is not bound to detect just one object class, we will do the evaluation by using only one casted model for detecting persons in images. The used model was originally learned from the VOC 2010 image database with the version 4.1 training code and is easily translated into the format described in section 4.2. The memory and runtime analysis need a different evaluation approach than computing the average precision of an algorithm.

We will distinguish between three essentially different implementations: the first one uses a normal block-wise convolution for computing the response maps, the two others take advantage of the convolution theorem in order to replace the normal convolution. Together with the original implementation, we have four implementations to compare against each other. We will also discuss smaller optimizations and implementation choices that influence the outcome.

A test image, depicted in figure 5.1, is used as a benchmark for the first two tests. It has a size of 604×403 pixels and shows a person in a more or less "ideal pose" for the detector. We tested the runtime of the code multiple times for each variation of the algorithm and computed the average over all executions. The parameters of the overlap-add procedure influence the memory usage while all other approaches always have the same footprint for a specific image.

In order to compute the average precision of any object detection system, we fall back on the VOC evaluation code [3]. A detector is applied to a database of images, containing all sorts of scenes with different kinds of objects in them. The detector returns possible detections to the evaluating system. The detections are

Figure 5.1: A benchmark image used for memory and runtime footprint determination.

sorted by their score and compared to ground truth bounding boxes. For every recall, the system computes the precision of the detector, successively increasing the recall and measuring the precision every time. This yields a precision-recall curve. The **average precision** is then defined by the integral over this curve.

Our test database contains 5105 different image from the VOC 2010 test set. We compare the average precision of all major implementations and slight variations of them in section 5.4.

All implementations are tested on two setups: a multi-threaded quadcore processor with 2.00 GHz evaluates runtime and memory usage by binding the process of each framework to just one core. To capture an average increase in speed, we use massive parallelization with OpenMP and a second server setup with 32 logical processors at 2.70 GHz and measure the time for the complete database.

5.2 Memory Footprints

Figure 5.2 depicts the memory usage of all major implementations over time. We will discuss every trace in the following.

The original code saves all intermediate results and only deallocates memory when padding a map of any kind. The first graph illustrates this very clearly: The

