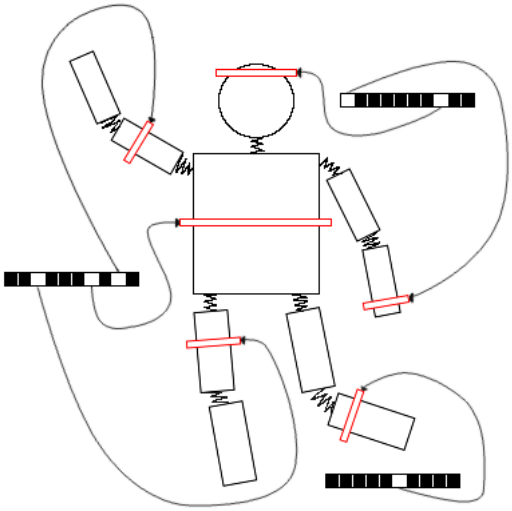
Imperial College London

Department of Computing



Sparse Coding for Object Detection with Deformable Part Models

by Michele Lo Russo

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Andrew Davison

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2014

Abstract

In this report, we describe an integration of sparse coding for Deformable Part Models (DPM), a modern framework for 2D object detection. We developed this system aiming to improve the detection speed of a C implementation of DPM, and analysed the results.

DPM is based on the concept that an object category can be modelled as a composition of parts connected to each other, with a certain degree of displacement allowed (deformation constraints). This method proved to be effective for generalised real-world detection, but is computationally expensive and does not easily achieve real-time performance, especially in multi-class detection contexts.

Using sparse coding allows to exploit the inherent redundancy in models and to reduce computational costs significantly when detecting several object classes simultaneously. This results in a considerable speed gain with only a small accuracy decrease.

We outline the phases of this integration and examine the results obtained in terms of speed and accuracy. We propose, as a future development, a parallelisation of the current implementation, to yield a further speed enhancement towards the real-time detection goal.

Acknowledgements

I would like to thank the following people for their support throughout the project:

- Paul Kelly, whose supervision has created the best atmosphere to work since the very beginning of the project. Bearing with my inexperience, he always explained patiently and carefully things and concepts that were new to me, providing many precious advices. His guidance, organisation and care have been of fundamental importance for the success of this project.

- Zeeshan Zia, for many reasons: first for kindly explaining the finer details of the computer vision parts of the project in the best and most comprehensible way possible. Without his contribution, grasping concepts almost completely new to me in such a short time would not have been possible. He also devised, together with Luigi Nardi, the founding idea of this project in the context of their research group. Finally, he pointed out the relevant literature, tracing a very clear path that aided me in the acquisition of the necessary knowledge to successfully complete the project. His continuous support has really been invaluable.

- Luigi Nardi, for devising the foundation of this project together with Zeeshan Zia, and for his remarkable and passionate help with the finer mathematical details. I also appreciated his appropriate suggestions of reference sources on computer vision, digital image processing and many other things.

- Liu Liu, the author of the library on which this project is based, who cooperated willingly, patiently answering all my questions about the code. Thanks to his suggestions and clarifications, I could understand the code in a very short time and act in the best way possible to keep my implementation tidy and compatible with the original one.

Contents

Chapter 1

Introduction

The Deformable Part Model (DPM) framework is a modern approach used in computer vision for 2D object detection. Developed on the concept of part based models — or pictorial structures, as they were first presented by Fischler and Elschlager in 1973 [18], the key idea behind this approach is that every category, or class, can be depicted, detected and identified as a composition of parts. These parts are allowed a certain degree of displacement from an "usual" position, hence the "deformable" in the name. A practical example of this could be the displacement that arms may have from a standard upright position due to the articulation of the shoulder joint.

Several variations of DPMs have been proposed, since the first by Felzenszwalb et al. in 2008 [17], and they are often compared on two main parameters: accuracy and speed.

This project focuses on a recent DPM implementation based on sparse coding features, as described by Song et al. in their "Sparselet Models" paper [40]. Their implementation, at the moment of this writing, has not been made publicly available, so we developed our own version by integrating the sparse features described in the paper into a C implementation of DPM.

This kind of approach is well suited to a multi-class detection context, because it is centred on an approximation of the features of DPM part filters using a shared dictionary of fixed size. This size is usually considerably smaller than the number of all the part filters from all the different class models, hence the computationally heavy operations (e.g. convolution) are executed fewer times, resulting in a speedup. The approximate result of the original operations can be computed by means of a sparse matrix multiplication with a specific "activation vector" for each part filter.

Since all the most time-consuming operations of DPM can be easily parallelised, they developed a GPU implementation based on CUDA that allows real-time detection of 20 different classes.

1.1 Motivation

Achieving accurate 2D object detection is a persistent challenge in computer vision, and in this context the trade-off between detection accuracy and speed has become crucial. In fact, a real-time and accurate object detection could be of fundamental importance for advancing many AI-related applications, ranging from automatic intrusion detection operated by surveillance cameras to object recognition performed by self-driving cars, or it could aid humanoid robots in better understanding the environment by localising and identifying objects in their sight, and so on.

DPM was developed in view of detecting articulated and deformable object classes, whose appearance can be modelled by appearance and deformation constraints of the parts composing the object. This is obtained through supervised learning with support vector machines (SVM). The purpose is that, after an adequate training performed on a set of annotated data, a DPM detector can automatically determine if a region in an image contains an instance of a class or not (binary classification).

However, the results obtained with DPM detection are still far below human visual performance, in terms of both accuracy and speed. A lot of research is going on to solve object class detection, from both a theoretical and an implementation point of view. In this context, we reviewed the state-of-the-art implementations of DPM, in particular the standard one by Felzenszwalb et al.

[23], developed in MATLAB, and one developed in C and based on a previous version of the standard one [15], included into CCV, a library of computer vision algorithms [30]. Following [40], we share the part appearance matching computation across multiple object classes by sparse coding to make the method scalable across these classes.

1.2 Aims

The aim of this project is to examine the outcome of the integration of sparse coding into a C implementation of the original DPM approach, as in [16].

Throughout the project, we aimed to maintain the highest compatibility possible with the code of CCV original implementation, so that the integration of additional modules would be as painless and unobtrusive as possible for the users of the library.

Furthermore, we evaluate how much a DPM implementation in C can benefit from sparse coding of part filters, in terms of speed while paying attention to accuracy, and to estimate how suitable is this kind of approach for parallel execution.

1.3 Contributions

- In order to define the context of our work, in chapter 2 we provide an introduction to object detection and DPM, outlining the main components of this approach. In chapter 3 we analyse the literature relevant to DPM and describe the datasets used to evaluate DPM detectors (section 3.6).
- In chapter 4, we give an overview of the DPM implementation provided in the CCV library. We provide a detailed analysis of the training (section 4.3) and detection (section 4.4) processes, and we explain the issues that arose from this analysis, and that made the integration of sparse coding modules more challenging (section 4.5).
- We describe the technical details of our sparse coding integration for CCV in chapter 5. The implementation of interfaces for dictionary learning and orthogonal matching pursuit, based on the SPAMS toolbox [35] and used to perform the approximation of part filters, is presented in section 5.2. Section 5.3 describes how the original part responses are reconstructed during the detection phase.
- In section 6.2, we introduce the tools that we used to evaluate our implementation, along with some additional ones that we developed for our evaluation needs. We describe the model set that we used for our tests in section 6.3, specifying the limitations of these models in terms of accuracy, and then we measure to what extent sparse coding of part filters could be beneficial in terms of accuracy (section 6.4) and speed (section 6.5).
- We conclude the discussion of this project with some notes on how the current implementation could be enhanced, and suggest which points could be addressed to make further progress in this research area (chapter 7).

Chapter 2

Background

2.1 Introduction

The purpose of this section is to introduce the background to understand how object detection with DPM works. This will be accomplished through an overview of how object detection is commonly performed (section 2.2), followed by a description of some components that are common to the most recent DPM and object detection implementations (section 2.3). Lastly, an explanation of the usual evaluation metrics is provided (section 2.4).

2.2 Object Detection and DPM

Object detection, a growing domain in computer vision, deals with identification and localisation of instances of object categories in images that can be obtained in real-time from a camera. In this report, we will refer strictly to 2D object detection performed with deformable part models.

Once an object in an image is detected, this recognition can be visualised by surrounding the object with a "bounding box". An example of this concept is shown in figure 2.1.

In order to allow generalisation and invariance to factors like scaling, illumination and shadows, the following operations are performed on the original image:

- First, an image pyramid is created, which means that several resized version of the original image are produced in order to produce invariance to scale. An example of image pyramid is shown in figure 2.2.

- Then, Histogram of Oriented Gradients (HOG) descriptors [6] are computed to store the features of classes in a simplified form which is invariant to lighting conditions (see section 2.3.2). An example of how a person class is stored after applying a HOG filter is shown in figure 2.3.

Therefore, to reduce the impact of the aforementioned factors, we need to produce alternative representations of the original image.

HOG filters are applied at each scale in the feature pyramid, and for each sub-window of the image we want to determine if there is an instance of a given class or not. This is a classification problem which can be solved by learning the appearance of categories with a support vector machine (see section 2.3.3). This involves a training phase and a testing phase:

- In the training phase, images are provided along with annotations of the position of objects in the image. This phase enables learning of the common features and parts appearance of all the examples belonging to a class.

- In the testing phase, new images without annotations are elaborated with the purpose of testing how the generalisation process of the training phase has been effective in representing the common, general appearance of a class and its parts.

Figure 2.1: Example of person detection with bounding box. Left: original image. Right: result image processed with the `voc-release5` detector [23].

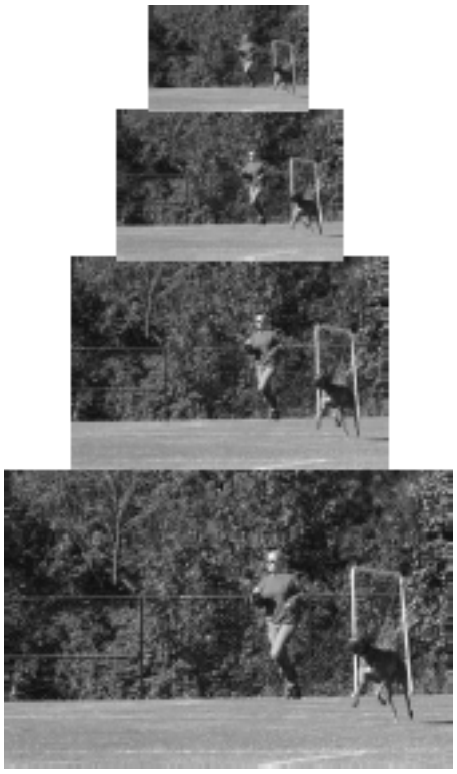


Figure 2.2: Example of image pyramid to allow invariance to scale (taken from [17]).

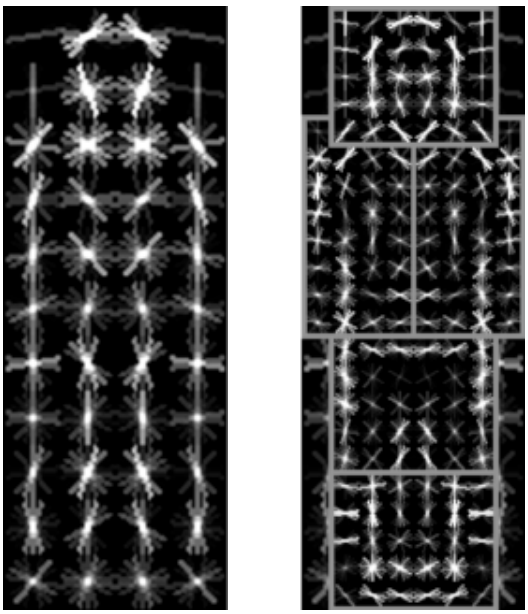


Figure 2.3: HOG descriptors for root and part filters of a person model (taken from [16]).

Therefore, the recognition process takes place in the testing phase and relies on the model of object category learnt during the training phase. To detect objects within an image, sub-windows of each scale in the feature pyramid are convolved with root and part filters in a sliding fashion. This allows collection of root and part filter responses for each model of interest (see section 2.3.1).

In DPM, a certain degree of part displacement from an expected anchor position is allowed. This is determined by applying distance transforms to parts and contributes to the overall score of a detection. Detections whose scores do not exceed a threshold value are accepted, otherwise they are rejected (as further explained in section 3.3).

An outline of this detection process is summarised in diagram form in figure 2.4.

In the next section, the typical components of DPM approach will be analysed in more detail.

2.3 Main Components of DPM

In this section, we give an overview of the main components in DPM approaches. We introduce the concept of root and part filters, which are characteristic of deformable part models, and then describe how these filters are stored using HOG encoding in order to have an efficient and reliable intermediate representation of the object appearance. Finally, we outline how support vector machines are used to train models in a DPM context. HOG-based description and SVM are not used just in DPM methods, but also in several other object detection approaches.

2.3.1 Root and Part Filters

Root and part filters are a core component typical of DPM-style methods:

- A root filter is a representation of the general appearance of the full-size object belonging to a class of interest. This filter is often stored at low resolution, because it aims to depict the most general features of the object, like its shape and aspect ratio.

- Part filters represent the parts in which the object is divided. For example, if we want to detect a person class, we could consider one part filter for the head, one for the torso, one for the lower part of the body, and so on. In DPM the location and size of part filters are estimated automatically when the class models are trained, but their number is not, so it has to be arbitrarily decided ahead of the training. Part filters are stored at a higher resolution than that of the root filter, because in this case we are interested in the finer details that may

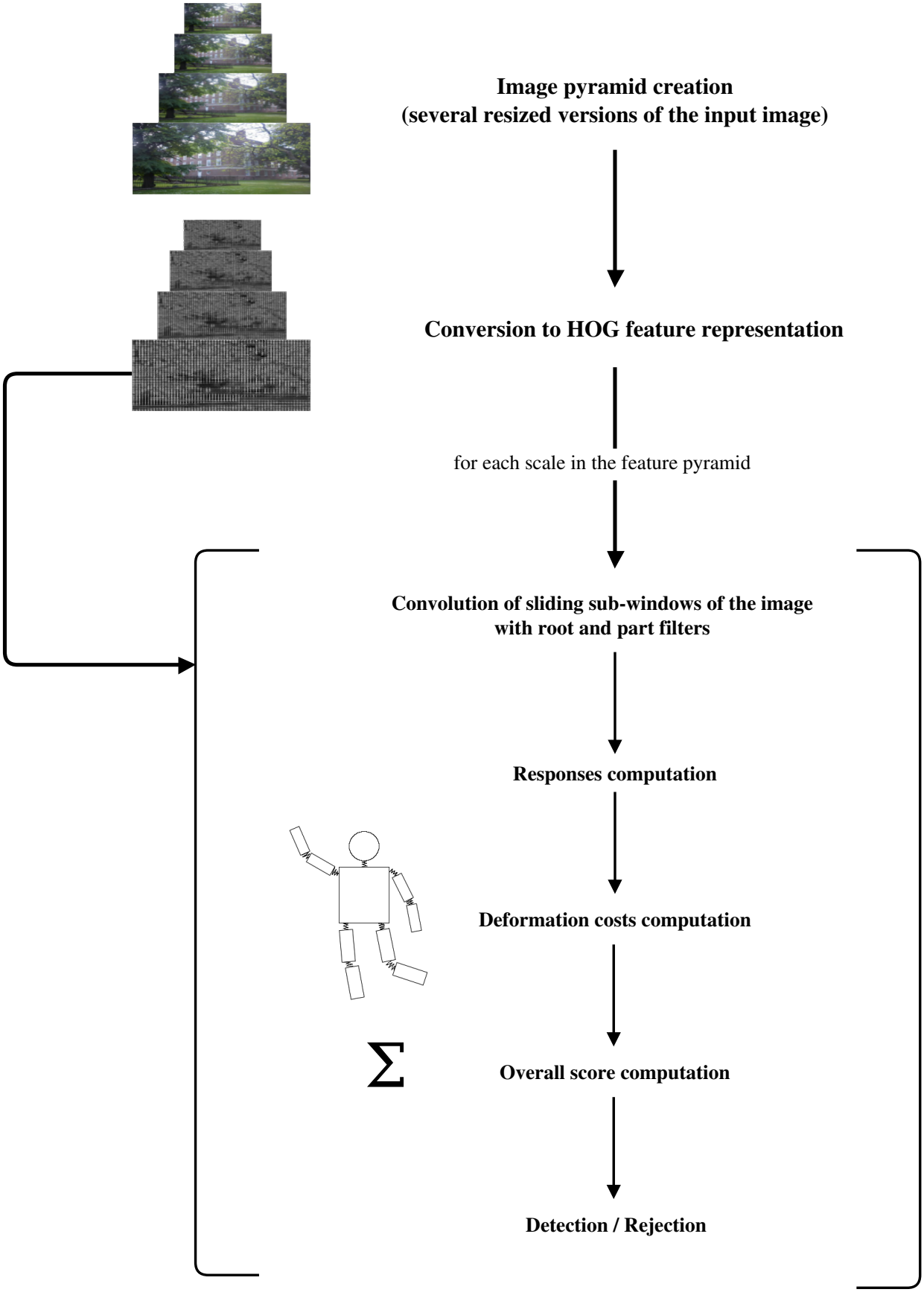


Figure 2.4: Diagram representing the phases of object detection with DPM.

distinguish a certain part from another one, so a better definition of details means higher precision in part classification.

When the detection process is carried out, a score is assigned for each root filter of each class that we want to detect. This happens for each scanned sub-window in the input image, and the higher is the correspondence of features to that of the root filter, the higher will be the score.

Similarly, each sub-window of the input image is matched to part filters too, and a score is assigned for each part filter. The summation of the root filter score and part filters scores constitutes the overall score of a detection. If this score is above a certain threshold, the detection is deemed to be correct, otherwise it is discarded.

2.3.2 HOG Descriptors

Although a large number of works have studied robust descriptors of image patches, Dalal and Triggs (2005) [6] systematically conducted a large study along multiple dimensions, arguing that intensity gradient or directions, histogrammed in the right way, could be characteristic of object appearance and shape.

Their implementation consists of the following steps, applied to the input image:

- application of gamma and colour normalisation filters;
- for every pixel, they compute the gradient with Gaussian smoothing and discrete derivative masks;
- a weighted vote is computed for edge orientation, based on the orientation of the respective gradient, which expresses the gradient magnitude of a pixel;
- votes are accumulated in a variable number of orientation bins, which may be signed or unsigned, namely sensitive to contrast or not.

Their evaluation shows that an increasing number of bins improves performance up to a value of 9, becoming irrelevant with higher values. The sign of contrast proved to be of lesser importance for person detection — which is the application on which their paper focuses — as this information often represents the high variability of background or clothing colours, and thus can be ignored.

In conclusion, they use a HOG descriptor to produce an alternative, compressed version of the original image, in order to store only the information that is pertinent to object recognition.

2.3.3 Support Vector Machines

A support vector machine (SVM) is a machine learning technique used to classify data. It is a typical example of supervised learning, because it uses a set of annotated training data.

As stated in [25], the training data comprises a target value and a series of observable features. SVM is used to build a model based on these features and on the annotations of the training set, which is then able to automatically classify the presence or absence of the target value in some new, unknown and not annotated data (testing set). If we plot the data, with a SVM we can determine a hyperplane, or a set of hyperplanes in the case of multi-dimensional data, that can be used to accomplish the classification task, as shown in figure 2.5.

In the case of object detection, the annotated data consists of the bounding box positions for the objects belonging to a class of interest. This means that we feed a set of images and a set of annotated bounding boxes for each image to train the SVM. This is called "positive training set". In addition to this, we feed a set of images that do not contain objects belonging to the class we want to classify. This is called "negative training set".

Merging the information that SVM can learn from positive and negative sets, when the training is completed a weight value is given to each feature learnt. In the case of HOG-based DPM, the features analysed by SVM are HOG features computed for each pixel of the input image.