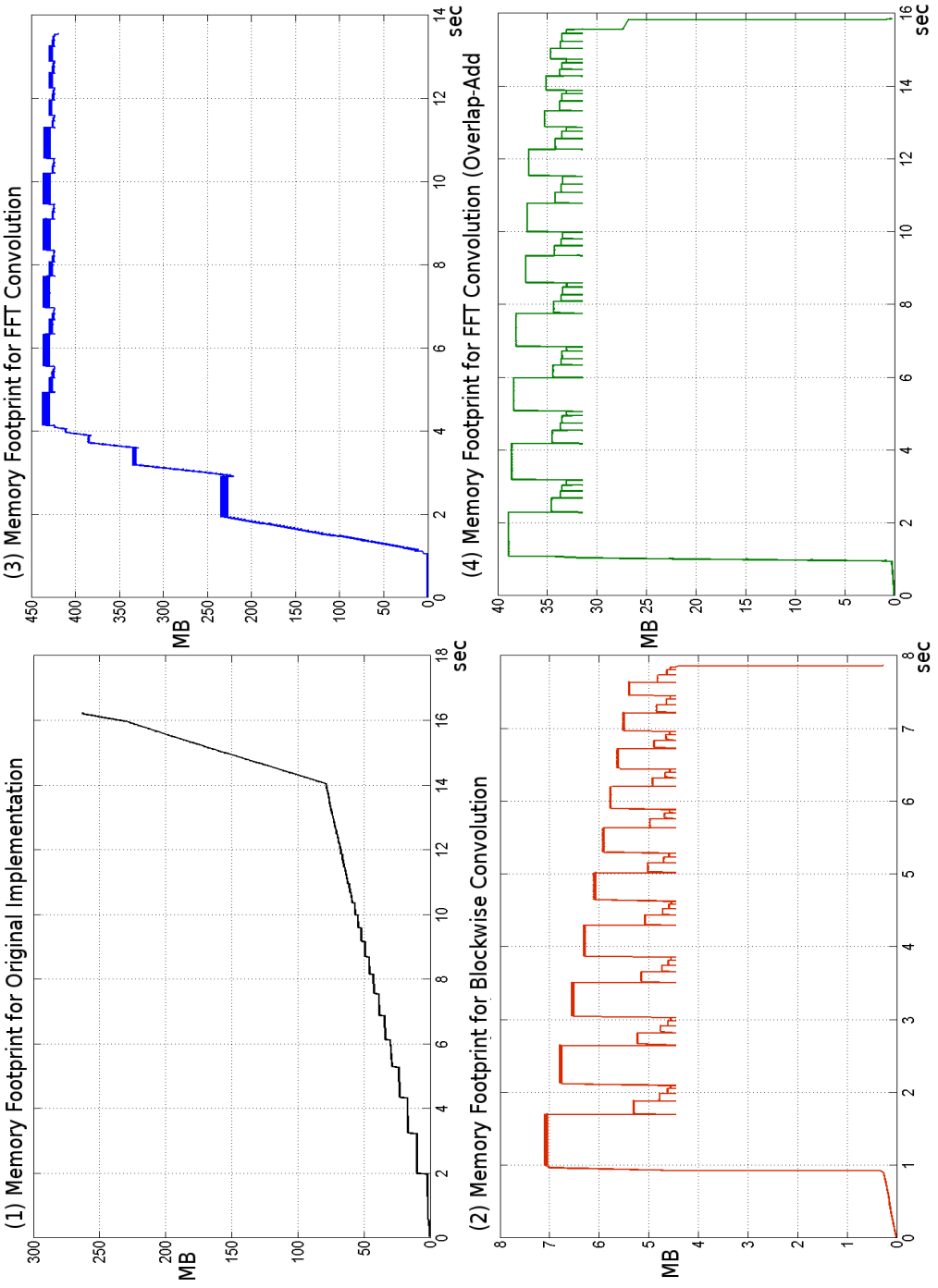

Figure 5.2: Memory footprints for all major implementations. From top down: original implementation, block-wise convolution, FFT convolution and Overlap-Add method.

memory trace steadily increases during the convolution in the first 13 seconds and even more at the end, especially when computing the distance transforms of the

response maps and subsampling the final score maps. This yields a total memory usage of more than 250 MB for our benchmark image.

The trace in the second subfigure paints a different picture. As we payed a lot of attention to minimizing allocations and deallocate memory if not needed any more, we can actually observe a severely smaller memory consumption. The approach of never pre-computing any information and storing it, but instead processing every bit of data as soon as it is available and as further as possible payed off significantly. We can first see that the highest memory consumption for the benchmark is less than 7 MB, only 2.6% of what the original code needs. Additionally, the code is much faster, which is only partially owed to memory management as we will discuss in the next section. We can also observe an interesting recurring pattern in the trace. The number of repetitive increases is directly connected to the number of levels in one octave, which is set to 10 for detecting objects. It is a reflection of the way of implementing the feature pyramid creation and steady processing.

The third trace was measured for the first simple FFT convolution. Although we minimized the costs of filter transformation (see section 4.3.2), the padding necessary for guaranteeing a correct convolution result by usage of an FFT is tremendous. Recall that we have to transform every filter with its additional padding and the feature map to frequency domain. This lets the memory usage explode even above the original implementation. We can observe that the code allocates memory in 5 stages within the first 4 seconds of execution. It then has generated enough filters for all upcoming levels in the pyramid and the memory usage does not grow from this point on. Overall, this approach uses about 430 MB of memory constantly.