The SVM training employed in DPM implementations consists of several iterations to progressively build a better model that increases the chance of getting a correct classification. Additionally,

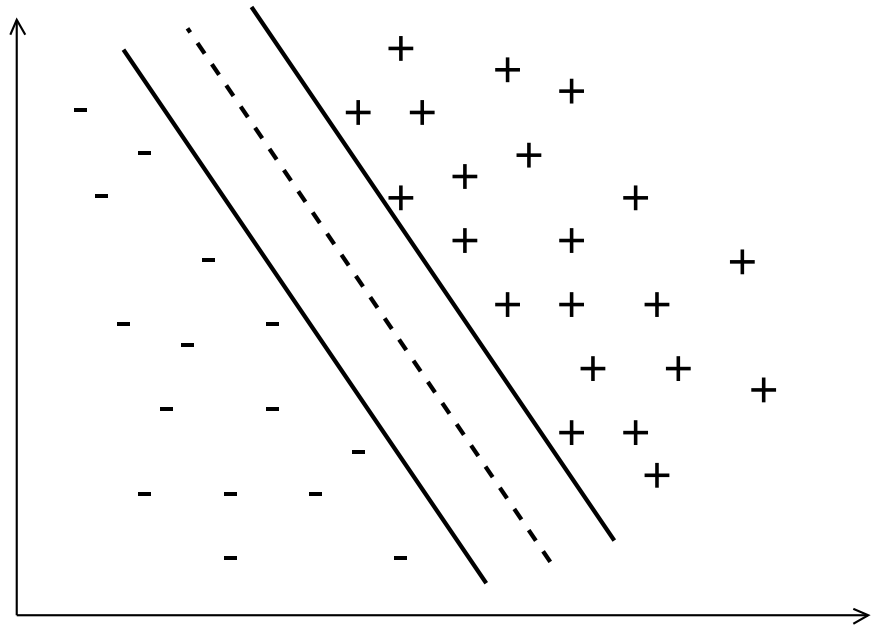


Figure 2.5: Example of SVM classification with positive and negative data plotted and a hyperplane between them.

a common practice to achieve this consists of including hard negative instances in the training data, by "hard negative mining". The set of hard negatives is formed by gathering the negative examples that are mistakenly identified as instances of a class, and this set is used to improve the previous model.

In [6] a linear SVM is used, while Felzenszwalb et al. [16] use a version with latent variables, which they call latent SVM. These latent variables are the positions and sizes of the part filters, which are iteratively learnt by an implicit gradient descent procedure.

2.4 Object Detection Evaluation Metrics

We will now explain the meaning of some terms that are used in the next chapters when referring to the evaluation of object detection methods. All definitions are adapted from [13] and [42].

True Positives. Image windows which have an object and are classified as positives.

False Positives. Image windows which do not have an object and are classified as positives.

False Negatives. Image windows which have an object and are mistakenly classified as negatives.

Precision. Also called positive predictive value, it measures the amount of classified objects that are relevant. This amounts to the fraction of true positives TP over the sum of true positives TP and false positives FP:

$$\text{precision} = \frac{TP}{TP + FP}$$

Recall. Also called true positive rate or hit rate, it measures the amount of relevant objects correctly classified. This amounts to the fraction of true positives TP over the total number of positives P (i.e. the total number of object instances in all the images under consideration):

$$\text{recall} = \frac{TP}{P}$$

Precision-recall curve. The curve derived from plotting precision as a function of the recall. Examples of this are shown in figure 2.6.

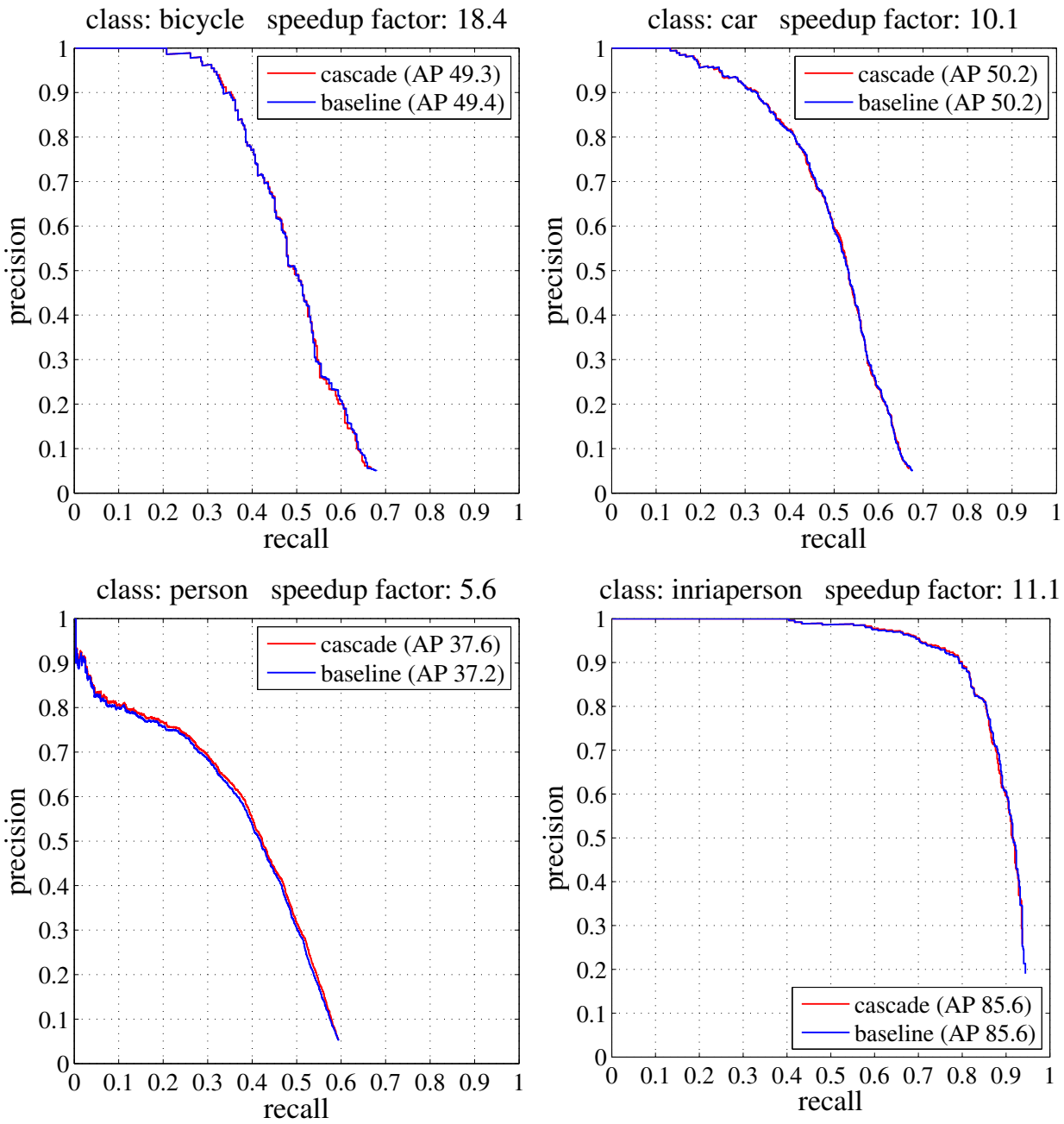


Figure 2.6: Sample precision-recall curves taken from [14], showing how little the average precision changes using the (faster) cascade approach with respect to the baseline DPM implementation of [16].

Average Precision. A measure that takes into account precision and recall simultaneously. In a data sequence like that exemplified by the precision-recall curves of figure 2.6, the average precision is computed by averaging the plotted precision-recall values.

2.5 Summary

We have discussed how object category detection algorithms work in general terms, without going into excessive detail. We describe different variants of DPM in more detail in chapter 3.

We have explained that DPM approaches are particularly successful in 2D object detection because they allow invariance to scale, illumination and variations due to viewpoint, displacement of parts and intra-class differences (e.g. people's clothing or poses).

Finally, we have described the most important components that can be found in a DPM implementation, and given a brief explanation of the commonly used metrics for object detection systems.

Chapter 3

Related Work

3.1 Introduction

A considerable amount of literature has been published on object detection. The purpose of this chapter is to introduce some of the relevant papers related to deformable part models.

First, we present Dalal and Triggs' work [6], which, even if not DPM-based, represented a fundamental milestone towards the development of the subsequent object detectors employing DPM. In fact, the use of HOG descriptors, which is described in their paper for the first time, has become the standard intermediate representation for the input image in DPM detection.

Then, we proceed describing, in chronological order, the characteristics of some DPM implementations that are relevant to this project or that marked significantly the evolution of DPM detection in terms of speed.

Finally, we conclude after presenting the most commonly used datasets in the context of object detection.

3.2 Histogram of Oriented Gradients

Dalal and Triggs introduced HOG descriptors in their 2005 paper [6] and thoroughly explained their potential in the context of detection of people in a general upright pose.

Their research shows that, even if powerful and simple, HOG as a dense image descriptor had been underestimated in those years due to the widespread preference for other feature descriptors, among which:

- Haar wavelets.
- Scale-invariant feature transformation with principal component analysis (PCA-SIFT).
- Shape contexts.

These approaches, commonly used at the time, were implemented and evaluated in the paper in order to make a comparison with HOG.

HOG features are extracted at the sub-window level of the image (cells). For each cell, a histogram of edge orientations is accumulated over the pixels contained in the cell, then combining all the histograms together to get a global representation.

This method allows to get information about shapes efficiently by means of a compressed representation based on a gradient structure. Furthermore, due to the application of photometric normalisation on the image, invariance to lighting conditions is also granted.

The paper evaluated several methods for gradient computation (uncentred $[-1, 1]$, centred $[-1, 0, 1]$, cubic-corrected, Sobel and diagonal masks). The findings show that centred masks yield the best results.

As for the shape of HOG blocks, the following shapes were tested:

- Rectangular, divided into rectangular spatial cells (R-HOG).

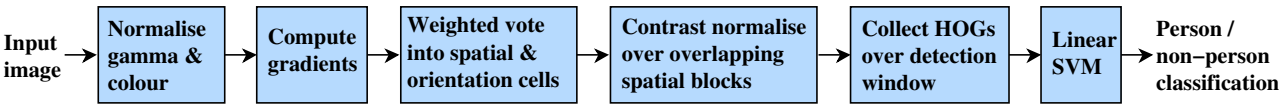


Figure 3.1: Overview of Dalal and Triggs' feature extraction and person detection chain [6].

- Circular, divided into log-polar cells (C-HOG).

For both arrangements of the descriptors, a number of experiments are performed, with respect to parameters like cell and block size, and the corresponding results in terms of performance.

Different schemes for block normalisation are also evaluated: L2-norm, L2-Hys, L1-sqrt, which all yield satisfying results, and L1-norm, which instead resulted in a performance reduction.

To achieve the actual detection of pedestrians, a soft linear SVM classifier is used. However, the classification process is not at the centre of this work, since the main purpose of the paper is to show the good results obtained by the HOG descriptor, that in fact led it to become the new state-of-the-art feature descriptor for object detection. Figure 3.1 represents the full scheme of this pedestrian detection chain.

Since this detector performed flawlessly on the MIT pedestrian database used at the time, a new dataset with pedestrians was produced in the context of this work: the INRIA Person dataset (see section 3.6.1). The purpose of this was to allow a broader and more challenging testing of the method.

In conclusion, this method is based on the innovative use of HOG descriptors in the context of pedestrian detection with a SVM classifier, but still in a "whole-object" fashion, namely without exploiting the fact that an object may be analysed as a composition of smaller parts. In fact, in the conclusion of the paper, the future developments in the field are foreseen, and a part based model is proposed to achieve more accurate detections in wider circumstances.

3.3 Discriminatively Trained Part Based Models

Felzenszwalb et al. [16] developed their first successful and widespread DPM implementation in 2008 [17], that soon became a point of reference in the object detection field and led to several other works based on part models.

First of all, in this work a reformulation of the HOG descriptors elaborated in [6] is used, avoiding redundant information storage through the use of principal component analysis to reduce feature dimensionality without a performance loss.

Secondly, their major contribution consists in engineering the pictorial structure method [18] to make it work well in challenging real-world images. Spring-like connections between pair of parts are used to represent the deformable configuration of objects. An example representation of this kind of pictorial structure can be seen in figure 3.2.

As stated in the introduction of the paper, simpler methods like [6] were preferred at the time because, on difficult datasets, they outperformed DPM approaches. The proposed explanation of this fact is that the training phase with SVM is easier with simpler models than with richer models.

Since this study suggests that a single deformable model is not representative enough of a category, a mixture model approach is proposed to cope with intra-class variations (e.g. different makes of bicycles), variations in viewpoint, and so on. Namely, this allows more than one model component for each object class. A model component is a set of one root filter and several part filters that collect features common to an intra-class variation of a given category.

This method, even if part-based, still requires annotations only for whole-object positions in the training set. In fact, when they train a part based model, they compute automatically and optimally positions and sizes of parts in a latent SVM framework. This computation is based on the number of parts, on the average area of the full-size bounding boxes and on the aspect ratio. In other words, it is necessary to retrieve information that is not explicitly given, namely the position of parts and, optionally, a label for each part. These are the latent (or hidden) variables.

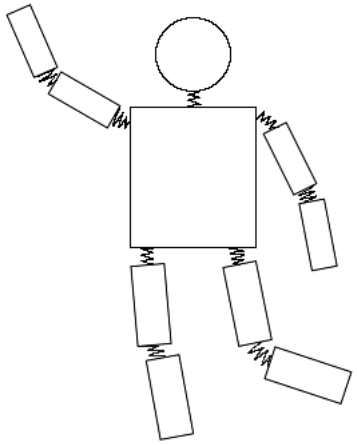


Figure 3.2: Example of pictorial structure with spring-like connections between parts, which allow a limited amount of deformation.

It is argued that providing a more complete labelling for the training phase (i.e. labels for parts) would take additional time and could be biased and suboptimal. Instead, finding effective parts automatically could be the key to increase performance.

In the models of this work, defined as "star-structured", we can identify a root filter — approximately corresponding to the HOG model of [6] — and several part filters. The root one is a coarse, low-resolution filter, and covers the whole object, while part filters are fine-grained and represent smaller components of the object at high resolution. This allows different roles for the two kind of filters: the coarse root filter can be used to retrieve the general, rough traits of an object, like shape and boundaries, whereas finer details for specific parts can be captured with part filters.

In the matching process, the image is convolved with root and part filters in a sliding-window fashion, storing the resulting responses. Subsequently, distance transforms and dynamic programming are used to compute deformation costs and get the best location of parts with respect to the root. The score for each object detection hypothesis is given by root and part filters scores minus the deformation costs, computed on the displacement of parts with respect to their anchor position:

$$\text{score}\left(p_0, \ldots, p_n\right) = \sum_{i=0}^{n} F_i' \cdot \phi\left(H, p_i\right) - \sum_{i=1}^{n} d_i \cdot \phi_d\left(dx_i, dy_i\right) + b \tag{3.1}$$

where p_i is a tuple of the form (x_i, y_i, l_i) specifying the position and level of the i-th filter, F_i' is the filter for the i-th part, and $\phi\left(H, p_i\right)$ is the vector concatenating feature vectors in the sub-window of a HOG feature pyramid H with top-left corner at p_i in row-major order. Also,

$$\left(dx_i, dy_i\right) = \left(x_i, y_i\right) - \left(2(x_0, y_0) + v_i\right) \tag{3.2}$$

represents the displacement of the i-th part from its anchor position, specified by the two-dimensional vector v_i for part i, and

$$\phi_d\left(d_x, d_y\right) = \left(d_x, d_y, d_x^2, d_y^2\right) \tag{3.3}$$

are deformation features. Finally, b is a real-valued bias term introduced to make scores of model components comparable when combined into mixture models.

In the evaluation section of the paper there is an overview of the results obtained by this system with a two-component model for each class of the PASCAL VOC [11] 2006, 2007 and 2008 datasets (section 3.6.2), and for the INRIA Person dataset (section 3.6.1). The average precision (AP) results vary according to the different classes and datasets, going from the 0.6–2.5% of the bird class of the 2007 dataset to the 61–63% of the car class of the 2006 dataset.

In the conclusion of the paper, the potential of information sharing among parts belonging to different classes is foreseen. This constitutes the focal point of this project and is what the Sparselet method [40] (section 3.5) is built on.

The code associated to this work is available online for download under the name `voc-release3` [15], whereas the most recent version to date is `voc-release5` [23].

3.3.1 Cascade Object Detection

Following their previous work, Felzenszwalb et al. [14] realised that the large dimensionality of the detection hypotheses could constitute a performance issue. Therefore, they enhanced the previous DPM implementation by adding cascade classifiers to quickly prune most of the hypotheses without losing detection accuracy. This allowed an increase of detection speed equal to 20 times that of the previous detection system.

In this new algorithm there is a model hierarchy based on the order of parts. The first evaluated are the "weak" models, comprising fewer parts, and then increasingly richer models in the hierarchy. Doing this, hypotheses scoring low with weak models are pruned early, while high-scoring ones are further examined using the next models in the hierarchy — the richer ones.

The partial hypotheses are pruned according to threshold values on their scores. The paper evaluates and defines admissible thresholds as the values under which no hypothesis leading to a correct detection is mistakenly pruned. To obtain safe and effective thresholds, which are called "probably approximately admissible" (PAA), this work exploits statistics of partial hypotheses scores collected during the training phase from positive examples.

This work also uses a reduced dimensional space for HOG features and weight vectors in the part filters. Principal component analysis is done on HOG features from training images to obtain a simplified appearance model by projecting each feature vector into the top k principal components. It is reported that with a value of $k = 5$ this method can evaluate a model six times faster than the original implementation.

General grammar models are also implemented in this work, and used to describe objects in terms of parts. This is exemplified in the paper with a grammar model for the person class. In terms of performance, the star-cascade algorithm generally outperformed the grammar-cascade one by a factor of two, but the latter had a better worst-case runtime than the first. The explanation lies in the use of brute force search in the grammar algorithm, that for a small number of locations is faster than distance transforms.

Finally, this updated approach is evaluated on the 20 classes of the PASCAL VOC 2007 dataset and on the INRIA Person dataset, paying particular attention to the average speedup obtained. The detection speed resulted to be, on average, more than 20 times higher than that of the baseline algorithm, with just a slight loss in detection accuracy on the VOC dataset. On the INRIA dataset, instead, the AP achieved is exactly the same as that of the baseline.

It is remarked, in the last section of the paper, that its main purpose is to contribute to the performance improvement of DPM detectors, and that this implementation could be employed for real-time applications on multi-core processors. In the conclusion it is stated that the performance of DPM based detectors has still substantial room for improvement.

The code associated to this work is included in `voc-release5` [23], available online for download.

3.4 Coarse-to-Fine DPM

This method, presented by Pedersoli et al. [38], can be defined as complementary to the cascade detection algorithm described in section 3.3.1. In fact, the focus is again on the acceleration of object detection with DPM.

The algorithm follows a coarse-to-fine approximation: detection is attempted first on a coarse, low-resolution model. If a positive response is received, further detection tests are performed on gradually higher resolution models and part filters. This allows an exponential reduction of filter evaluations (i.e., the number of times parts are matched to the image is remarkably reduced), which is considered of primary importance for speed enhancement.

Therefore, the object models in this work comprise several multiple-resolution HOG descriptors, organised in a hierarchical way (parent-child) to control the displacement between parts, with deformation constraints also between parts belonging to the same level of the hierarchy (sibling-to-sibling).

In a comparison with the cascade approach, it is argued that this approach does not need adjustment of a threshold value on scores, so with the coarse-to-fine inference it is possible to achieve a significant (10 times) speedup in the training phase too. Moreover, it is noted that the two approaches, exploiting different algorithms to achieve acceleration, could be combined to further improve the speedup.

The paper evaluates this implementation on both the INRIA Person and PASCAL VOC 2007 datasets. On the INRIA dataset, testing different sizes for HOG cells, it is observed that larger cells do not improve AP, but just significantly reduce speed. Therefore, a smaller size for cells is preferable, and it is stated that it already implies consistent deformation for parts owing to multiple resolutions. Then, on the same dataset, the coarse-to-fine inference is compared with the original DPM inference of [17], showing that the loss in terms of AP is minimal if compared to the gain in speed, achievable also in the training phase — unlike the cascade approach.

Finally, on the 20 object classes of the PASCAL dataset, this method proved again not to be particularly detrimental to AP, whereas it dramatically increases detection speed.

The code associated to this work is available online for download at `http://iselab.cvc.uab.es/CoarseToFine` (accessed September 1, 2014).

3.5 Sparselet Models

Song et al. [40] developed this method with the aim of scalable object recognition, i.e. sharing of represented features among multiple object classes instead of having to evaluate each object class detection independently. Additionally, they demonstrated real-time performance with a GPU-optimised implementation.

The main principle of the method is that part filters from different classes can be approximated to build a dictionary of shared filters via sparse coding. Since the size of this dictionary could be set considerably smaller than the number of all the part filters from all the different class models, this approach allows a substantial reduction of computation costs, which results in increased speed. The original values are reconstructed from the approximate ones in the last phases of the detection process, ensuring that the computationally heaviest operations are executed on a smaller number of elements.

A filter in this system is seen as a sparse linear combination of shared dictionary elements, constituting an efficient intermediate representation that can be exploited also to detect novel object classes. This implementation is built on the star-structured DPM from [16], and the paper specifies that, as remarked also in [38], this approach is complementary to the cascade one [14] to reduce filter convolution costs and achieve speedup.

More precisely, the aim is to build a dictionary of filters $D = \{D_1, D_2, \ldots, D_K\}$, called "sparselets" in the paper, that approximates in an optimal way the part filters $P = \{P_1, P_2, \ldots, P_K\}$ gathered from a set of DPM models, subject to a sparsity constraint. This is expressed by the following optimisation problem:

$$\begin{aligned} &\min_{\alpha_{ij}, D_j} \sum_{i=1}^{N} ||\mathrm{vec}\,(P_i) - \sum_{j=1}^{K} \alpha_{ij} D_j||_2^2 \\ &\text{subject to} \quad ||\boldsymbol{\alpha}_i||_0 \leq \varepsilon \quad \forall i = 1, \ldots, N \\ &\qquad\qquad\quad ||D_j||_2 = 1 \quad \forall j = 1, \ldots, K \end{aligned} \tag{3.4}$$

where $P_i \in \mathbb{R}^{h \times h \times l}$ is a part filter, $D_j \in \mathbb{R}^{lh^2}$ is a dictionary element, $\boldsymbol{\alpha}_i \in \mathbb{R}^K$ is an activation vector, ε imposes a cap on the number of activations, h is the filter size (filters are assumed of square

shape for simplicity) and l is the feature dimension. An approximate solution to this problem can be computed using greedy algorithms like orthogonal matching pursuit (OMP), which iteratively estimates optimal matching projections of the input data P on the dictionary D.

The main advantage of this method is that the dictionary filters can be convolved with the sliding sub-windows of the input image and then, since convolution is a linear operation, the responses of the original part filters can be easily reconstructed through sparse matrix multiplication with the activation vector estimated for each part filter. This allows to replace the computationally expensive convolution operation with a much lighter sparse matrix multiplication. The result is approximately equal to the responses that would have been obtained convolving sub-windows of the image with the original part filters:

$$\begin{bmatrix} \text{—} & \Psi * P_1 & \text{—} \\ \text{—} & \Psi * P_2 & \text{—} \\ & \vdots & \\ & \vdots & \\ & \vdots & \\ & \vdots & \\ \text{—} & \Psi * P_N & \text{—} \end{bmatrix} \approx \begin{bmatrix} \text{—} & \boldsymbol{\alpha}_1 & \text{—} \\ \text{—} & \boldsymbol{\alpha}_2 & \text{—} \\ & \vdots & \\ & \vdots & \\ & \vdots & \\ & \vdots & \\ \text{—} & \boldsymbol{\alpha}_N & \text{—} \end{bmatrix} \begin{bmatrix} \text{—} & \Psi * D_1 & \text{—} \\ \text{—} & \Psi * D_2 & \text{—} \\ & \vdots & \\ \text{—} & \Psi * D_K & \text{—} \end{bmatrix} = AM \tag{3.5}$$

where Ψ represents the feature pyramid of an image, $*$ is the convolution operator, A and M are a shorthand for the matrix of activation vectors and the matrix of sparse filters responses, respectively.

Therefore, the score for a detection can be computed according to the following equation:

$$\begin{aligned} \text{score}_{\text{recon}}(\omega) &= m_0(\omega) + \sum_{i=1}^{N} \max_{\delta} s_i(\omega + \delta) - d_i(\delta) \\ \text{where } \; s_i(\omega) &= \sum_{\substack{j=1 \\ \forall \alpha_{ij} \neq 0}}^{K} \alpha_{ij} \left(\Psi * D_j\right)[\omega] \end{aligned} \tag{3.6}$$

where d_i represents quadratic deformation costs, δ the displacement value and ω the current position and scale in the feature pyramid.

Since all the convolution operations $\Psi * D$ are pre-computed, the reconstructed score for a part filter becomes:

$$s_i(\omega) = \sum_{\substack{j=1 \\ \forall \alpha_{ij} \neq 0}}^{K} \alpha_{ij} M_j[\omega] \tag{3.7}$$

The paper states that, with this scheme, around $Klh^2 + N\mathbb{E}[\,||\alpha_i||_0\,]$ operations are required per feature pyramid, while a thorough convolution-based detector would require about Nlh^2 operations. Here K is the size of the sparselets dictionary, N is the total number of part filters, h is the filter size and l is the feature dimension. This shows that the convolution time $\left(Klh^2\right)$ in this system does not depend on the number of part filters (N), which raises if new models are added, but only on the dictionary size, which not necessarily increases with the number of classes.

Therefore, as the number of classes — and then of part filters — increases, a gain in speed can be obtained, equal to the ratio between the complexity of the convolution kernel and the average activation:

$$\frac{lh^2}{\mathbb{E}[\,||\boldsymbol{\alpha}_i||_0\,]}$$

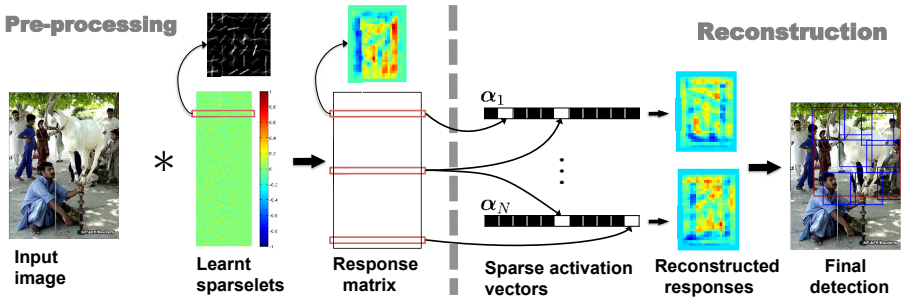


Figure 3.3: Overview diagram of the sparselets method (taken from [40]).

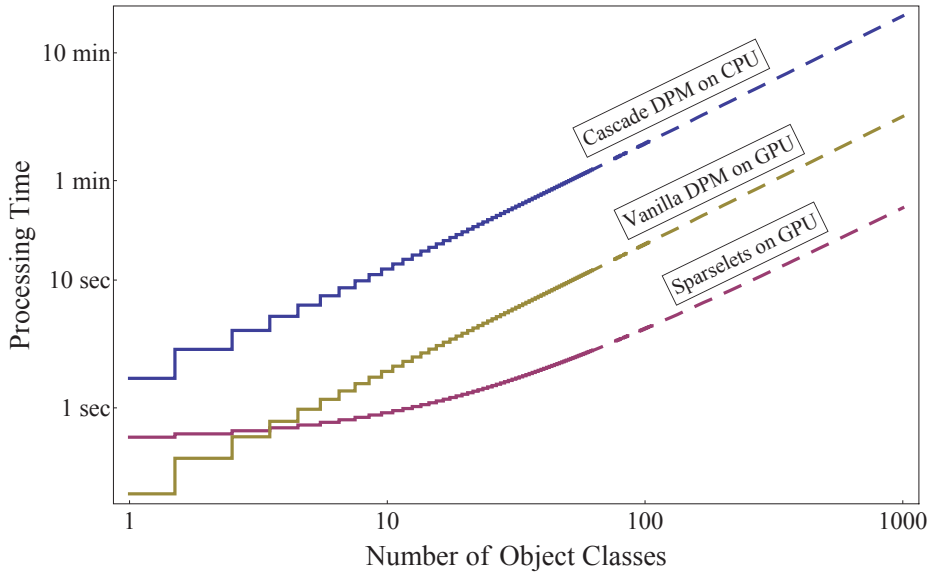


Figure 3.4: Speed comparison of GPU and CPU implementations as a function of the number of classes (taken from [40]).

This means that, in this method, the focal point for speedup is the sparsity in the activation vector. Figure 3.3 shows a diagram summarising this detection algorithm.

For evaluation purposes, the sparselets method is compared in the paper with the cascade one [14], and the two are also interleaved. It was found that a CPU implementation, though, limited the speedup of the method, so it was decided to turn to a GPU CUDA implementation. In fact, the most expensive and time-consuming operations of a DPM approach — HOG computation, convolution and distance transforms — are all parallelisable.

In this work, it was first implemented a vanilla DPM approach on GPU, based on the original DPM approach of [16], and then the sparselets one. A comparison among these two GPU implementations and the CPU cascade one is shown in figure 3.4. The runtime tests are made on a 3.10GHz Intel Core i5 CPU and an NVidia GTX 580 GPU with 512 cores running at 772MHz. The dataset used for these tests is the PASCAL VOC 2007 one, and the dashed portion in the plot is an extrapolation based on a linear model of runtime. The speed observed on the GPU is more than 30 times greater than that of the CPU cascade version, allowing real-time detection.

To determine an appropriate sparselet size, values were tested taking into account two parameters: pre-computation time and representation space. In the GPU experiments the size used was 3×3, while in the CPU ones it was 6×6.

The experiments on the VOC 2007 dataset show that, for well performing models, the response reconstruction error is reasonably small. In general, the sparse reconstruction method maintains the AP with just 20 bases.

Models trained with the ImageNet dataset [7] (section 3.6.3) were also tested on a selected set of object categories taken from the 2011 TRECVID MED challenge [39], which comprises videos of many kinds of events. Due to this high variability, high precision was achieved only at relatively low recall.

Better results were obtained with the dictionary learnt from PASCAL categories and tested on new, unseen categories of the ImageNet set. In this case, with 40 bases, the sparse coding-based method succeeds to maintain the AP, unlike similar approaches (e.g. singular value decomposition).

In conclusion, the findings reported in the paper show that the sparselets approach outperforms other methods with an equivalently compact intermediate representation of parts in terms of accuracy, and all the previous ones in terms of speed.

3.6 Datasets

In this section, we present briefly the datasets used to train, test and evaluate the detection systems described in the previous sections.

3.6.1 INRIA Person

This dataset was created in the context of [6] to allow more challenging people detection experiments in a standardised way. It comprises people in upright position and is divided in two formats:

- Original positive and negative images with annotation files for positives.
- Positive and negative images in normalised 64×128 resolution.

The images derive from a previous dataset, called `GRAZ_01`, and from a digital collection of photographs taken by the authors over a long period of time. These photographs have been cropped to highlight pedestrians, mainly those on the background.

The original images are organised in "Train" and "Test" folders, each with:

- Positive examples.
- Negative examples.
- Annotations for positive examples.

Normalised images are divided in train and test folders too, and they have subfolders for:

- Normalised positive examples centred on the person with their left and right reflections.
- Original negative examples.

3.6.2 PASCAL Visual Object Classes

The PASCAL VOC [11] challenge was held every year from 2005 to 2012 and consisted in evaluating the performance of object recognition systems in several ways. The associated datasets provide:

- A standardised image set with annotated positives to train the detector.
- MATLAB code to evaluate the efficiency of detection methods and compare them.

Being the VOC 2007 dataset the most frequently cited in the papers described in this chapter, we describe it to exemplify the features of a PASCAL VOC dataset. The subsequent datasets have similar features, with an increased number of images and more sophisticated tasks for the challenge (i.e. not pertaining to object detection). In addition to that, the new images in the most recent datasets contain generally harder instances to detect than those of the 2007 dataset, reflecting the increased expectations from advanced object recognition systems.

While VOC 2005 challenge had only four different object classes and 2006 had ten, from 2007 onwards the VOC challenge employed 20 classes, divided in 3 groups (person, animal and vehicle). These were then adopted again in all the next challenges.

The main competitions in 2007 were:

- Classification: predicting the presence or absence of a class instance in a test image.
- Detection: predicting the position (with bounding box) and label of each class instance in a test image.

The set is split in two halves: one for training and the other for testing. Annotation files specifying the positions of bounding boxes and the respective class labels for each object are provided for all the image files in the 2005–2007 datasets, whereas they are provided for the training images only in the 2008–2012 datasets. In total, the 2007 dataset comprises 9,963 images containing 24,640 annotated objects.

Finally, a development kit consisting of training and validation tools is provided, as well as some MATLAB functions that can be used to evaluate the detectors and make comparisons among them.

3.6.3 ImageNet Database

The ImageNet project comprises a rapidly growing image database organised according to the WordNet hierarchy [7]. It is made available as a resource for researchers and educators in the image and vision fields.

According to the statistics published at http://image-net.org/about-stats (accessed September 1, 2014), on April 30, 2010 it counted over 14 millions of images, and over one million of them have bounding box annotations, comprising more than 25 high-level object categories.

All the images are annotated by humans and subject to quality controls.

3.7 Conclusions

From the literature survey carried out in this chapter we can conclude that:

- Since their first use in [6], HOG descriptors have become the standard feature representation for object detection systems based on DPM. In their first implementation, they were still used for a whole-object classification. This produced a good performance on well-trained models, but lacked the flexibility and generalisation attained by the subsequent DPM detectors.
- The classic DPM implementation of [17] introduced pictorial structures with HOG-based appearance models and deformation penalties to enhance models and achieve more generalised detection results. This entails the separation of HOG feature data into root and part filters, organised into multiple-component mixture models to catch intra-class variability. Each component comprises a root filter and a number of part filters, which are convolved with sub-windows of an image yielding a score that, after subtracting deformation costs computed with distance transforms, may or may not lead to a detection.
- The cascade DPM implementation upgrades the classic one by evaluating models that are progressively richer in terms of parts. When the lighter models, with less parts, score low on a region, a detection hypothesis is discarded without convolving models with more parts. This early pruning of low-scoring hypotheses reduces the overall time spent on convolution, thus producing a substantial speedup. A reliable threshold for this pruning operation is computed in the training phase, so that no hypothesis leading to a correct detection is rejected.
- The coarse-to-fine DPM approach implements a different algorithm to achieve speedup, alternative to the cascade one. It evaluates models with progressively higher resolution and hierarchical organisation of part filters (parent-child and sibling-sibling constraints).
- The Sparselets approach is the most recent variant of DPM, and it is particularly efficient for multi-class detection. It achieves real-time performance on GPU with 20 object classes, and its selling point is the approximation of part filters into a shared dictionary of filters. These are considerably fewer than the part filters, so they are employed in the most expensive

Name	Model type	Data organisation	Performance
HOG for Human Detection	Single component	Root (whole-object) filter only	+ accuracy ++ speed – generalisation
Classic DPM	Multiple-component mixture models	Root filter + part filters for each model component	++ accuracy – speed
Cascade DPM	Weak models (fewer parts) evaluated first + richer models	Root filter + part filters for each model component	++ accuracy + speed
Coarse-to-Fine DPM	Low-resolution models evaluated first + higher-resolution models	Root filter + hierarchy of part filters for each model component	++ accuracy ++ speed
Sparselets DPM	Multiple-component mixture models	Root filter + dictionary of sparse filters shared among model components	++ accuracy +++ speed (real-time on GPU)

Table 3.1: Main characteristics of the object detection systems presented in this chapter.

operations (i.e. convolution) in their place, and then the results are reconstructed through sparse matrix multiplication.

- As the object detection systems evolved, the same happened to the datasets used by computer vision researchers to train, test and evaluate their implementations. We have seen that the first datasets were limited to class-specific images (INRIA Person) or a limited number of classes (VOC 2005 and 2006 challenges), and then evolved to a gradually higher number of images and classes (20 in VOC 2007–2012 challenges). The most recent datasets, like ImageNet one, comprise millions of images and several different object categories, providing a massive amount of material for computer vision applications.

The traits of the methods exposed in this chapter are summarised in table 3.1.

Chapter 4

Analysis of the DPM Implementation in CCV

4.1 Introduction

This project integrates the sparse coding approach of [35] into a DPM implementation written in C and included in a library called CCV. In this chapter, we will introduce the details of this original implementation that are necessary to understand the context of our work, and then draw some preliminary considerations that had to be made before getting to the actual implementation.

4.2 CCV: A Modern Computer Vision Library

CCV, a shorthand for C-based/Cached/Core Computer Vision Library, is a project distributed under BSD 3-clause license and freely available for download from [30]. As stated by the author on his website, the idea for this library derives from the dissatisfaction for the OpenCV framework [5]. Therefore, in February 2010 he announced his aim of building a lightweight library entirely based on C functions, with the following characteristics:

- improved speed;
- better memory management through efficient cache mechanism;
- modern algorithms;
- compliance with distributed systems and modern compilers [31].

CCV can be considered alternative to OpenCV in many ways, since it prioritises what OpenCV fails to address [33]. The selling points of this library are:

- built-in cache mechanism to reduce potentially redundant image preprocessing operations;
- cross-platform compatibility;
- application driven philosophy [29];
- efficient C implementation of several state-of-the-art algorithms in computer vision.

The algorithms implemented in CCV are the following:

- Brightness Binary Feature (BBF), a very fast object detection algorithm for rigid objects (e.g. faces, banners, boxes, etc.) originally described in [1] and then subsequently improved in [26].
- Integral Channel Features (ICF), a rigid object detection algorithm more robust than BBF, which employs width-first search (WFS) trees from [26] and was originally conceived in [8] and then improved in [24] and [2].

- Deformable Part Models (DPM), the object detection algorithm of interest in our project, which if compared to the previous ones is slower but more efficient in detecting a wide range of complex objects (i.e. objects that can be simplified into a meaningful set of smaller parts). The DPM implementation in CCV is the original one of [16], and on top of this version we integrated additional modules to produce the sparse representation of filters and response reconstruction of [40]. The details of our implementation are in chapter 5.
- Convolutional Neural Networks (CNN), a deep learning based algorithm for image classification as presented in [28] and with parameters set according to [41].
- Stroke Width Transform (SWT), an algorithm for text detection which aims to recognise features typical of text and identify text regions using geometric signatures. This implementation refers to [10].
- Tracking-Learning-Detection (TLD), an algorithm exposed in [27] which integrates learning and detection for long-term object tracking.
- Scale-Invariant Feature Transform (SIFT), a classical algorithm used for detection of local features, based on [34].

The code is organised in a modular way: the common image processing operations are carried out by efficiently generalised functions, that can be called from the different contexts of the aforementioned algorithms.

4.3 Model Training in CCV: `dpmcreate`

The DPM implementation included in CCV comes with a command-line interface (CLI) to train class models. In this section we will examine how it works, which are the most important functions involved, what they do and how.

4.3.1 The `dpmcreate` Command-Line Interface

This CLI initialises a model training, and has several input parameters, which are summarised in tables 4.1 and 4.2.

An example of command to initialise model training is the following:

```
./dpmcreate --working-dir person_voc07/ --model-component 2 --model-part 8
--positive-list voc07/person.samples --background-list voc07/no-person.samples
--base-dir ../../DPM/VOC2007/JPEGImages/ --negative-count 10000
--negative-cache-size 3000 --data-minings 60 --root-relabels 25 --relabels 15
```

which issues a command to:

- initialise model files in directory `person_voc07/`;
- initialise a model with 2 components, each with 8 parts;
- get a list of positive examples (and their bounding boxes annotations) from file `voc07/person.samples`;
- get a list of negative ("background") examples from file `voc07/no-person.samples`;
- find all the training images in folder `../../DPM/VOC2007/JPEGImages/`;
- initialise a SVM with 10,000 negative examples;
- cache 3,000 negative examples;

Parameter label	Description
`--working-dir`	Specifies the directory to store intermediate files and models.
`--model-component`	Sets the number of components for the mixture model.
`--model-part`	Sets the number of parts per model component.
`--positive-list`	Specifies the path to a file containing a list of images with positive examples and the relative bounding boxes information.
`--background-list`	Specifies the path to a file containing a list of background images, namely without positive examples.
`--negative-count`	Sets the number of negative examples that should be collected to initialise the SVM.

Table 4.1: Table of `dpmcreate` CLI's required parameters.

- perform 60 `data-minings`, which are used to discover hard examples;
- perform 25 `root relabel` operations, which are used to optimise the root filter;
- perform 15 `relabel` operations, which are used to optimise the part filters.

The objective function minimisation of the latent SVM training is done with stochastic gradient descent, for which the following parameters could be passed to the CLI:

- `--iterations`: specifies the maximum number of iterations for a data-mining;
- `--alpha`: starting step-size;
- `--alpha-ratio`: step-size decrease at each iteration;
- `--margin-c`: the constant to control the relative weight of the regularisation term in the objective function [16].

These are optional parameters. A table which explains briefly all the optional parameters is table 4.2.

All the required parameters are collected from the given command, ensuring with `assert` statements that they are all provided. This executable takes care of parsing the positive and negative file lists, and then it delegates the actual model creation to the function `ccv_dpm_mixture_model_new`.