The last implementation of the Overlap-Add method was implemented to address this massive memory problem. By specifying a fixed size of the FFT, we have to transform every filter only once when loading a model. The disadvantage of this approach is that instead of transforming a feature map only one time forward and backward, we have to compute multiple FFTs for the pieces of the map instead. The resulting memory trace is very similar to the straight forward implementation of the block-wise convolution in time domain and shows the same repetitive pattern. However, this pattern starts "higher" due to padding again. The transformed filters are much larger that the original ones to guarantee correct results. This approach needs a little less than 40 MB space at its peak. The overall consumption of course can be influenced by the initial size of the FFT which can be chosen freely. We found that making the patches smaller, for instance 32×512, needs only 12 MB of memory. However, as the number of FFTs increases, so does the computation time. We tried different sizes for an FFT (depicted in figure 5.3) and found that a size of 64×2048 works best in terms of minimal execution time. This setup was also chosen for measuring the last memory trace.

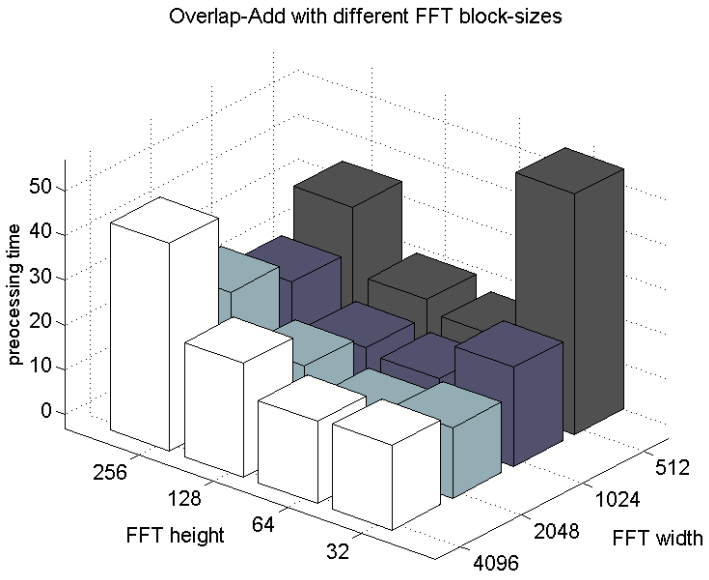

Figure 5.3: Execution times of the overlap-add implementation for different FFT sizes on the benchmark image. The fastest execution can be achieved using a window of 64×2048.

Two of our solutions outperform the original system if we only consider the memory that was used. The initial plan to try to fit the whole problem into the cache of a processor is not achieved. The block-wise convolution version has a very small footprint, but is unfortunately still too big. Due to excessive padding, both other alternatives are definitely not suited.

5.3 Runtime Evaluation

Minimizing the memory footprint is only one way to speed up code. Another option is to find syntactical optimization, i.e. other ways of expressing a problem mathematically. We created three versions of code that use different concepts for reaching the same objective. We will discuss the execution time of all variants in this section. Figure 5.4 depicts the time consumptions of all major steps in each system, which are feature pyramid computation, filtering the scores, performing a distance transform on the responses with additional aggregation of the results in the score pyramids and all steps for locating objects and improving the results. The overall execution time is shown on the right of the figure.

We can determine a few general facts: Post processing has no real impact on the overall time. As we assumed, the major share of execution time is needed for computing the filter responses via convolution. Interestingly, computing the feature pyramid itself is not very expensive compared to that. The distance transform though takes another significant part of time.