4.3.2 `ccv_dpm_mixture_model_new`

This function takes as input parameters those gathered by the `dpmcreate` tool, and uses them to build class models with stochastic gradient descent. The function declaration in `ccv.h` is the following:

```
void ccv_dpm_mixture_model_new(char** posfiles, ccv_rect_t* bboxes, int posnum,
char** bgfiles, int bgnum, int negnum, const char* dir, ccv_dpm_new_param_t
params);
```

Therefore, it requires the following data:

- a list of images containing positive instances (`posfiles`);
- an array of bounding boxes stored in the `ccv_rect_t` struct (`bboxes`);
- the number of positive examples (`posnum`);
- a list of images without positive instances (`bgfiles`);

Parameter label	**Description**	**Default value**
`--base-dir`	Changes the directory where to look for the images listed in `positive` and `background lists`.	current directory
`--root-relabels`	Sets how many times the relabel procedure should be done to optimise root filters (for multi-component models only).	20
`--relabels`	Sets the number of relabels to optimise part filters.	10
`--data-minings`	Sets the number of data-minings for hard negative examples.	50
`--iterations`	Sets the maximum number of iterations for stochastic gradient descent optimisation.	1,000
`--alpha`	Sets the learning rate value for stochastic gradient descent.	0.01
`--alpha-ratio`	Specifies how much `alpha` decreases at each iteration.	0.995
`--margin-c`	Sets the constant C that controls the relative weight of the regularisation term in latent SVM training.	0.002
`--balance`	Sets a value to balance the weight of positive and negative examples.	1.5
`--symmetric`	Specifies if the object is considered symmetric (1), which means computing only half of the features, or not (0).	1
`--grayscale`	Specifies if to consider the colour information of images (0) or not (1).	0
`--include-overlap`	Specifies the maximum percentage of overlap allowed between an expected bounding box and one deriving from a detection. Above this value, they are considered the same object.	0.7
`--discard-estimating-constant`	Sets a discarding constant, that could be 0 or 1, used when estimating bounding boxes.	1
`--percentile-breakdown`	Specifies a value in range $[0.00 - 1.00]$ that constitutes the percentile interval to compute recall and the relative threshold value.	0.05
`--negative-cache-size`	Specifies the number of negative examples that should be cached. Forced to be > 100 and $<$ `negative-count`.	2,000

Table 4.2: Table of `dpmcreate` CLI's optional parameters.

- the number of background files (`bgnum`);
- the number of negative examples to initialise SVM (`negnum`, passed with `--negative-count` to `dpmcreate`);
- the path to the directory to store intermediate and final models (`dir`, passed with `--working-dir` to `dpmcreate`);
- a `ccv_dpm_new_param_t` struct containing the learning parameters parsed by dpmcreate (`params`).

This function has two library dependencies: `libgsl` [21] and `liblinear` [12]. In order to train models with CCV, it is necessary to compile it with these libraries installed.

4.3.3 Control Flow Overview

The control flow of the model creation function can be summarised in the following steps:

1. Bounding boxes data is processed to compute average area and aspect ratio of positive examples.
2. Root filters are initialised for all components: positive examples are grouped by their aspect ratio, assigned to each model component and then used to compute root filter dimensions.
3. If the model comprises more than one component, the root filter of each component is optimised with a coordinate-descent approach.
4. Part filters are initialised for all components.
5. The loop to optimise root and part filters with stochastic gradient descent starts:

   At the beginning of each `relabel` loop, part responses are collected from positive examples and assigned to the model components.

   At the beginning of each `data-mining` loop, negative examples are collected and assigned to the model components.

   The actual gradient descent iterations are performed to optimise root and part filters using the hard examples collected in the previous step. This is done for each model component. At each iteration, loss values are computed for positive and negative SVM weights. If these values are too small, the `data-mining` will be interrupted before reaching the maximum value specified by the `--iterations` parameter passed to `dpmcreate`.

   At the end of each `data-mining` loop, several values of recall — one for each percentile, according to the value passed to `--percentile-breakdown` — are computed to assess the performance of the model on the training data.
6. Once the optimisation process finishes, the final bounding box sizes and scale are estimated for the root filter of each model component, using the linear regression tools of the GSL library.

Since model training takes a lot of time — in the range of hours, often days —, this function uses a checkpoint mechanism to intermittently save into files the intermediate models, containing also all the information computed at model initialisation (root filter and part filters sizes, etc.), and the progress of stochastic gradient descent.

Hence, almost any step of the aforementioned process, and in particular all the most time-consuming ones, need not be repeated if, for example, the program crashes. The program can be interrupted at any moment and, when run again, if the intermediate files can be found in `--working-dir`, the execution will automatically resume from where it stopped the time before.

The overall model training process is outlined in figure 4.1, which shows only the most important functions involved. These are associated with the corresponding numbers of the control flow list sketched in this section.

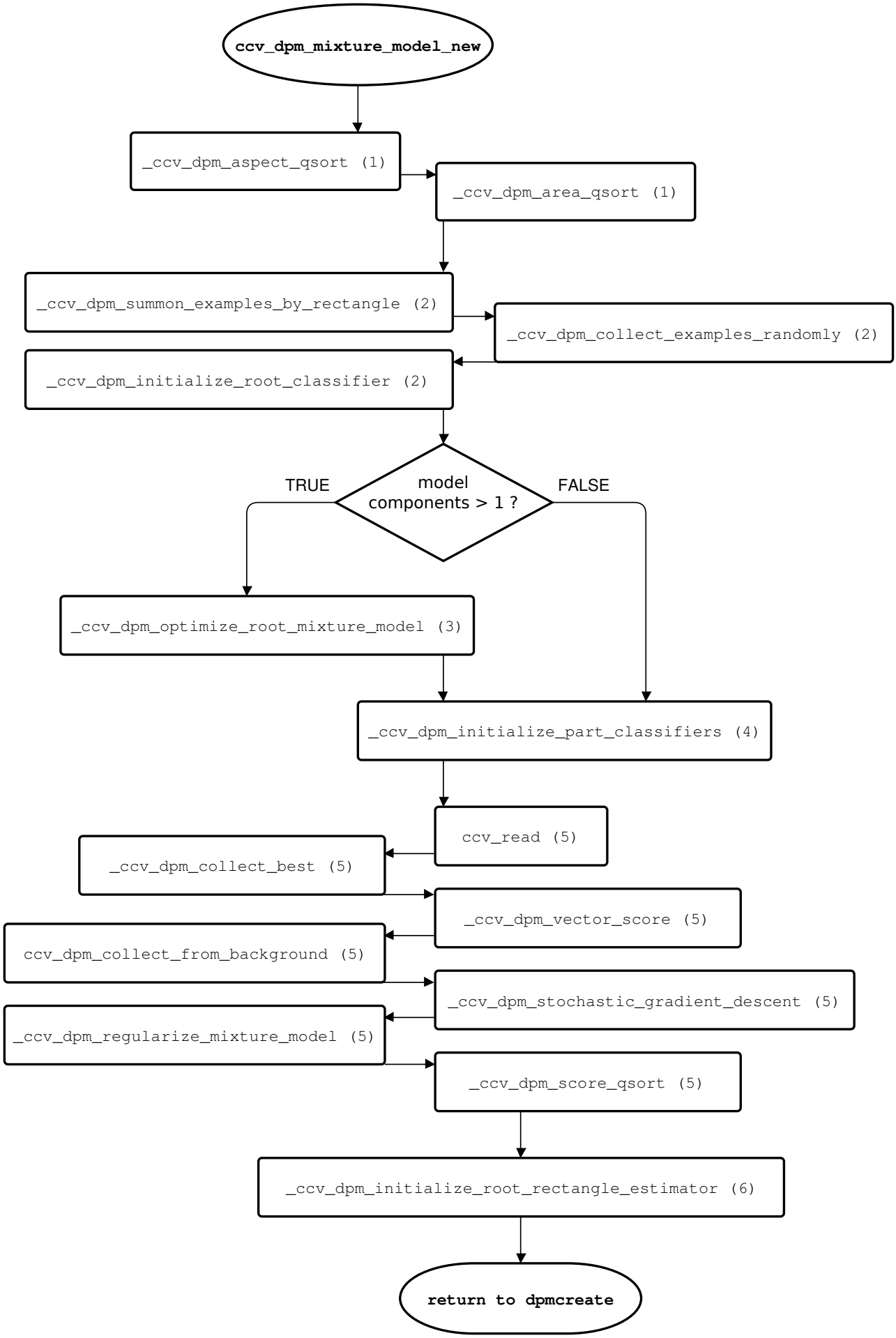


Figure 4.1: Flowchart of the main functions involved in model creation.

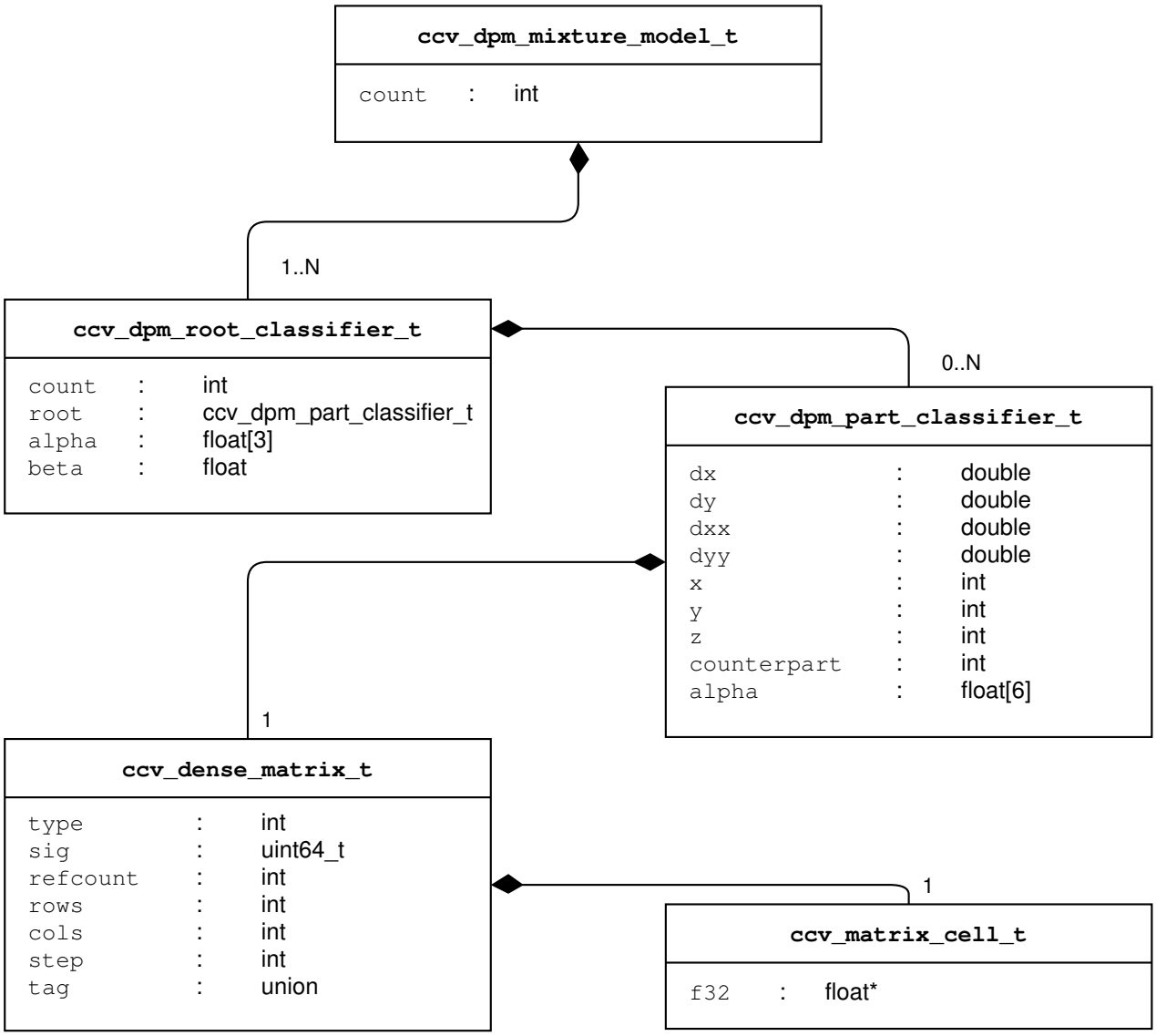


Figure 4.2: UML-like diagram of CCV's DPM structs involved in model training.

4.3.4 Model Format

We will now explain how a DPM model is stored both during and after training. There is a `ccv_dpm_mixture_model_t` struct defined in `ccv.h`, that is used to keep model data during the training process[1]. This is the container of a chain of other structs, as outlined in figure 4.2.

Summarising:

- A mixture model has a `count` and an array of model components (`ccv_dpm_root_classifier_t`).
- A model component contains a `count` and an array of part filters (`ccv_dpm_part_classifier_t`), plus a struct of the same kind for the root filter, parameters for bounding box estimation (`alpha` array) and a bias value for score computation (`beta`).
- A root/part filter struct contains the feature weights data (`w`), stored as a `ccv_dense_matrix_t` array, and for part filters only:
  - deformation features `dx`, `dy`, `dxx`, `dyy`;
  - `x`, `y` and `z` coordinates (`z` is fixed to 1 since we deal with 2D objects);
  - if the object is symmetric, the index of its `counterpart` in the part filters array;

[1]The definitions of structs that are relevant to DPM can be found in appendix A.

Line(s)	Description	Type
1	Model status flag: '.' = final model; ',' = intermediate model.	`char`
2	Number of model components (written twice in intermediate model files).	`int`
3	Size of root filter (*rows cols*).	`int` (×2)
4	Bounding box estimation parameters for root filter.	`float` (×4)
$[5 - R_{end}]$	Feature weights for root filter. $(R_{end} = 4 + rows)$	`float` ($\times(rows \times cols \times 31)$)
R_{end}+1	Number of part filters.	`int`
R_{end}+2	`x y z` coordinates.	`int` (×3)
R_{end}+3	Deformation features `dx`, `dy`, `dxx`, `dyy`.	`double` (×4)
R_{end}+4	Bounding box estimation parameters for part filter.	`float` (×6)
R_{end}+5	Size of part filter (*rows cols*) and `counterpart` (if `symmetric`).	`int` (×3)
$[P_{start} - P_{end}]$	Feature weights for part filter. $\begin{pmatrix} P_{start} = R_{end} + 6 \\ P_{end} = R_{end} + 5 + rows \end{pmatrix}$	`float` ($\times(rows \times cols \times 31)$)
$\cdots$ the scheme is repeated as from line R_{end}+2 for all part filters $\cdots$		
$\cdots$ the scheme is repeated as from line 3 for all model components $\cdots$		

Table 4.3: Data organisation of intermediate and final model files.

  - parameters for bounding box estimation (`alpha` array).

- The size of part filters are stored in the `rows` and `cols` fields of the weights matrix `w`.
- The actual weights are stored as an array of floating numbers inside a `ccv_matrix_cell_t` struct.

This structure can be easily mapped to that of the MATLAB models that can be trained with `voc-release3` [15], whereas it is quite different if compared to that of the `voc-release5` [23] models. In particular, it lacks of class labelling, which could be particularly useful in a multi-class detection context.

Once a `data-mining` finishes, the current intermediate model is saved into a file in `working-dir`, with the following name format:

$$model.r.d$$

where r and d are the numbers of the current `relabel` and `data-mining`, respectively. The file that will store the final model is also updated after each `data-mining`. This is saved simply as *model* in `working-dir`. The intermediate and final model files are normal text files, and they store the data according to the scheme outlined in table 4.3.

4.3.5 Performance Analysis

We have analysed the performance of `dpmcreate` using the GNU gprof profiler. It is important to mention that there are considerable performance differences depending on the convolution routine used. CCV supports two convolution methods:

1. KissFFT [4], a simple but slow implementation, which is shipped with the code and included in the `3rdparty` folder inside `lib`.

2. FFTW3 [20], which is substantially faster but not included with the code. It is an external library and needs to be already installed on the system and detected by the configuration tool.

In the case of `dpmcreate`, however, the impact of the convolution routine used is essentially lesser than that for the detection program, `dpmdetect`. While we will conduct an opportune performance comparison between the two in section 4.4.4, here for conciseness we show only a profile example in which KissFFT is used for convolution, since we have experienced that the results were practically equivalent to that of FFTW3, especially in long-term training.

In order to inspect and present the results of our analysis more handily and efficiently, we have converted the profile data into graphical form using Gprof2Dot [19]. The result of this is shown in figure 4.3.

At a glance, it is apparent the quantity of functions that overcome the percentage threshold to appear in the graph. In fact, all the functions whose total percentage time is below 0.5% are filtered out. Nonetheless, there are a lot of different functions in this graph, especially if compared to the much simpler and linear structure of the detection program (section 4.4.4).

The training in this example was one of medium length — around 6 hours on a 3.40GHz Intel Core i7-3770 CPU — with the iterative parameters (relabels, data-minings and iterations) set to lower values than the default ones. The figures could be considerably different in very short model trainings or very long ones, but we deemed this to be representative of the general distribution of work in `dpmcreate`. Broadly speaking, we can say that the duration of the training is proportional to the number of:

- model components;
- total positive examples;
- total negative examples;
- negative examples cached;
- part relabels;
- data-minings for each part relabel;
- stochastic gradient descent iterations for each data-mining.

We are going to explain which figures will differ, and how much, according to the training duration.

The considerations that we have drawn from the example case of figure 4.3 are the following:

- A third of the execution time is spent on convolution: `ccv_filter`, that in turn calls convolution routines from the KissFFT library (34% of total time). This figure would be higher for a very short training, but a lot smaller for a very long one, to such an extent that it could even disappear from the graph for a training that lasts for days (i.e. its total time would not be over 0.5%). This means that the impact of convolution decreases as the execution time increases.
- Another third of the time is spent on caching operations, e.g. hash generation: `ccv_cache_generate_signature` (33% of total time). This figure tends to be proportional to the total execution time, so it would be even two times that of this example in a long training session.
- Other relevant functions in terms of performance are those for:
  - score computation: `_ccv_dpm_vector_score` (13% of total time);
  - HOG computation: `ccv_hog` (12% of total time).

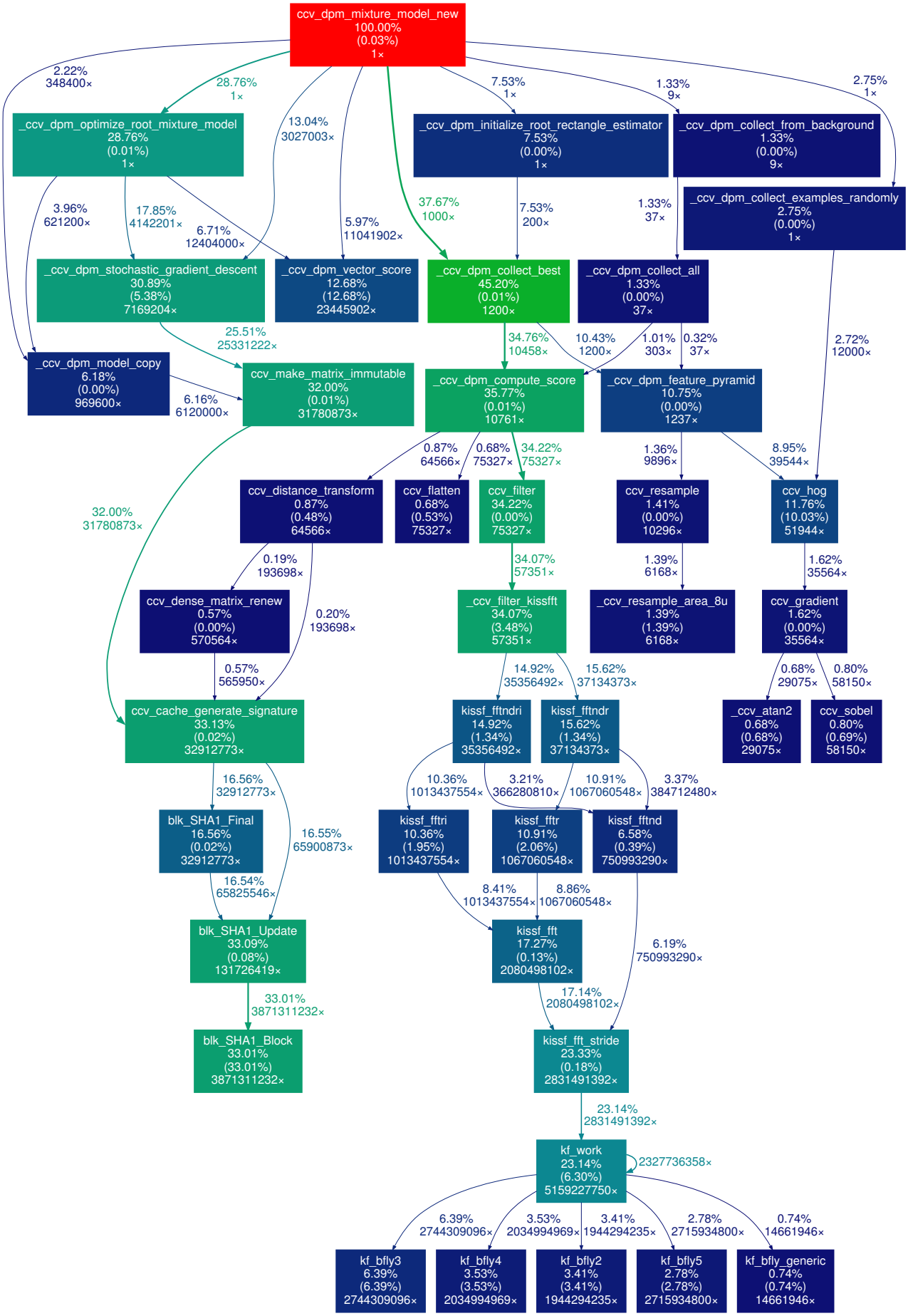


Figure 4.3: DOT graph of `dpmcreate` profile data. A block contains: function name, % of time spent in the function (included children), % of time spent in the function (self only) and total number of calls. Values near arrows between functions: % of time spent and number of calls.

These functions keep their share of execution time almost constant for any duration of the training execution.

A parallelisation of the training process has been attempted, but without success due to time constraints and to the internal complexity of the process itself. In particular, the caching mechanisms of matrices (e.g. those performed by functions like, in the example above, `ccv_make_matrix_immutable` or `ccv_dense_matrix_renew`) proved to be an obstacle to parallelisation. The heavy cache usage, which is a cornerstone of CCV, leads to the conclusion that this library, in cases like that of `dpmcreate`, was not conceived to be easily parallelised.

4.4 Object Detection in CCV: `dpmdetect`

To perform object detection on an image or a set of images, CCV provides the user with another command-line interface, `dpmdetect`. As with `dpmcreate`, this program stands between the user and the internal functions of CCV, and it is of particular relevance for our sparse implementation. In fact, to detect objects with part responses reconstruction from sparse filters, we produced an alternative version of this program, that we called `dpmsparsedetect`, which is broadly moulded on this one, and described in section 5.3.

Therefore, in this section we are going to explain how the object detection process is implemented in CCV.

4.4.1 The `dpmdetect` Command-Line Interface

This CLI is much simpler than the model training one. It takes less parameters and recognises them only by their order, which is therefore essential in this case — unlike in `dpmcreate`. The format of a command-line call execution of dpmdetect is the following:

```
./dpmdetect <image.png> <person.model>
```

or alternatively:

```
./dpmdetect <filelist.txt> <cat.model>
```

In the first case, `dpmdetect` will perform object detection on the file `image.png` using the model `person.model` (the extension `.model` is merely illustrative, there are no constraints on the file name). In the second case, namely the case in which the first parameter will not be recognised as an image file by `dpmdetect`, it will be read as a text file and each line will be interpreted as a path to an image. This allows batch detection of images using, in the example above, the model `cat.model`. Moreover, if an additional parameter is provided after that of the model file, the program will attempt to change the directory to that specified by this parameter. Further parameters are ignored.

The flow of control is likewise simple:

1. The first parameter is parsed with the function `ccv_read`, which builds an image as a `ccv_dense_matrix_t`. If recognised as an image, the result is passed directly to the function `ccv_dpm_detect_objects`. Otherwise, the parameter is read as a text file, and in a loop each line is parsed with `ccv_read` and the result passed to `ccv_dpm_detect_objects`.

2. The second parameter is parsed with the function `ccv_dpm_read_mixture_model`. This checks with an `assert` that the content of the first line is the flag character '.' (format check), and then trustfully reads the data inside it and builds a `ccv_dpm_mixture_model_t` struct, that constitutes another input parameter of `ccv_dpm_detect_objects`. This function is the one in charge of the actual object detection.

However, in the original implementation, the second parameter had to be a single model file. This means that multi-class detection was not enabled. We thought this was a bit too much restrictive for our scopes, and since the code in `ccv_dpm_detect_objects` supported already an array of models as input, we decided to overcome this limitation. We changed the code in order to enable multi-class detection also in the original `dpmdetect` program, so that the following command would be accepted:

```
./dpmdetect <image.png / filelist.txt> <model-list.txt>
```

This was easily done by just following a procedure similar to that explained before about the first argument. We parse the second argument looking for the model flag characteristic of final model files, and:

- If the first line contains '.' only, we assume it to be a single model file, so the procedure is the same as before.
- Otherwise, we assume it to be a list of model files, with the same format as the list of images — one line, one file. Hence, we scan the file and count the number of lines, so that we can allocate a `ccv_dpm_mixture_model_t` array of the appropriate size. Then we scan the file again, passing each model file to `ccv_dpm_read_mixture_model`, thus building the model array in a loop. Finally, this array is passed to `ccv_dpm_detect_objects`, together with the other usual input parameters.

In conclusion, we would like to add a further remark on the excessively limited interface provided for detection. It could be argued that some additional parameters should be settable by the user, like at least the detection threshold (i.e. the value to determine if a detection hypothesis is accepted or not). Instead, this parameter and some relevant others (e.g. the scaling interval for image pyramid creation) are hard-coded inside the `ccv_dpm.c` file, and hence inaccessible to the user without changing and re-compiling the code. This has already been discussed with the author, who agreed on the point, and so it constitutes an important entry in the list of future improvements.

4.4.2 `ccv_dpm_detect_objects`

This function takes as input a target image, an array and number of models and a struct containing detection parameters, and returns an array of bounding boxes. The function declaration, from `ccv.h`, is the following:

```
ccv_array_t* ccv_dpm_detect_objects(ccv_dense_matrix_t* a,
ccv_dpm_mixture_model_t** model, int count, ccv_dpm_param_t params);
```

The `ccv_dpm_param_t` struct `params` is defined in `ccv_dpm.c` with these fields and values:

```
const ccv_dpm_param_t ccv_dpm_default_params = {
    .interval = 8,
    .min_neighbors = 1,
    .flags = 0,
    .threshold = 0.6, // 0.8
};
```

A summary of the control flow is outlined as follows:

1. A HOG feature pyramid of the image is created, at several different scales.
2. The array to store the final bounding boxes is initialised (`result_seq`).

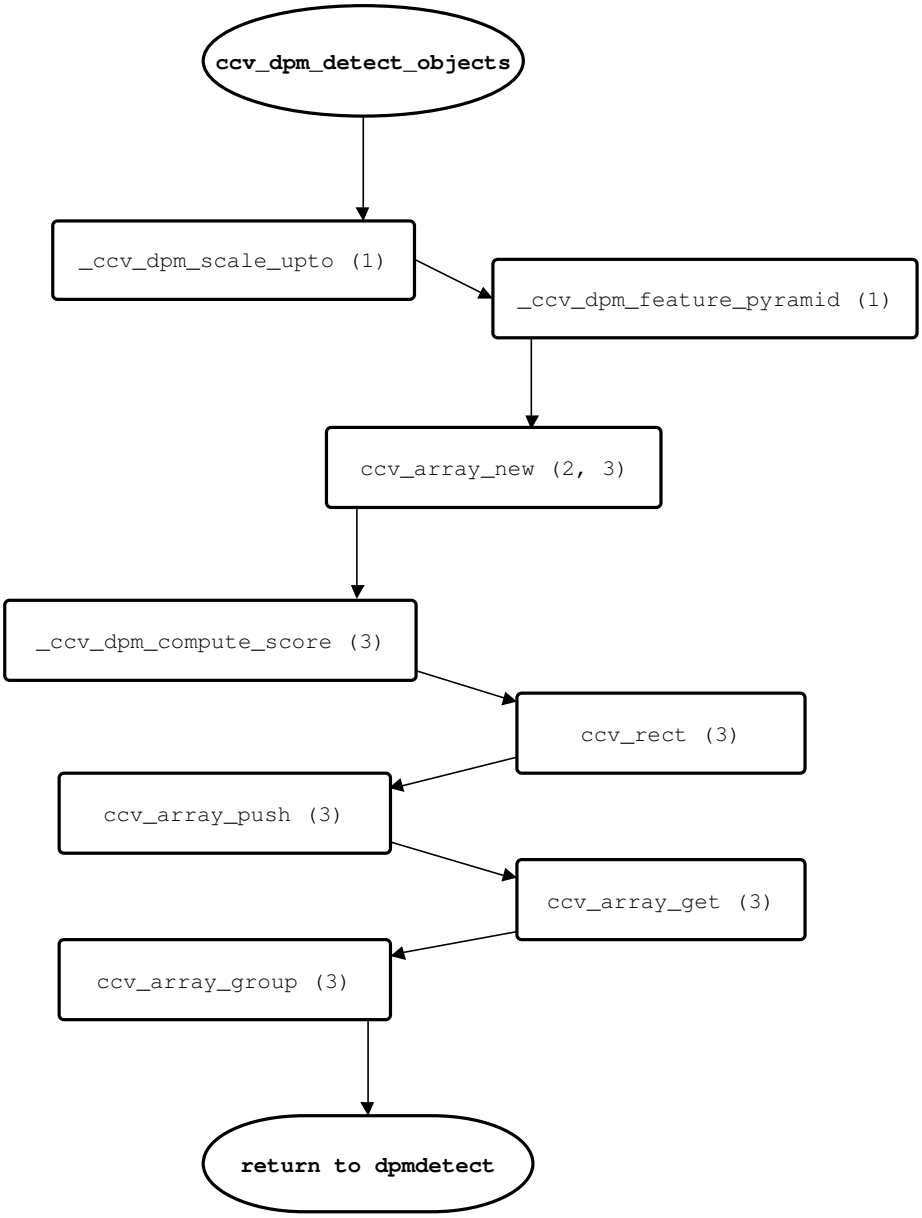


Figure 4.4: Flowchart of the main functions involved in object detection.

3. For each model (loop on count):

   Array structures to hold temporary bounding box data are initialised (`seq` and `seq2`).

   For each scale of the feature pyramid and each model component:

   Responses are computed for root and part filters, followed by distance transforms for part filters only, in a sliding window fashion. The results are stored into response matrices.

   For each sub-window of the current scale, detection scores are computed, and a `ccv_rect_t` data structure representing a bounding box is created and added to the `seq` array.

   Repeated detections of the same object, that may be caused by the sliding-window mechanism, are filtered out through non-maximum suppression (NMS) [36]. The remaining bounding boxes are added to the `result_seq` array.

By virtue of the default values in `params`, NMS is applied for every model evaluated, but this can be disabled by changing the value of `params.min_neighbors` to 0. If instead we change the value

of `params.flags` to that of the constant `CCV_DPM_NO_NESTED`, NMS will be performed only once at the end of the whole process, computing the average bounding box of neighbour bounding boxes.

Moreover, in this implementation, NMS is done through a union-find algorithm adapted from OpenCV's `cvSeqPartition` function. This performs a much more "aggressive" pruning of detections than that of the MATLAB implementations, which is much simpler and checks just if the percentage of overlapping area between two bounding boxes is greater than a given amount — 50% in `voc-release5`.

An overview of the main functions involved in the control flow of `ccv_dpm_detect_objects` is shown in figure 4.4. More in-depth details about these functions will be given in parallel with the description of the sparse implementation in section 5.3.

4.4.3 Detection Output

`ccv_dpm_detect_objects` returns to `dpmdetect` an array of `ccv_rect_t` structures, representing bounding boxes. From this array, for each detected object `dpmdetect` prints the following information to the standard output:

- The path of the current image file (only if performing batch detection).
- The top-left corner coordinates of the root's bounding box, along with its width, height, score and number of parts.
- The top-left corner coordinates of each part's bounding box, along with its width, height and score.
- The total number of detections and time elapsed (only if performing single image detection).

The output of `dpmdetect` on a single image can then be piped to the `dpmdraw` ruby script to produce a new version of the original image, with bounding boxes drawn for root and parts. This is better explained in section 6.2.

4.4.4 Performance Analysis

It is particularly interesting to analyse the performance of `dpmdetect`, because many DPM variants aim to achieve speedups in the detection phase.

We present here three different profile graphs, obtained running `dpmdetect` on the following detection tasks:

1. Detection of a 2-components motorbike class on a set of images containing motorbike examples, selected from the VOC [11] 2007 test set (233 over the 4952 total), using KissFFT [4] for convolution (figure 4.5).
2. Detection of a 3-components bicycle class on the whole VOC 2007 test set (4952 images), using KissFFT as the convolution routine (figure 4.6).
3. Detection of the same 3-components bicycle class on the same test set, but using FFTW3 as the convolution routine (figure 4.7).

A first thing we can notice is that, in all the cases, the largest fraction of time is spent on the convolution branch (`ccv_filter`, 64% of total time in the first example, 94% in the second and 50% in the third). Therefore, a good approach to accelerate the process would be trying to reduce the number of total convolution operations, and this is exactly what the sparselet method does.

In addition to this, comparing the three graphs we can observe the following things:

- Other things being equal, FFTW3 allows a considerable time saving on convolution if compared to KissFFT (50% vs. 94%).
  This leads also to a bigger percentage impact of the other functions in the FFTW3 example. In fact, more functions appear in the graph in figure 4.7 than in that of figure 4.6, because in the latter they do not reach the 0.5% threshold to appear in the graph. Such a small impact is mainly due to the huge amount of time spent on KissFFT convolution.

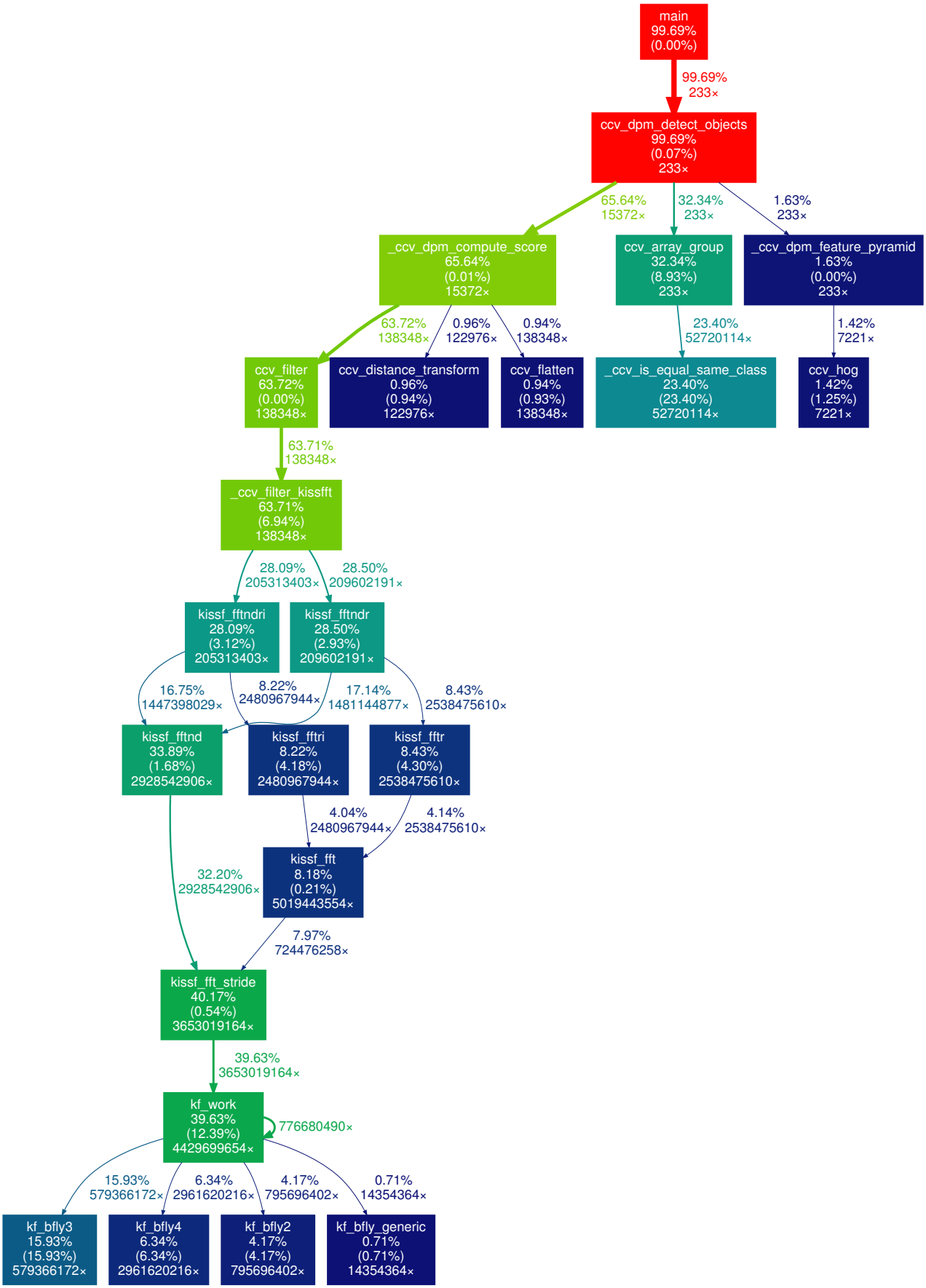


Figure 4.5: DOT graph of `dpmdetect` profile data. Example case 1: detection of a 2-component motorbike class over 233 images containing motorbikes.

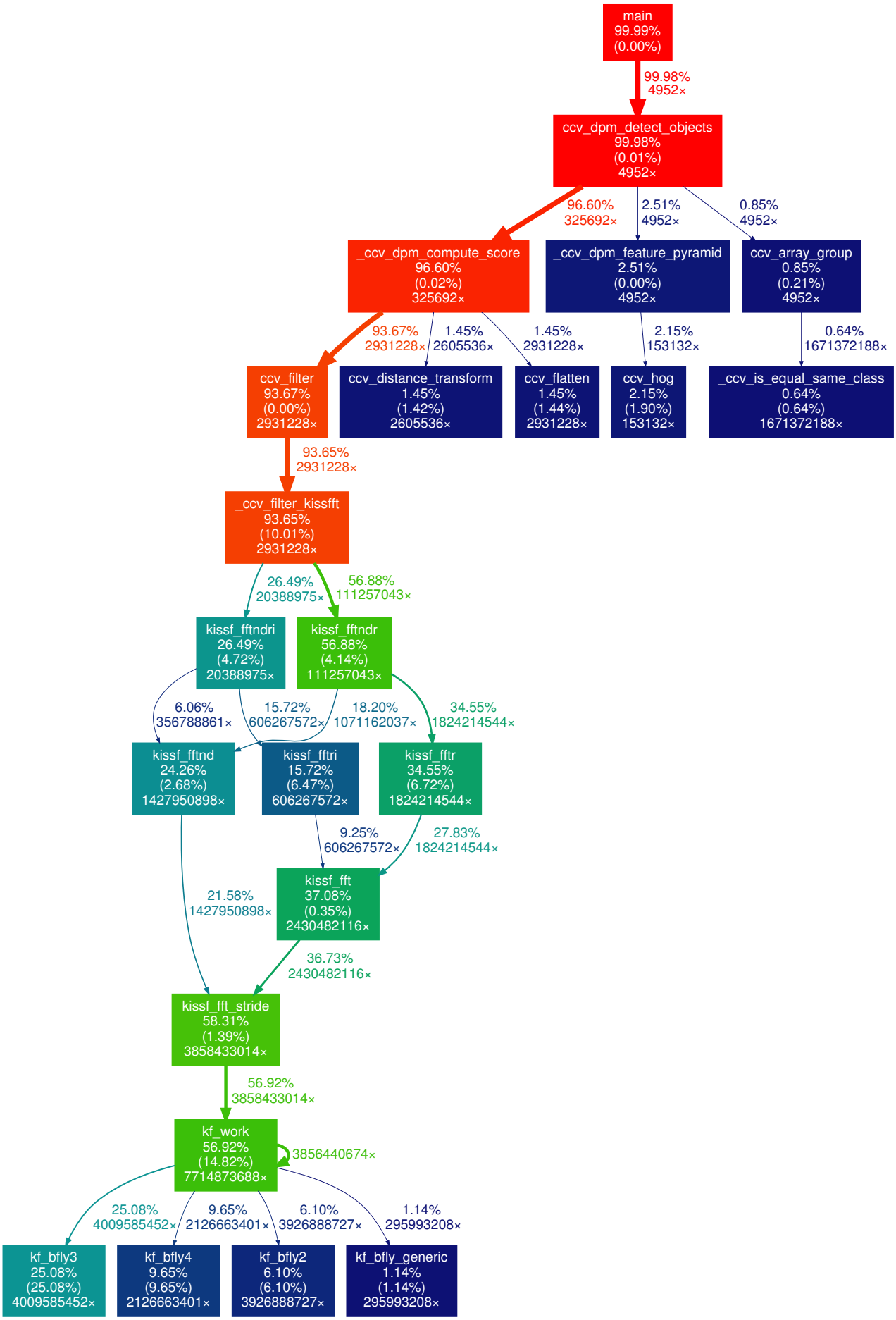


Figure 4.6: DOT graph of `dpmdetect` profile data. Example case 2: detection of a 3-component bicycle class over 4952 images (VOC 2007 test set), with KissFFT convolution.