We found that we can compute the feature pyramid twice as fast as the original

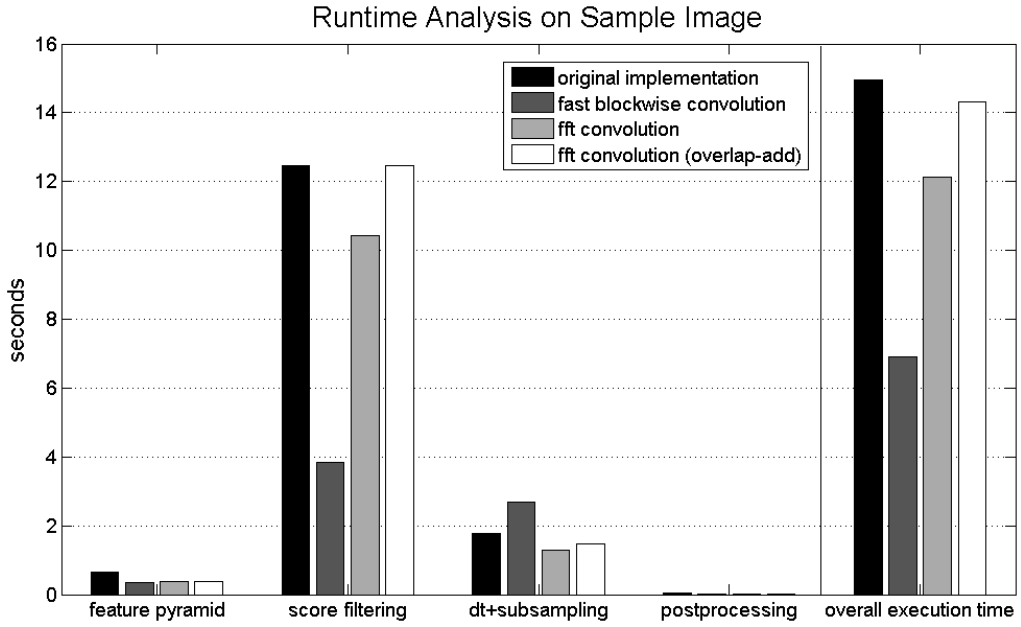

Figure 5.4: Time consumption of major steps in all implemented versions and overall execution time.

implementation, mainly because we avoid the extra padding. If we use a lookup table for determining the best orientation and magnitude of each gradient, we can be nearly three times faster (about 0.24 seconds for the whole pyramid). For just one image this might seem insignificant. For a big database though, we can see a clear improvement in execution time.

Scoring a model is very expensive compared to feature pyramid computation. Again, we can see a clear improvement for the straight forward block-wise convolution implementation, which is three times faster than the original. We can find three reasons: first, we use a highly optimized function that performs each dot product between a filter and any subwindow. Second, we choose an optimal and intuitive memory model for feature maps and filters which allow fast access. Third, we allocate only one response map with an ideal size which can be used for every filter. The implementation is the fastest of our three versions and also the shortest one.

We hoped to improve the convolution by taking advantage of the convolution theorem with the other two versions. While both are a little bit faster that the original code, they are much slower than the intuitive implementation. We can see the reason for this considering a small example: given an average filter with the size of 6×6 HOG-cells and a feature map computed from the benchmark image at its original size with additional padding for detecting objects at the border of the image, the block-wise convolution needs 7.4 million multiplications in order to

yield a correct response map. The FFT versions only needs 0.54 million complex multiplications in frequency domain plus the effort to perform a forward - and inverse transformation. The FFT used in our implementations has a computational complexity of $O(n\log(n))$ and needs $\frac{n}{2}\log(n)$ complex multiplications as well as $n\log(n)$ complex additions. Taking just the multiplications into account, our example uses 6.9 million complex multiplications (which equals $6.9 \cdot 4 = 27.9$ normal multiplications) for both forward and inverse transformation and therefore much more than we need for the simple convolution. Although the above example is only a very rough estimate, it shows that due to extensive padding necessary to replace the convolution, the FFT versions are in total more expensive.

The result of the convolution theorem contains far too much information which is why we have to subsample our correct response map. The notion of using an FFT to replace a convolution only works if we want to substitute a normal convolution instead of a block-wise one. The complete convolution of the aforementioned filter with the feature map would need 230 million multiplications instead of only 7.4 million. This difference cannot be made up by the transformations. In fact we can see that for this problem, the computation of multiple FFTs is far too slow. As a result, the overlap-add implementation, which performs a lot more transformations for smaller patch sizes, is even slower than the normal convolution theorem version.