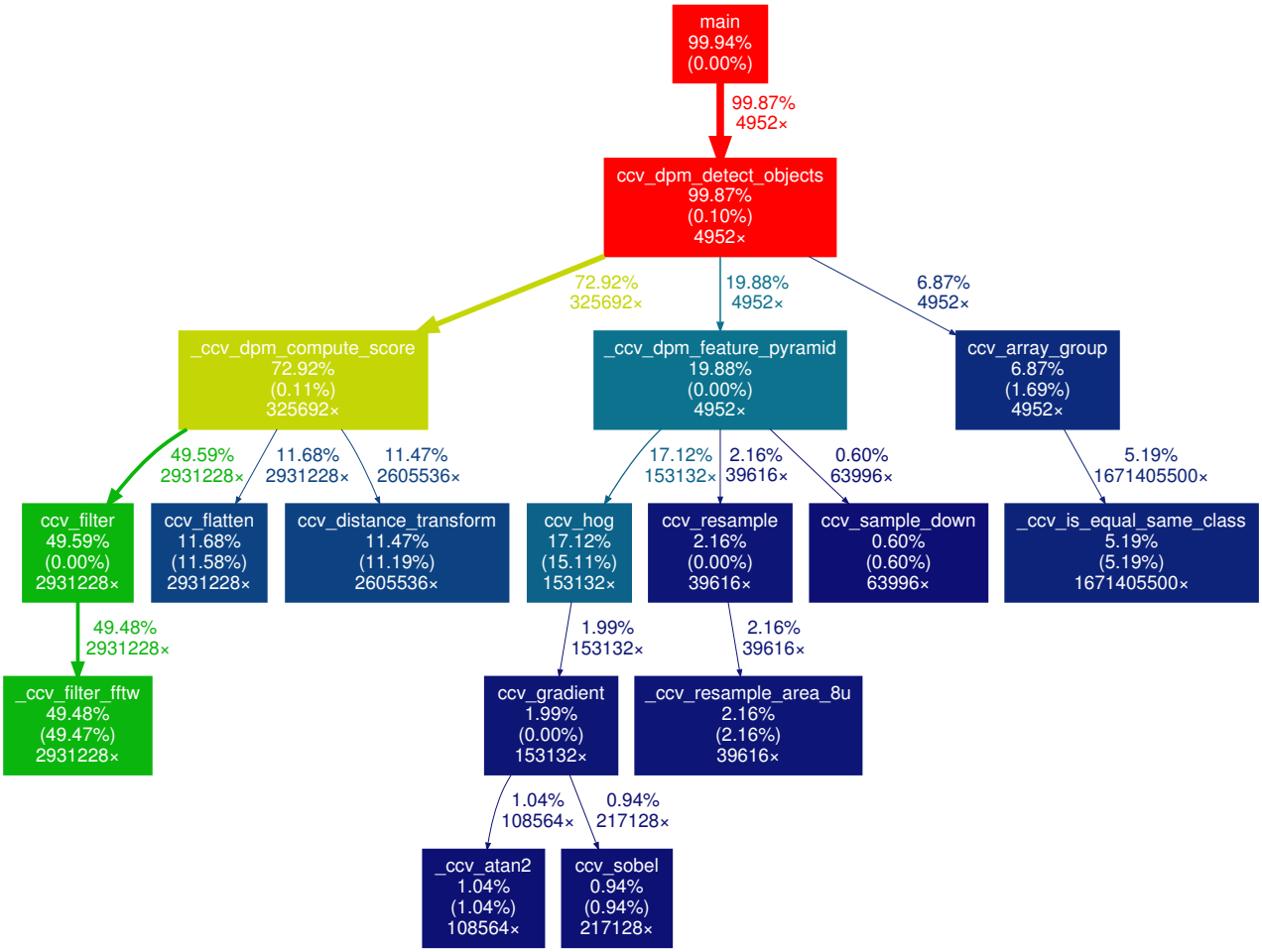


Figure 4.7: DOT graph of `dpmdetect` profile data. Example case 3: detection of a 3-component bicycle class over 4952 images (VOC 2007 test set), with FFTW3 convolution.

- The impact of the NMS functions (branch starting from `ccv_array_group`) is greater with smaller detection sets (first case: 32% of total time) than with bigger ones (second and third cases, 1% and 7% respectively). This depends, however, mainly on the precision of the model used, because an excessive amount of detection hypotheses means more NMS work to do — and most of the resulting detections are likely to be false positives. Hence, in the first place it may point out the greater effectiveness of the bicycle model.
- The time spent on the other functions is almost constant in the first two cases, and anyway proportional with the third case, in which we have seen that the impact of convolution is reduced and hence that of the other functions is increased.
  - The `_ccv_dpm_feature_pyramid` branch, that leads to HOG computation, is the most time consuming among the lesser ones (2% in the first case, 3% in the second, 20% in the third).
  - Then we have other functions branching from `_ccv_dpm_compute_score`:
    * `ccv_distance_transform`, for distance transforms computation (less than 1% in the first case, 1.5% in the second, 11.5% in the third);
    * `ccv_flatten`, which converts a multi-channel matrix to a single-channel one (less than 1% in the first case, 1.5% in the second, 12% in the third).

It is worth to mention again, as we did in section 4.3.5, that the work in detection phase is split among fewer functions than in the training phase.

4.5 Conclusions

From the analysis conducted in this chapter, and in view of an integration of sparse coding into the DPM implementation of CCV, we have drawn the following considerations:

- The code is not optimised and was not conceived for an easy integration of sparse filters.
- The format used to store models into text files is not trivial and rather obscure. Less information is stored if compared to `voc-release5` MATLAB models. For example, CCV's DPM models lack label information.
- It will be necessary to develop specific tools that are able to parse exactly the kind of structure that is represented into DPM model files. These are presented in section 5.2.
- In view of an implementation of additional tools and modules, keeping compatibility with CCV's principles and style should be taken in high consideration.
- The DPM implementation in CCV was not enabled and optimised for multi-object detection, whereas it would be pointless for a sparselet approach to be used in anything but a multi-object detection context.
- Part responses are not pre-computed at the beginning of the detection processes. Instead, they are computed before each distance transform in the innermost loop, when the function `_ccv_dpm_compute_score` is called. This means that to allow response reconstruction, which needs pre-computation of responses to be optimal, we need to change the process and the operations executed in the score detection function.
- As it emerged from several discussion with CCV's author, DPM is currently a neglected approach in the context of object detection in CCV. This means less support and reduced reliability of the current implementation.

In the next chapter, we will describe in detail how we coped with these issues.

Chapter 5

Sparse Filters Integration for CCV's DPM

5.1 Introduction

In this chapter, we are going to examine in details how we realised the integration of sparse coding in the DPM implementation provided with CCV and described in chapter 4.

The work has been divided in two main phases:

1. In the first phase, we implemented a C++ interface to parse models stored in CCV format (see section 4.3.4), take the relevant data from part filters and approximate it using the SPAMS toolbox [35] to create a dictionary of sparse filters:

   (a) Data from part filters is processed and gathered to form an $M \times N$ matrix, where M is the data size and N is the total number of part filters gathered from different model classes. The data size, or "signal size" in SPAMS lexicon, corresponds in this case to the number of feature weights pertaining to a part filter, which as we have seen in section 4.3.4 amounts to $rows \times cols \times feature\ dimension$ of part filter. In our test cases we used 4×4 filters whose dimensionality is set to 31 in CCV according to [16].

   (b) The dictionary of sparse filters and the activation vectors that allow to reconstruct the original data are stored in a text file at the end of this phase.

2. In the second phase, we extended CCV's function set to allow a dedicated object detection for this implementation:

   (a) We realised a modified version of `dpmdetect`, which we named `dpmsparsedetect`, that takes as input a file containing sparse dictionary and activation vector data, as it is produced at the end of the first phase.

   (b) Then we created a function modelled on the original `ccv_dpm_detect_objects`, in which we inserted a modified flow of control with additional layers to perform convolution on sparse filters only, and then reconstruct the original part filter responses via sparse matrix multiplication.

A considerable effort has been made in order to achieve the following outcomes:

- Full compatibility of our C++ interface with the original SPAMS implementation. We aimed and succeeded in avoiding changing the code of the SPAMS functions that we used, by substituting the original MATLAB interface with a C++ one that is suitable to the particular needs of our project. This had to be done at the expense of realising separate modules — hence independent from the CCV library — comprising the dictionary learning tool (`traindl`) and the OMP approximation tool (`omp`). These are written in C++, unlike the CCV library, which is written exclusively in C.

- An extension of the DPM code in CCV that is the least invasive and as much compatible as possible with the existing structure. We aimed and succeeded in producing a code base that mimics as much as possible the organisation and style of the original `ccv_dpm_detect_objects` function. Moreover, we paid particular attention not to create memory leaks or bugs, so every new step we added in the control flow has been extensively tested in order to assess the correctness of the procedure as well as its memory efficiency. As a result, our sparse implementation succeeds in maintaining the low resource usage that was characteristic of the pre-existing implementation.

The remainder of this chapter is structured as follows:

- In section 5.2 we describe in detail how we planned and realised the sparse dictionary training.
- In section 5.3 we provide a detailed analysis of how we implemented the reconstruction of part responses.

5.2 Approximation of Part Filters: `traindl` and `omp`

In this section, we will give a detailed explanation of how we employed the SPAMS modules to train a dictionary of sparse filters (`traindl`), and retrieve the coefficient matrix of activation vectors to reconstruct the original data (`omp`).

The SPAMS toolbox [35], currently at version 2.5, comprises optimisation functions to solve several sparse estimation problems. These are written in C++ and can be used through a MATLAB interface, that allows the compiled C++ code to be called from within MATLAB. Since version 2.2, SPAMS has also two new interfaces, for python and R.

In this project, we are interested mainly in two functions, which in the original MATLAB-interfaced implementation are called: `mexTrainDL` and `mexOMP`.

5.2.1 `mexTrainDL`

A general function for dictionary learning, it aims at performing the following optimisation:

$$\min_{\boldsymbol{D}\in C} \lim_{N\to+\infty} \frac{1}{N}\sum_{i=1}^{N} \min_{\alpha_i} \left(\frac{1}{2}||\boldsymbol{X}_i - \boldsymbol{D}\alpha_i||_2^2 + \Psi\left(\alpha_i\right)\right) \tag{5.1}$$

where $X \in \mathbb{R}^{M\times N}$ is the input data matrix, D is the dictionary that will be learnt, α is a matrix of coefficients, Ψ controls the amount of sparsity in this matrix and C is a constraint set for the dictionary. In our case, we do the sparse coding with OMP (param.mode = 3), so the objective from 5.1 becomes:

$$\begin{gathered}\min_{\boldsymbol{D}\in C} \frac{1}{N}\sum_{i=1}^{N} \left(\frac{1}{2}||\boldsymbol{X}_i - \boldsymbol{D}\alpha_i||_2^2\right) \\ \text{subject to } ||\alpha_i||_0 \leq \lambda \qquad \forall i = 1,\ldots,N\end{gathered} \tag{5.2}$$

where, again, λ represents the parameter that regulates the level of sparsity in α. The constraint we use for the dictionary is the default one (param.modeD = 0):

$$\begin{gathered}\boldsymbol{D} \in \mathbb{R}^{M\times K} \\ \text{subject to } ||\boldsymbol{D}_j||_2^2 \leq 1 \qquad \forall j = 1,\ldots,K\end{gathered} \tag{5.3}$$

The input parameters of `mexTrainDL` are the following:

Parameter	Semantics
D	A pre-existing dictionary (optional); if not provided, it is initialised with random elements from the training set.
K	Size of the dictionary; optional if D is provided (it takes its size).
λ	Controls the amount of sparsity in the coefficient matrix.
iter	Number of iterations in dictionary learning; if negative it performs the computation for the given number of seconds.
mode	Defines the type of dictionary learning problem.
modeD	Defines the constraints on the dictionary.
clean	If true, prunes the dictionary from unused elements.
verbose	If true, prints more execution information.
numThreads	If the code is compiled with OpenMP on a multi-core or multi-cpu platform, it specifies the number of threads to use. With -1, the default value, it uses all the available CPUs / cores.

Table 5.1: Partial list of parameters to customise the `mexTrainDL` dictionary learning task. Some, which are not relevant to our case, have been omitted.

1. An $M \times N$ matrix of data `X`, from which the dictionary will be learnt.
2. A `param` struct that allows the customisation of several settings.
3. A `model` resulting from a previous dictionary learning task, to resume it from where it was interrupted (optional).

Table 5.1 summarises the parameters that can be passed to the function with the `param` struct.

The output parameters are:

1. The dictionary `D` learnt from data matrix `X`.
2. A `model` that allows to restart the learning task from where it is stopped (optional).

5.2.2 `mexOMP`

This function implements an Orthogonal Matching Pursuit (OMP) algorithm: it takes the original data matrix `X` and the dictionary `D` learnt from it, and computes a matrix `A` of coefficient vectors $\alpha_1, \ldots, \alpha_N$ that solve, for each column X_i in `X`, the following NP-hard problem (in this specific case):

$$\begin{aligned} &\min_{\alpha_i \in \mathbb{R}^K} ||\boldsymbol{X} - \boldsymbol{D}\alpha_i||_2^2 \\ &\text{subject to } ||\alpha_i||_0 \leq L \qquad \forall i = 1, \ldots, N \end{aligned} \tag{5.4}$$

where L represents the maximum number of elements in each decomposition.

In addition to the data matrix `X` and the dictionary `D`, this function takes as input parameter also a `param` struct to specify some relevant settings. These are briefly listed in table 5.2.

The output parameters are the matrix of coefficient vectors `A` and, optionally, the greedy regularisation `path` of the data matrix `X`.

5.2.3 Building a K-sized Dictionary from DPM Part Filters: `traindl`

We implemented this command-line interface to emulate as much as possible the functionality of the `mexTrainDL` interface for MATLAB. Hence, we take input arguments with the scheme outlined in table 5.3.

Parameter	Semantics
L	Maximum number of elements in each Cholesky-based decomposition; by default is $\min(M, K)$.
ε	Threshold on the squared ℓ_2-norm of the residual.
λ	Penalty parameter.
numThreads	If the code is compiled with OpenMP on a multi-core or multi-cpu platform, it specifies the number of threads to use. With -1, the default value, it uses all the available CPUs / cores.

Table 5.2: List of parameters to customise the `mexOMP` execution.

Keyword	Description	Default value
`--K`	Size of dictionary (required).	—
`--lambda`	Sparsity level (required).	—
`--iter`	Number of iterations (required).	—
`--lambda2`	Optional parameter.	10^{-9}
`--mode`	Dictionary learning problem. In range $[0-6]$.	2 `(PENALTY)`
`--posAlpha`	Adds positivity constraints on the coefficients; incompatible with mode $= 3, 4$.	`false`
`--modeD`	Constraint type for the dictionary. In range $[0-3]$.	0 `(L2)`
`--posD`	Adds positivity constraints on the dictionary; incompatible with modeD $= 2$.	`false`
`--gamma1`	Parameter for modeD ≥ 1.	0
`--gamma2`	Parameter for modeD $= 1$.	0
`--batchsize`	Minibatch size.	$256 \times (\text{numThreads} + 1)$
`--iter_updateD`	Number of BCD iterations for the dictionary update step.	1
`--modeParam`	Optimisation mode. In range $[0-3]$.	0 `(AUTO)`
`--rho`	Optional tuning parameter for modeParam ≥ 1.	1.0
`--t0`	Optional tuning parameter for modeParam ≥ 1.	10^{-5}
`--clean`	Cleans unused elements from dictionary.	`true`
`--verbose`	Displays more information during execution.	`true`
`--numThreads`	Number of threads to use on enabled multi-core / multi-cpu machines.	-1
`--file`	Optional path to text file containing a list of model files.	—
`--outfile`	Optional output file to store the computed dictionary.	`sparse.dict`

Table 5.3: Input arguments of the `traindl` interface for dictionary learning.

The parameters gathered from the command line are then used to build a `ParamDictLearn<double>` object, which is a data structure conceived in SPAMS specifically to store those parameters. Even if CCV stores the feature weights as floating numbers (cf. table 4.3) we decided to use the `double` type here because the `float` precision setting in SPAMS is reported as experimental and not extensively tested.

While the dictionary learning parameters are parsed in a way that is almost directly translated from the MEX interface for MATLAB, our implementation diverges about how the input data matrix `X` is built. In fact, in our case the input matrix is composed by the feature weights of DPM part filters. Therefore, we accept as input parameters the model files built by CCV's `ccv_dpm_mixture_model_new` in two ways: either as a list of file paths after the other command line arguments — and without an option keyword —, or as a text file containing a list of paths to model files (in the same way they are passed to our extended version of `dpmdetect`, see section 4.4.1), signalled by the `--file` keyword, or a combination of both methods.

Since the `traindl` and `omp` CLIs share several common operations, in our design we decided to implement these common functions in a separate `.hpp` file. Hence, their description is just briefly mentioned at this stage, and postponed to section 5.2.5 for more details.

Once the program determines how the input model files are passed, it parses them with the `parseModelFiles` function. This returns a vector of arrays of double pointers, each of which contains the feature weights of a part filter. This vector is then used to build a single array of `double` pointers through the `getDataFromPtrVector` function. We do this to comply with the constructor of SPAMS `Matrix` class, which accepts this kind of data structure to build a non-empty matrix, namely one with input data provided.

The next step is the actual dictionary learning, accomplished using `trainDL`, a wrapper function that builds a SPAMS `Trainer` object to train the dictionary from the input matrix. The input parameters needed by this function, and hence by the `Trainer` class constructor or functions, are:

- the `Matrix<double> X`, built with the data taken from the model files;
- the `Matrix<double> D`, an empty matrix to store the dictionary data;
- the `ParamDictLearn<double> param`, containing the dictionary learning parameters;
- the size K of the dictionary (mandatory command-line argument);
- `batchsize` and `NUM_THREADS`, taken from command-line arguments or from default values if omitted (see table 5.3).

Then the `Trainer` object calls the function `train` with the data matrix as input parameter, and from this it creates and stores internally the dictionary. This can then be obtained with the function `getD`, which copies the internal dictionary into the `Matrix` object passed to it. This is why we pass also an empty `Matrix` object `D` to the wrapper function: it will be eventually filled with the learnt dictionary.

Finally, the dictionary data is stored in a file. This is accomplished through our `saveToFile` function, which can either save the file to a conventional path (`sparse.dict` in current directory) or save it to a path specified by the user with the `--outfile` command-line keyword. After this and a quick memory cleanup, the execution of `traindl` finishes.

5.2.4 Computation of the Matrix of Coefficient Vectors: `omp`

Similarly to `traindl`, the `omp` CLI takes command-line arguments with the scheme explained in table 5.4.

The parameters in this case are much fewer, and they all have a default value except for the path to the dictionary file created with `traindl`, that has to be specified by the user.

It is also necessary to provide the same model files that were used to build the dictionary with `traindl`. In fact, we need to extract the original data again, since it constitutes the matrix `X` which is a required input parameter of the OMP function. The design is the same as before:

Keyword	Description	Default value
`--dict`	Path to the dictionary data file, created using `traindl` (required).	—
`--L`	Maximum number of elements in each decomposition.	$\min(M, K)$
`--eps`	Threshold on the squared ℓ_2-norm of the residual.	0
`--lambda`	Penalty parameter.	0
`--numThreads`	Number of threads to use on enabled multi-core / multi-cpu machines.	-1
`--file`	Optional path to text file containing a list of model files.	—

Table 5.4: Input arguments of the `omp` interface for coefficient matrix computation.

command-line option `--file` or list of paths after the other arguments. Then we use the function `parseModelFiles` as in `traindl` to extract part filters' weights from the model files.

In this case, we also need to parse the data from the dictionary file and store it into a suitable format — which, as we said before, is an array of `double` pointers. To achieve this, we call the `getDictionaryData` function, which takes the dictionary file and the data size M (i.e. the number of feature weights in a part filter in this case), and returns the dictionary size K and an array of pointers to build a `Matrix` object holding the dictionary.

Consequently, we parse the three main OMP parameters — L, ε and λ — forcing them to be single-valued. In fact, they could be also vectors that specify a different value for each input signal X_i, but this is reported as an experimental feature in SPAMS, so we disabled it in our implementation.

Then the data gathered is passed to the `OMP` wrapper function, which needs the following input parameters:

- the `Matrix<double> X`, built with the data taken from the model files;
- the `Matrix<double> D`, built with the data taken from the dictionary file;
- the `SpMatrix<double> alpha`, an empty sparse matrix to store the coefficient vectors;
- three pointers to the values of L, ε and λ;
- the sizes of L, ε and λ (which in this case will all be 1), and the usual `NUM_THREADS` variable.

Finally, we append the path of each model file and the data from the sparse matrix obtained with `OMP` at the end of the dictionary file, using again the function `saveToFile` with a flag that specifies that the `app` mode has to be used when the `ofstream` object is constructed. After the usual cleanup, the execution of `omp` finishes.

5.2.5 Shared Functions and Compilation Settings

In this section we are going to describe briefly:

- the behaviour of some common functions used in both `traindl` and `omp` modules;
- the settings used to compile the new modules.

`parseModelFiles`. Parses model file names obtained directly from input arguments or listed into a text file. When reading the model files, it discards all the data about root filters and also the other information about part filters (e.g. deformation features). It checks that the sizes are consistent, requiring a square filter of the same size for all part filters. For this reason, an option to set a square filter size was added to `dpmcreate`. It returns: a vector of arrays of double pointers, the sizes m and n of the part filters matrix and, if called from `omp`, also a vector of paths to model files.

`getModelsFromFile`. Called when the `--file` option is used, its purpose is to convert the list of model files in the provided file to an array of file names (with the classical type `char**`). Then, this will be passed to `parseModelFiles`, along with other file names that may have been specified directly in the command-line.

`saveToFile`. Used to save data in the file provided as input parameter. If called by `traindl`, the data will be written in the standard `openmode` for `ofstream` (`out`), so any content that may already be present in the output file will be overwritten. In this case, what is passed to this function and saved into the file is the dictionary data. Otherwise, when called by `omp`, this function is provided also with the list of paths to model files and a flag specifying that the `append` `openmode` has to be used when creating the `ofstream` object. This means that the content of the output file will not be overwritten, because we assume that it contains the dictionary data already, and what we want to do is to append the list of model files used and the coefficient matrix data after the pre-existing dictionary data.

`getDataFromPtrVector`. Converts the data stored into a `vector<double*>` object to an array of `double` pointers, that is the format required to construct a SPAMS `Matrix` with existing data. It also frees the memory used by the input vector, which is no longer needed.

`getDictionaryData`. Used by `omp` to read the data from the dictionary file. First, it reads the file and stores its contents in a vector of arrays of `double` pointers. At this point, we have the same data structure that can be passed to `getDataFromPtrVector`, so we call this function to produce an array of `double` pointers (again, because this is the type required to build a SPAMS `Matrix`).

Compilation settings. We extended the original `makefile` in `bin` folder, adding the compilation settings for the new modules. We use `g++` as the compiler, with the following compilation flags:

```
CFLAGS_SPAMS = -O3 -mtune=native -fomit-frame-pointer -Wall -fopenmp
```

and these library flags:

```
LDFLAGS_SPAMS = -O -lblas -llapack -lgomp
```

Hence, the BLAS, LAPACK [9] and GOMP [22] libraries are all needed in order to compile `traindl` and `omp`. Both the compilation and library flags are the same used to compile the MATLAB-interfaced version of SPAMS.

5.3 Response Reconstruction and Object Detection

In this section, we will examine in detail how we exploit, in the DPM detection chain, the dictionary of sparse filters created with the SPAMS toolbox (as outlined in section 5.2).

In particular, the main advantage is that we can compute convolution only on the K sparse filters of the dictionary, instead that on the N part filters of all the model classes. Since K can be set as smaller than N as desired — provided that accuracy is not significantly affected —, this approach would result in a considerable speedup.

It should be obvious that the advantage of this sparse filters method is apparent only in multi-class object detection. In fact, to achieve a speedup, the size K of the dictionary has to be smaller than the number of part filters N, but N is of course much bigger than the number of parts of a single model. Therefore, in most cases, K will be greater than that number too, so for single-class object detection this approach would be certainly slower and more inconvenient than the classical DPM.

Line(s)	Description	Type
1	Filter size ($rows$ $cols$), dictionary size (K).	`int` ($\times 3$)
$[2 - K + 1]$	Weights data for sparse filters.	`double` ($\times(rows \times cols \times 31)$)
$K + 2$	Number of models involved (M).	`int`
$[M_{start} - M_{end}]$	Paths of model files. $\begin{pmatrix} M_{start} = K + 3 \\ M_{end} = K + 2 + M \end{pmatrix}$	`char**` (text)
$[M_{end} + 1]$	Size of coefficient matrix (N K).	`int` ($\times 2$)
$[A_{start} - A_{end}]$	Coefficient matrix data. $\begin{pmatrix} A_{start} = M_{end} + 2 \\ A_{end} = M_{end} + 1 + N \end{pmatrix}$	`double` ($\times(N \times K)$)

Table 5.5: Data organisation of a complete `sparse.dict` file.

5.3.1 A New Interface for Detection: `dpmsparsedetect`

The process described in section 5.2 is placed ideally between the training and testing phases of a DPM detector. What we are going to deal with now is a process that actually replaces the original detection, as done in CCV by `dpmdetect` (section 4.4.1).

We created a tool with the same role as `dpmdetect`, named `dpmsparsedetect`, that can be called with a command in this format:

```
./dpmsparsedetect <image.png> <sparse.dict> [1]
```

or:

```
./dpmsparsedetect <image_list.txt> <sparse.dict> [1]
```

1. The first argument is again an image on which the detection is carried out, or alternatively, in the second case, a text file containing a list of images.

2. The second argument is the file produced by `traindl` and extended by `omp`, that we conventionally named `sparse.dict`. This contains the information about dictionary data, model files and coefficient matrix for reconstruction. The format of a complete `sparse.dict` file is shown in table 5.5.

3. The third argument is an optional integer, which specifies the index of a model in the list stored inside `sparse.dict`. For example, if `sparse.dict` was produced from these model files:

   ```
   bicycle.model
   car.model
   cat.model
   person.model
   ```

   the index '1' stands for the first model in the list (`bicycle.model`), '2' for the second (`car.model`), and so on. This argument is used to perform detection on one class only, thus allowing evaluation of a single model. This is done mainly to evaluate a model's accuracy, because as we said before there is a speed loss when this method is used for single-class detection.

Essentially, `dpmsparsedetect` triggers a detection process which is similar to the original one, but with some new layers added in the control flow, to deal with the fact that we are now doing

convolution with sparse filters, so we need to reconstruct the original responses before computing a detection score. We have implemented a new function, called `ccv_dpm_sparse_detect_objects`, which is in charge of this extended process (see section 5.3.2).

The way `dpmsparsedetect` reads image files is exactly the same as that described for `dpmdetect` in section 4.4.1, but in this case there is more to do: we need to read and store the sparse filters dictionary and the coefficient (`alpha`) matrix contained in `sparse.dict`. Moreover, since the data stored in this file concerns only the feature weights of part filters, we still need to read the model files to store the other relevant data (root filter weights, deformation features, etc.), and this is why we listed their paths in `sparse.dict`.

Therefore, `dpmsparsedetect` operates as follows:

1. The dictionary data is stored employing a `ccv_dpm_root_classifier_t` struct to mimic a model component with K parts, that are the sparse filters contained in the first part of `sparse.dict`. We called it `sparse_classifier`.

2. The size of a sparse filter is read, along with the total number (K). Then a `ccv_dpm_part_classifier_t` array is allocated for K elements, and the actual data is read line by line and stored into the `w` matrix of each `ccv_dpm_part_classifier_t` (refer to section 4.3.4 and figure 4.2 for the organisation of these data structures).

3. In this way, all the sparse filters are stored in the array in a loop, and then the array itself is assigned to the `part` field of `sparse_classifier`.

4. After all the sparse filters are read, we read a line with the number of models involved. This is followed by the paths of the model files, which are used to read and store models into an array in a loop, exactly in the same way as it is done in our extended version of `dpmdetect`.

5. The next line in `sparse.dict` contains the number of part filters N and the dictionary size K. These are used to allocate a `ccv_dense_matrix_t` of the appropriate size. In fact, since the data we use to build the `alpha` matrix comes from a file and was converted to a dense matrix at the end of `omp` execution for simplicity, we decided to use this CCV data structure, instead of the one called `ccv_sparse_matrix_t`, because of its ease of use.

6. The next lines are read in a loop on N, thus storing the data pertaining to the `alpha` matrix.

Once this initial setup is done, we have collected the data that needs to be processed just once at the beginning, so we can delegate the actual detection process to the `ccv_dpm_sparse_detect_objects` function.

5.3.2 `ccv_dpm_sparse_detect_objects`

This function broadly follows the same scheme outlined for `ccv_dpm_detect_objects` (section 4.4.2), but with a modified flow of control comprising new layers to achieve reconstruction of original part filter responses.

The function declaration, which has been added to the header file `ccv.h`, is:

```
ccv_array_t* ccv_dpm_sparse_detect_objects(ccv_dense_matrix_t* a,
ccv_dpm_root_classifier_t* sparse_classifier, ccv_dense_matrix_t* alpha,
ccv_dpm_mixture_model_t** models, int num_models, int model_index, ccv_dpm_param_t
params);
```

Therefore, the input parameters of this function are:

- the image on which detection is carried out (`a`);
- the data structure containing data of sparse filters (`sparse_classifier`);

- the matrix containing the coefficient vectors (`alpha`);
- the array of model structures (`models`), their number (`num_models`), the model index to evaluate a single model (`model_index`);
- the detection parameters (`params`).

A detailed description of the control flow is the following:

HOG pyramid. First of all, we compute the HOG feature pyramid, at several different scales, as in `dpmdetect`.

Pre-computation. We convolve all the sparse filters with all the scales of the HOG feature pyramid. This stage concentrates all the computationally heavy part of the process in one function: `_ccv_dpm_compute_sparse_responses`.

Reconstruction. Due to the pre-computation of sparse responses, we can pre-compute also the responses of all the part filters of the models we are evaluating, at all scales of the feature pyramid. This is done by reconstructing the original responses via sparse matrix multiplication. Being convolution a linear operation, we multiply the sparse responses with the coefficient vectors of the `alpha` matrix, thus obtaining approximately what would have been the original responses of part filters. Each part filter is associated to a coefficient vector, namely a row in the `alpha` matrix. As in [40], we do the multiplication only over non-zero elements of the coefficient vector: the temporary 2D array that we use to compute the reconstructed responses has all its elements initialised to 0.0 with `memset`, so an `if` statement on the current coefficient is enough to skip multiplication when it is zero.

Score computation. With all the part responses pre-computed, we can plug each of them into the function that computes scores for root and part filters: `_ccv_dpm_sparse_compute_score`. This differs from `_ccv_dpm_compute_score` only in the fact that the latter receives an empty `responses` matrix and then fills it with part responses. In this case, we have already built that matrix in the pre-computation step, so in the new function we just compute the root response and distance transforms on the part responses.

Detection hypothesis. From this point, the process is exactly the same as in `dpmdetect`: we compute a score on sliding sub-windows of the image by summing the root and part scores computed in the previous step. This constitutes the score for a detection hypothesis.

Non-maximum suppression. As in `dpmdetect`, we employ again the non-maximum suppression algorithm to prune erroneous multiple detections of the same object (unless this step is disabled, as explained in section 4.4.2).

Each of these steps is done looping on the models, their components and the scales of the HOG feature pyramid.

An important design choice is checking if we are evaluating just a single model or not at each step of the process where it is sensible to do so. With this condition check, in fact, we can avoid performing operations on all the models but the one we want to evaluate. Unfortunately, this cannot prevent doing the convolution on all the sparse filters, so if we really want to evaluate how much the sparse method affects accuracy on a single model, we need to sacrifice speed.

At the end of the process, we clean up the memory used by all the additional data structures that we have allocated in this function, and return the array of bounding boxes exactly as in `dpmdetect`.

5.3.3 Detection Output and Memory Cleanup

The detection output produced by `dpmsparsedetect` looks exactly like that of `dpmdetect`, described in section 4.4.3.

Furthermore, a very important thing that has to be done before the end of execution is freeing the space allocated in `dpmsparsedetect` for the new data structures: the `sparse_classifier`, all the sparse filters and the `alpha` matrix.

5.4 Conclusions

In this chapter, we have shown how we designed and realised the integration of a sparse filters approach for the DPM implementation included in CCV. We have seen that the work has been organised in two main phases:

1. Implementation of tools to produce sparse filters by approximation of part filters from several object class models.

2. Development of an interface and an updated flow of control to perform detection with sparse filters.

We have outlined the challenges posed at each step of the implementation, and how we dealt with them to successfully achieve our task. This implied a thorough knowledge of the structure of a DPM model file, where the part filters data is stored, and of how the detection process was implemented in CCV.

Moreover, since ours is an extension of an existing implementation, a lot of effort was needed to ensure that our design choices were unobtrusive and highly compatible with the original code.

In the next chapter, we will see how this new DPM detection approach can be evaluated from the point of view of speed and accuracy.

Chapter 6

Evaluation

6.1 Introduction

In this chapter, we describe the tools available to evaluate object detection systems in CCV and `voc-release5`, and those that we developed to suit better our needs. Then we show the results of our tests on some example classes.

6.2 Evaluation Tools

Both the CCV library and `voc-release5` provide useful programs to aid the evaluation of the respective detection systems, but they differ in many ways. In CCV, the following programs are provided:

- `dpmext`, a script that extracts bounding box data from the files contained in the folder passed as input parameter. The data is printed to the standard output in the following format [32]:

  ```
  <File Path> x y width height \n
  ```

  where `x` and `y` represent the coordinates of the top-left corner of the bounding box.

- `dpmvldtr`, a script that prints to the standard output the average recall and the number of false positives computed from a list of object detections in `dpmdetect` format (see section 4.4.3).

- `dpmdraw`, a script that uses detection data in `dpmdetect` format to draw bounding boxes on an image.

In `voc-release5`, we have several visualisation and testing functions. We list here only those that we used for our evaluation:

- `visualizemodel`, a function that allows to visualise the HOG representation of root and part filters of a model stored as a MATLAB struct with `voc-release5` format.

- `clipboxes`, a function to clip the bounding box coordinates of a pool of detections to the image boundary.

- `nms`, a function that performs non-maximum suppression in the `voc-release5` fashion: it greedily selects high-scoring detections and discards those whose area is covered by a previous detection over a certain percentage threshold.

- `pascal_test`, a function that does approximately the same as `dpmdetect`: given a VOC dataset (specified by year and set label, e.g. `train`, `test`) and a class model, it computes all the detections and returns a cell array of detection matrices. Each matrix contains the

detection coordinates and scores. It is important to note here that the format is different from that used in CCV: instead of having the width and height data of a bounding box, here we have the bottom-right corner coordinates.

- `pascal_eval`, a function to compute the score of a list of detections for a model on a given VOC dataset. It returns the average precision and a list of precision and recall values that can be used to plot a precision-recall curve.

Since we decided to work in the CCV framework but using the VOC 2007 and 2012 datasets for our tests, we have developed some new, modified versions of the original `dpmext` and `dpmvldtr` scripts:

- `vocext` is written in python and allows to parse the annotation files in a VOC 2007–2012 dataset folder in order to extract and write into separate files a list of positive and negative files. For positive ones, bounding box data is also retrieved and stored in CCV format. We added also a functionality to automatically produce positive and negative lists for all the 20 classes or only for those whose files are not present yet. The files can be saved to a custom path if provided by the user, or in the folder where the script is run with a standard filename if these optional arguments are omitted. The purpose of this script is to provide files that can be passed to `dpmcreate` as `positive-list` and `negative-list` arguments.

- `voclist`, which simply lists all the images of a VOC dataset with at least one instance of a target class. This script lists each relevant file just once, unlike `vocext` that, for positive examples, lists a file as many times as the number of instances of the target class in the file. Moreover, bounding box information are omitted. The reason for this additional script is that, while `vocext` is devised to aid the listing of files that are necessary for the training phase, this instead is used in the testing phase to see the performance of a model on a set of selected images that all contain instances of that class.

- `vocvldtr`, which is a version of `dpmvldtr` adapted to parse the annotation format used in the files of VOC datasets. Since the format in these files is different from the one that `dpmvldtr` is able to parse, we developed this program to evaluate CCV detections performed on VOC datasets with the same metrics of `dpmvldtr`.

Moreover, we also planned to write an alternative version of `dpmdraw` to surround detected objects with more visible (slightly thicker) bounding boxes, omitting part detections if desired by the user, and with class labels. This makes particularly sense in a multiple-class detection context, where we want to see clearly to which class every detection refers, but this kind of detection was not supported by CCV natively. Moreover, there is no label information inside model files, so they are "anonymous". For these reasons, we abandoned that objective leaving it as a proposal for future improvement of the library.

6.3 Model Set

We assessed the accuracy of the sparse coding framework using a set of 10 model classes with a fixed size for part filters set to 6×6.

It is worth noting that the original implementation provided an automatic computation of optimal part sizes. However, even if we forced the models to have 6×6 part filters, we have not experienced a significant loss of AP if compared to the classical training, other things being equal. This is confirmed by the fact that, when training models in `voc-release5`, the part filters are automatically set to a 6×6 size.

In addition to that, we must acknowledge that the models we have used in our tests are not particularly accurate in detection. This is because a thorough training needs a lot of expertise in adjusting training parameters and above all a lot of time and computation, since some training sessions last for days.

Name	VOC set label	C	P	+	−	*BG images*
aeroplane07	2007 trainval	1	8	331	12,000	4,771
bicycle07_1	2007 trainval	1	6	418	12,000	4,756
bicycle07_2	2007 trainval	2	6	418	12,000	4,756
bus07	2007 trainval	1	8	272	12,000	4,814
car07_1	2007 trainval	1	6	1,644	12,000	4,250
car07_2	2007 trainval	2	6	1,644	12,000	4,250
motorbike12	2012 trainval	2	8	801	2,000	16,550
person07_1	2007 trainval	1	6	5,447	12,000	2,916
person12_3	2012 trainval	3	8	2,000	3,000	7,542
tvmonitor12	2012 trainval	2	8	893	2,500	16,480

Table 6.1: Parameters used to train the model classes used in our evaluation experiments. C: number of components. P: number of parts. +: number of positive examples used. −: number of negative examples used. *BG images*: number of images from which negative examples were collected. The `bicycle07_2` and `car07_2` model classes were trained with a reduced number of relabels, data-minings and gradient descent iterations.

In this case, the focus is not on the accuracy of the models, that could improve with better training parameters and more time dedicated do it, but rather on how well the sparse coding method manages to maintain the accuracy of the original detection method.

The parameters used to train the models are summarised in table 6.1. In total, we have 122 part filters, from which we have built two dictionaries of sparse filters: one with 40 bases and the other with 20. All the tests were carried out on the whole VOC 2007 test set (4,952 images).

6.4 Accuracy Assessment

A performance comparison for 5 of the 10 models tested is shown in figures 6.1 to 6.5. For each class, the precision-recall curve and average precision (AP) are shown for:

Baseline. The standard detection done with `dpmdetect` using a single model file.

Sparse (40 bases). A detection done with `dpmsparsedetect` using the dictionary with 40 bases.

Sparse (20 bases). A detection done with `dpmsparsedetect` using the dictionary with 20 bases.

For the sparse test sets, we have used our indexed detection mode (see section 5.3.1) to force single model detection, so that the results could be compared with the baseline ones.

Since `vocvldtr` gives only a measure of the recall and the number of false alarms, we used the `pascal_eval` function to carry out this evaluation. This implies that the detection output produced by `dpmdetect` and `dpmsparsedetect` was converted to a suitable format to be parsed with that MATLAB function. The non-maximum suppression step was placed right after this conversion. We have decided to disable non-maximum suppression in CCV, and use instead the `nms` function of `voc-release5` just before assessing detections because, in some previous tests that we had done, we had experienced a general loss of AP with CCV's non-maximum suppression[1]. The detection threshold was set to 0.0 throughout all the tests.

Overall, we can notice how the curve shapes that define average precision are approximately preserved in most of the cases. The dictionary with 40 bases gives a small loss in terms of AP for almost all models, whereas the one with just 20 bases proves to be generally less accurate.

[1]Additional tables showing the impact on AP of non-maximum suppression done by CCV and voc-release5, as well as the AP comparison for the remaining 5 models included in the sparse dictionaries can be found in appendix B.