The distance transform is not much faster than in the original implementation. We are a little faster in the merging process of the score pyramids due to careful memory management, but we did not change the computation of the distance transform apart from slight adaption. There is some optimization potential, namely Felzenszwalb's star cascade, which we did not implement in this work. One value in figure 5.4 stands out: the distance transform after a normal block-wise convolution takes twice as long as in other computations. We did not find a reason for this in the code, the time measuring points are actually the same as for the FFT convolution. Applying the code to other images besides the benchmark image showed the same variation.

As mentioned before, the post-processing is not very time consuming. It involves searching for maximums within the final score pyramid, computing bounding boxes and excluding double detections.

The overall execution times on the right of figure 5.4 indicate that all our three implementations are faster than the original code, but only the block-wise implementation has a real advantage in terms of computational time. Another reason for being faster in general which we did not explicitly mention is that the whole code is implemented in C++. This means that we have no expensive context changes that arise when calling a C function from Matlab code, and also the code is compiled and executed instead of interpreted which gives us another gain in computational time.

We also made a test-run for both the original implementation and our fastest implementation on the 2 GHz multi-threaded quad-core processor. The test on 5105 images with massive parallelization in Matlab and C++ yields overall execution times of 6.2 and a little under 2 hours, respectively. A big server with 32 virtual processors needs about 15 minutes for our setup, which breaks down to 5 images per second.

5.4 Average Precision

Time and memory footprints only measure quantitative gains between different code versions. Until now, we have no statement about the quality of our detections, compared to the original system. This is why we analyze the average precision as a measure of quality in this section.

We apply a classifier on a test set of images with N different objects depicted in them, e.g. cars, people, sofas, etc. The subset $P \subset N$ describes all object that we are actually trying to detect, e.g. people. The classifier returns a set $F \subset N$ of instances that we believe to be correct detections, but that actually contains only $F_t \subseteq F$ positive instances. The **recall and precision** of our retrieval result are then defined by

$$\text{recall} = \frac{|F_t|}{|P|} \tag{5.1}$$

and

$$\text{precision} = \frac{|F_t|}{|F|}, \tag{5.2}$$

respectively. Recall measures how many of the desired objects in the whole test set we found, while precision indicates how many of our retrieved detections are correct instances. Both measures are correlated, which normally yields a low precision for a high recall and vice versa.

The VOC challenge evaluation code for object detectors computes a recall/-precision curve by sorting all detections by score, taking a range of top scoring instances and computing the precision on this set by comparing them to the ground truth. A detection is considered to be correct, if its bounding box clearly overlaps with the ground truth bounding box. They iteratively increase the number of detections in the set and determine the precision every time. They define the **average precision** to be the integral over the resulting curve. The average precision of a perfect object detector would be 1.0 as it returns all positive instances of an object class in a test set and no false positive detections.

We analyzed the average precision of all aforementioned implementations on a test set of 5105 images, without applying the bounding box prediction in the original implementation, because we cannot apply it to our systems as we discarded the interim results and are not able to retrieve the highest scoring positions of the part filters. The results are depicted in figure 5.5.

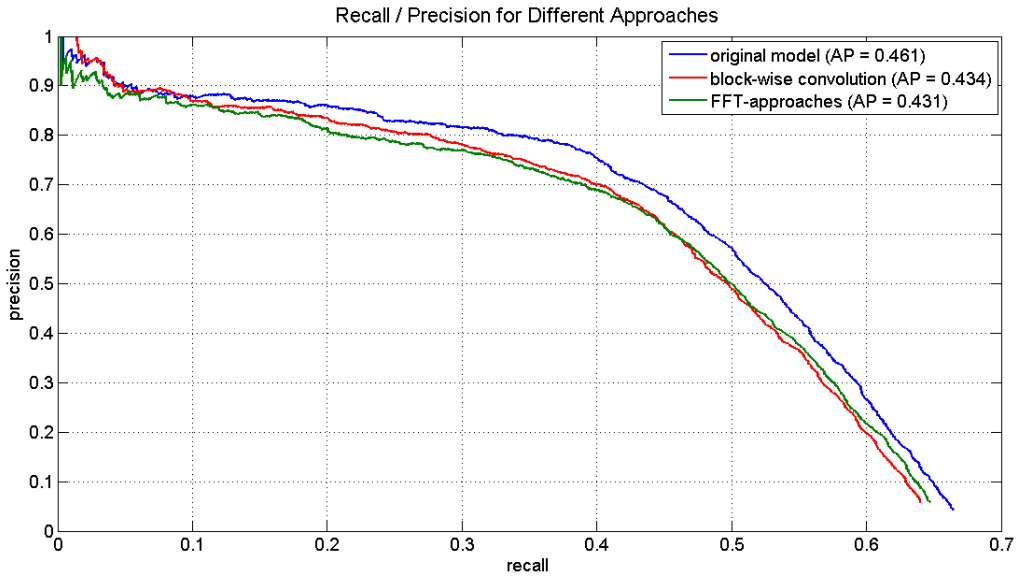

Figure 5.5: Precision and Recall for our implementations and the original code, without bounding box prediction.

The blue trace shows the average precision of the original implementation, the red one depicts the block-wise convolution variant and the green graph combines both FFT approaches as they yield the same average precision. We observe that our models surprisingly perform notably worse compared to the original implementation. The block-wise convolution starts with a better precision for low recalls but slopes for recalls higher than 0.1.

We could not identify just a single cause for these noticeable differences. The most obvious one is that we implemented nearly all computations with single-precision floating point variables, in contrast to Matlab which uses double precision. The linear structure of the general object detection algorithm supports error propagation, hence small numeric variations between our code and the original implementation within the first steps of the algorithm can lead to noticeable big differences in the final results. Small deviations from the ground truth (generated by Felzenszwalb's object detection system) can be amplified during further processing and lead to completely other results. We found that our code detects persons very well, although the final scores a different from those computed with the original code. Using either FFT approach for convolution yields a worse score than achieved with the block-wise convolution in time domain. This is not surprising as the forward- and inverse transformation to frequency domain is not entirely lossless.

While investigating the causes for differences in average precision, we found that using different interpolation methods for rescaling the source image in the feature pyramid step yields significantly worse average precisions. Figure 5.6 de-

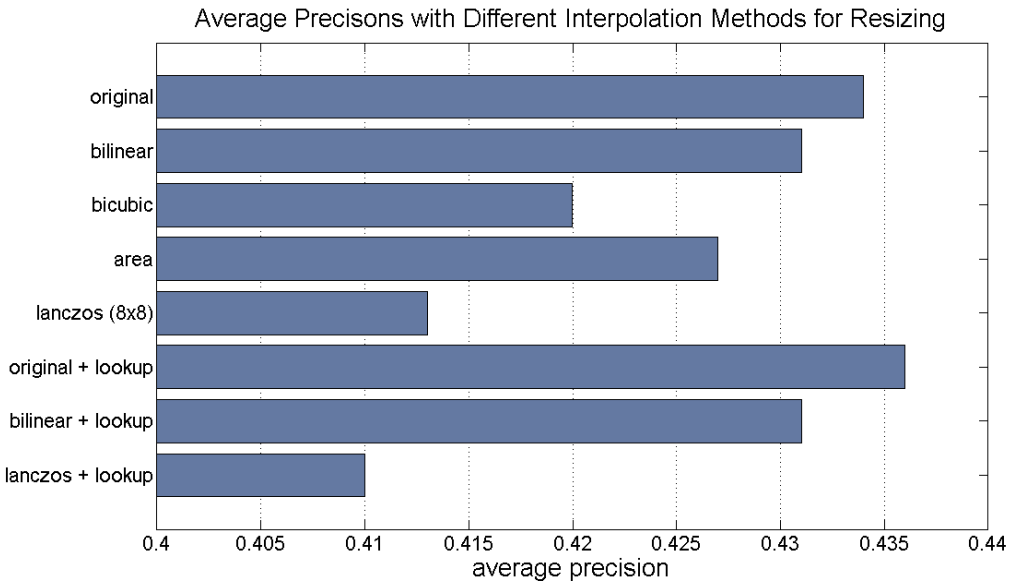

Figure 5.6: Caparison of average precisions for different interpolation methods in feature pyramid computation, with and without using lookup tables for gradient classification.

picts the differences. The interpolation method from the original code yielded the best average precision. We further used the resize function from the OpenCV-framework with different interpolation flags. Surprisingly, using more sophisticated interpolation approaches yields reduces the average precision, e.g. Lanczos interpolation achieves an average precision of 0.412, two percentage points less than our best implementation and 4 less than the original code.

We also investigated the influence of HOG-lookup tables on the average precision. Lookup tables only work with integer number images, so we loose informations during downsizing and rounding the values to whole numbers. We found (figure 5.6) that using lookup tables has an influence, but not always a negative one: the average precision for the original resizing method even improved a little bit. In general, we only see an insignificant disadvantage in using lookup tables for classifying orientation and magnitude of gradient vectors, compared to the slight advantage through saving time.

Overall, we could not identify just one cause for a worser average precision in our models. We found that the approaches in general perform very well on single images. Improving the average precision of our implementation will be one of our main objectives in the future.

5.5 Final Ranking

After comparing the memory footprint, runtime and average precision, we like to do a final ranking for our implementations. Although none of our codes could re-

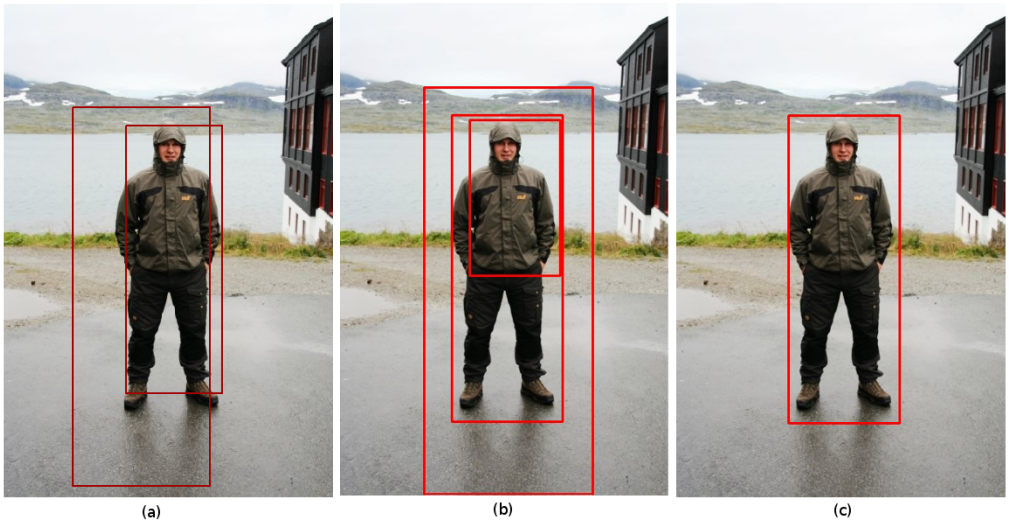

Figure 5.7: Some detections on the benchmark image. (a) original implementation (b) our best system (c) our system with submodel re-ranking

produce the average precision of the original system due to aforementioned causes, the simple convolution shows a very good performance in terms of both memory and time consumption. The notion of fitting the whole computation into the cache of a processor can not be implemented, as 7 MB do not fit into most common caches. The attempts of speeding up the computation using the convolution theorem did not work, although the approaches are in any case faster than the original code. For future work, we will use block-wise convolution technique.

Last but not least, we present a small optimization for person models. We observed that the overlap criterion in non maximum suppression sometimes leaves double detection due to very different sizes of the bounding boxes. We could decrease the allowed overlap in order to delete more boxes, but this could for instance lead to a situation where the smallest box for the upper part of the body is left, leaving half of the body uncovered. Figure 5.7 (b) depicts this situation.

We additionally implemented a procedure that tries to avoid these situations by ranking the detections by submodel id. We use an alternated, restricted overlap criterion to detect two boxes that cover the same object. If one of these boxes comes from a submodel that was trained on the upper part of the model and the other one the whole body, we discard the first one even if it initially had a higher score. Giving certain submodels a higher priority leads to an improvement in bounding box detection for some object instances, e.g. the benchmark image in figure 5.7 (c). Although we obtain a different precision-recall curve, the average precision does not change.

Chapter 6

Conclusion and Future Work

In this thesis, we introduced an implementation of an object detection system based on discriminatively trained, part based models. We described how a model is build from histograms of oriented gradients and applied to feature pyramids. The deformable structure of a model, defined by multiple part filters and ensured via generalized distance transform, allows for detecting objects with a wide variety of appearances in images. We implemented three basic versions of this approach which work with different convolution approaches. We showed that the most straight forward implementation speeds up the computation by a factor of three and is thereby faster than all other implementations. The notion of replacing the convolution of model filters with a feature pyramid with an equivalent approach in frequency domain could be implemented successfully. However, we could not achieve the same speedup as our best implementation due to a significant decrease in operations owed to the definition of block-wise convolution. The gained knowledge about the structure of the approach will be helpful for improving it in the future.

Future work will focus on our best implementation. We found that it does not reach the same average precision as the original implementation and identified a few sources for this discrepancy. Increasing and adapting the average precision will be the main focus of future work. Additionally, we did not exploit the latest technology for fast and parallel computing. Re-implementing the algorithm for execution on a GPU will be another important task. We found that the most expensive part of the approach mainly is owed to a vast number of dot products which can be parallelized on a modern graphics card very efficiently.

We did not put a large focus on speeding up the distance transform. It can be improved by restricting the area in which it is computed for each response map, based on the natural structure of a model.

The step of locating an object and computing its size also leaves room for improvements. We may not be able to use a bounding box predictor due to the lack of necessary interim results, however there might be ways of improving bounding boxes based on multiple detections for one instance.

Last but not least, we can achieve a noticeable speedup with micro-optimizations. We suspect that we can omit a few computations without loosing performance. However, this step needs an extensive performance analysis on a big test set confirm our presumptions.

Bibliography

[1] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, September. [cited at p. 28]

[2] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 1, 6, 9]

[3] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html. [cited at p. 21, 23, 41]

[4] P. Felzenszwalb and D. Huttenlocher. Distance transforms of sampled functions. Technical report, Cornell Computing and Information Science, 2004. [cited at p. 12, 13, 30]

[5] P. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE Computer Society, August 2008. [cited at p. 2]

[6] P. F. Felzenszwalb, R. B. Girshick, and D. McAllester. Discriminatively trained deformable part models, release 4. http://people.cs.uchicago.edu/ pff/latent-release4/. [cited at p. 22]

[7] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Pictorial structures for object recognition. *Int. J. Comput. Vision*, 61(1):55–79, January 2005. [cited at p. 15]

[8] P.F. Felzenszwalb, R.B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9):1627 –1645, 2010. [cited at p. 1, 2, 7, 8, 9, 16, 17, 18, 19, 21, 22, 28, 41]

[9] R.B. Girshick. *From Rigid Templates to Grammars: Object Detection with Structured Models*, 2013. [cited at p. 2]

[10] I. T. Jolliffe. *Principal component analysis*. Springer, New York, 2002. [cited at p. 6]

[11] Richard G. Lyons. *Understanding Digital Signal Processing (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. [cited at p. 10]

Danksagung

I would like to thank Prof. Dr. Rainer Lienhart for enabling me to write this thesis under his supervision and enriching my work with a lot of constructive ideas.

I also would like to express my deep gratitude towards my father, who for the last 5 years not only ensured my financial independence and therefore allowed me to completely focus on my studies, but who always had a sympathetic ear for the problems of a young student and a helpful fatherly advice when needed the most.