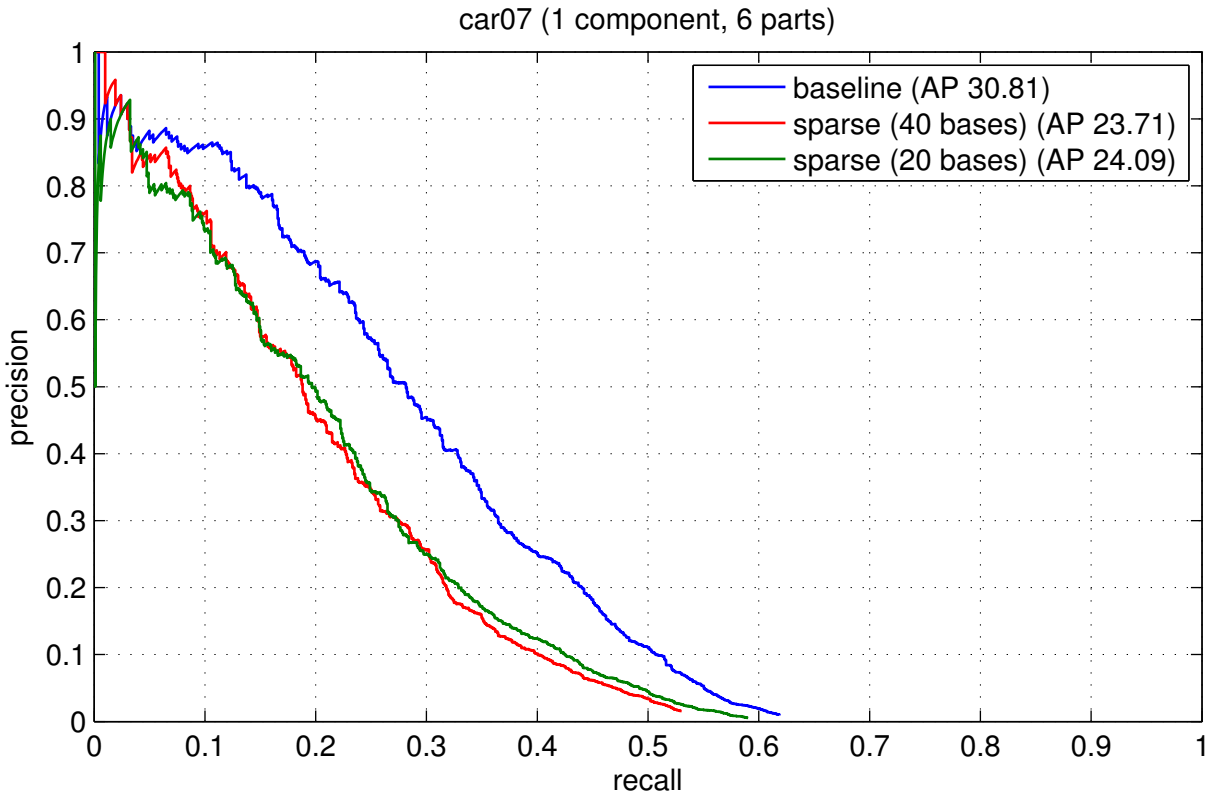


Figure 6.1: Precision-recall curves for the `car07_1` model.

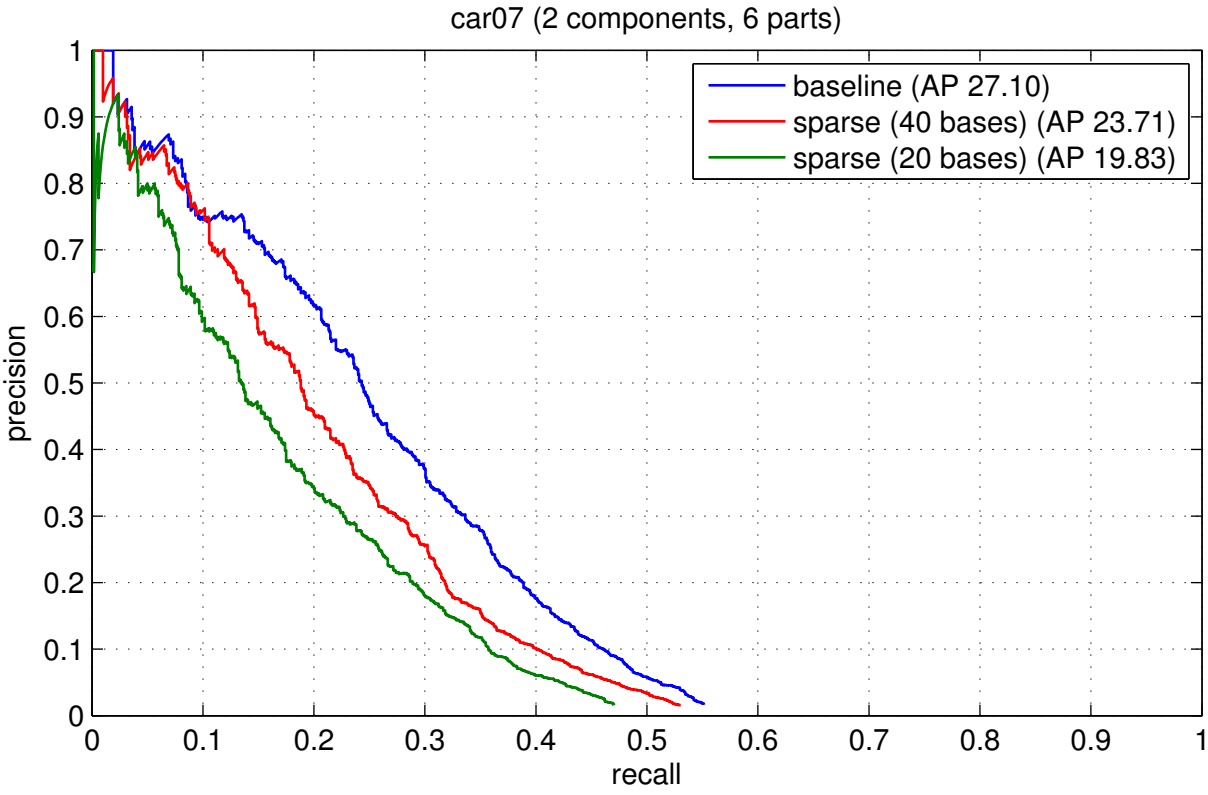


Figure 6.2: Precision-recall curves for the `car07_2` model.

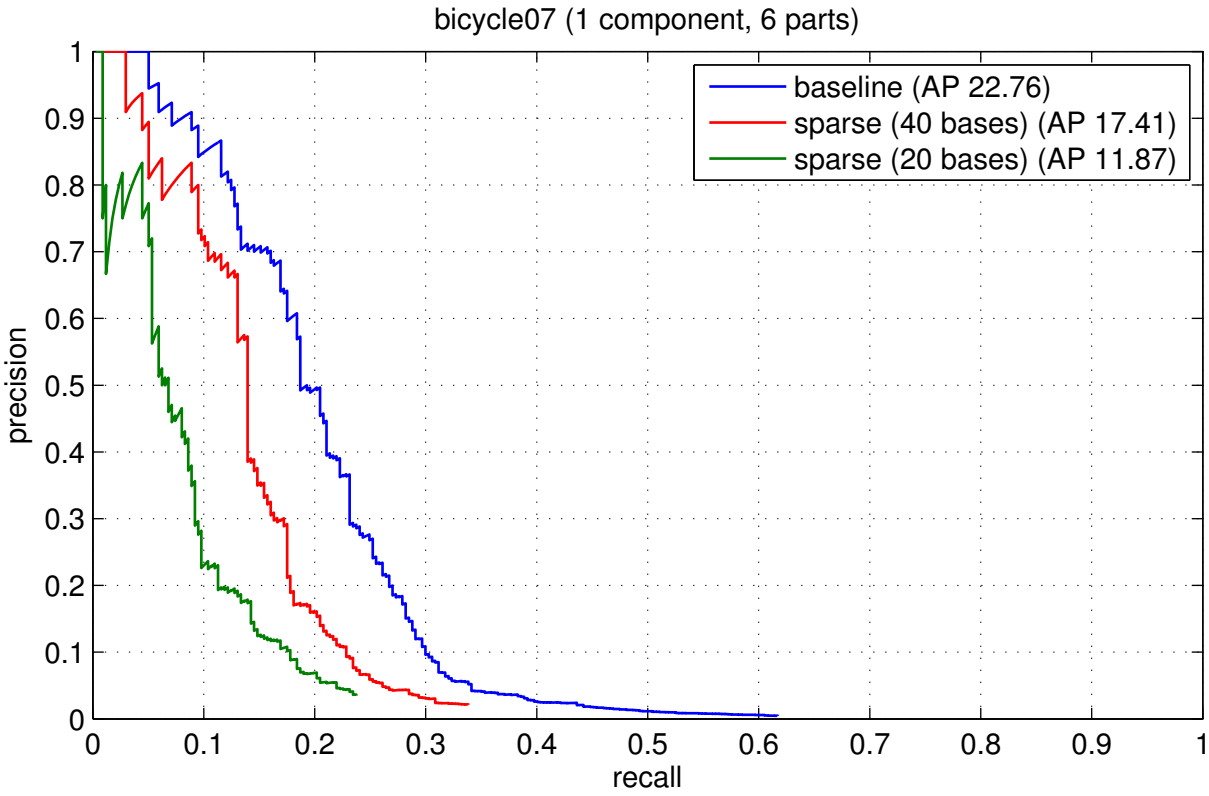


Figure 6.3: Precision-recall curves for the `bicycle07_1` model.

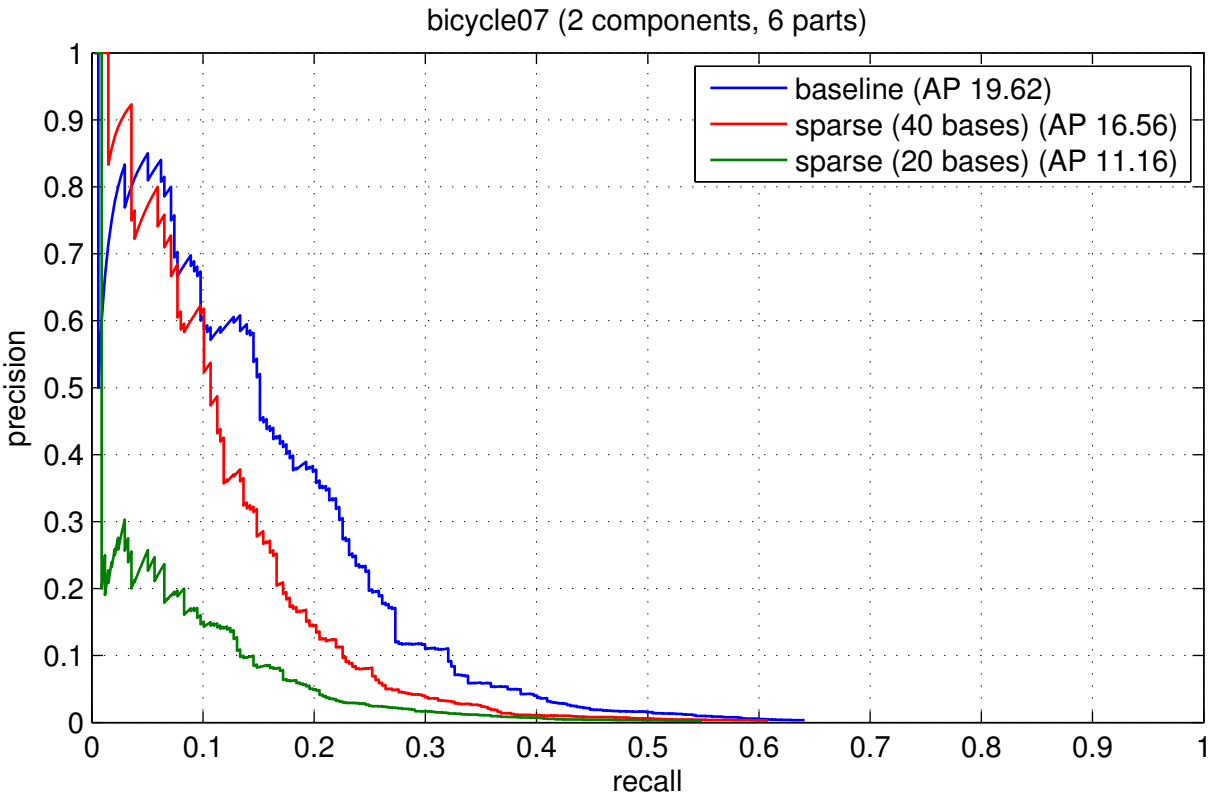


Figure 6.4: Precision-recall curves for the `bicycle07_2` model.

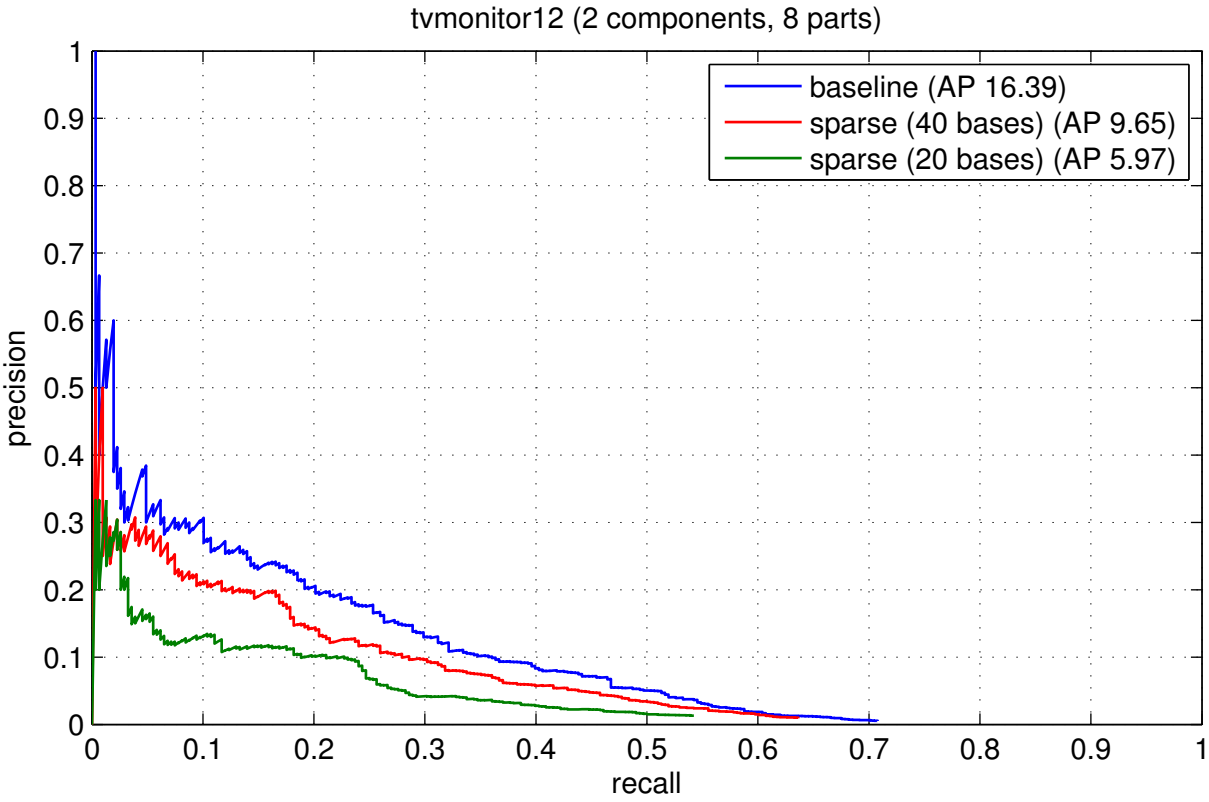


Figure 6.5: Precision-recall curves for the `tvmonitor12` model.

Detection type	Total number of parts	Elapsed time
`baseline`	122	17h 36min 37s
`sparse (40 bases)`	40	7h 49min
`sparse (20 bases)`	20	4h 25min 34s

Table 6.2: Elapsed times for the three detection tests, performed with 10 models on the 4,952 images of the VOC 2007 test set.

6.5 Speed Assessment

We have measured the time elapsed to compute detections on the 4,952 images of the VOC 2007 test set. The values of the three detection tests are shown in table 6.2.

To keep the same context, the speed tests were done with the same model set used for accuracy evaluation. However, unlike the previous tests, in which we evaluated each model separately, in this case we need to do multi-class detection with both `dpmdetect` and `dpmsparsedetect`.

Therefore, for each image we have 122 part filters to be convolved with `dpmdetect`, whereas with `dpmsparsedetect` we convolve only 40 or 20 of them. Because of this, the speed gain is apparent even with our naïve implementation, that does not make use of multi-processing or sophisticated code optimisations. Just reducing the number of convolution operations produces a noticeable speedup, while the detection accuracy is not particularly compromised, as we have seen in section 6.4.

Furthermore, we wanted to compare the performance of `dpmdetect` to that of `dpmsparsedetect`, so we generated with Gprof2Dot [19] the profile graphs shown in figures 6.6 and 6.7. These analyse the execution of the two detection programs using a set comprising 5 models with 3×8 parts each (120 parts in total). The tests were done on the whole VOC 2007 test set.

We can immediately notice that the impact of convolution is halved: we have around half of

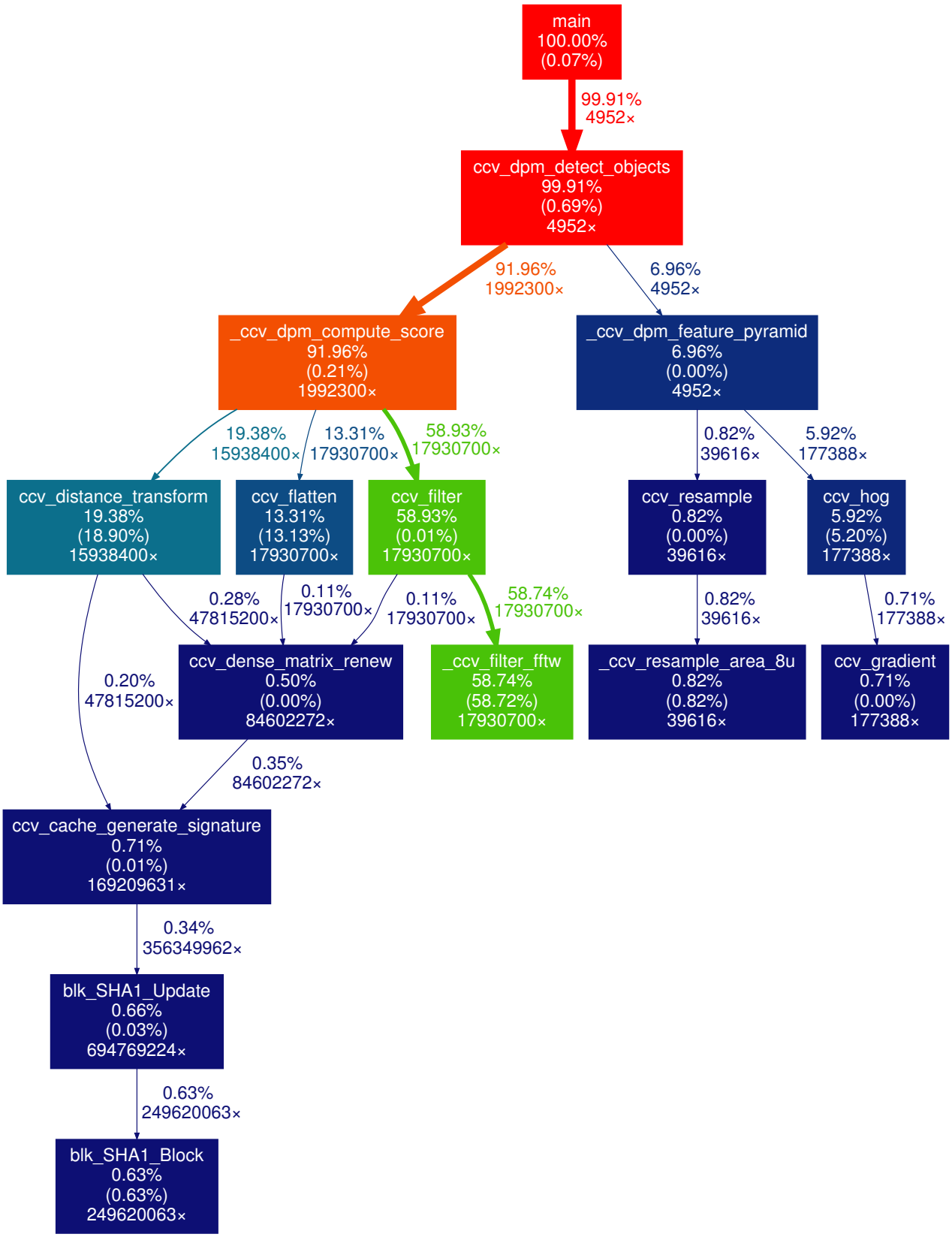


Figure 6.6: DOT graph of `dpmdetect` profile data. Detection of 5 classes comprising a total of 120 part filters on the 4,952 test images of the VOC 2007 dataset.

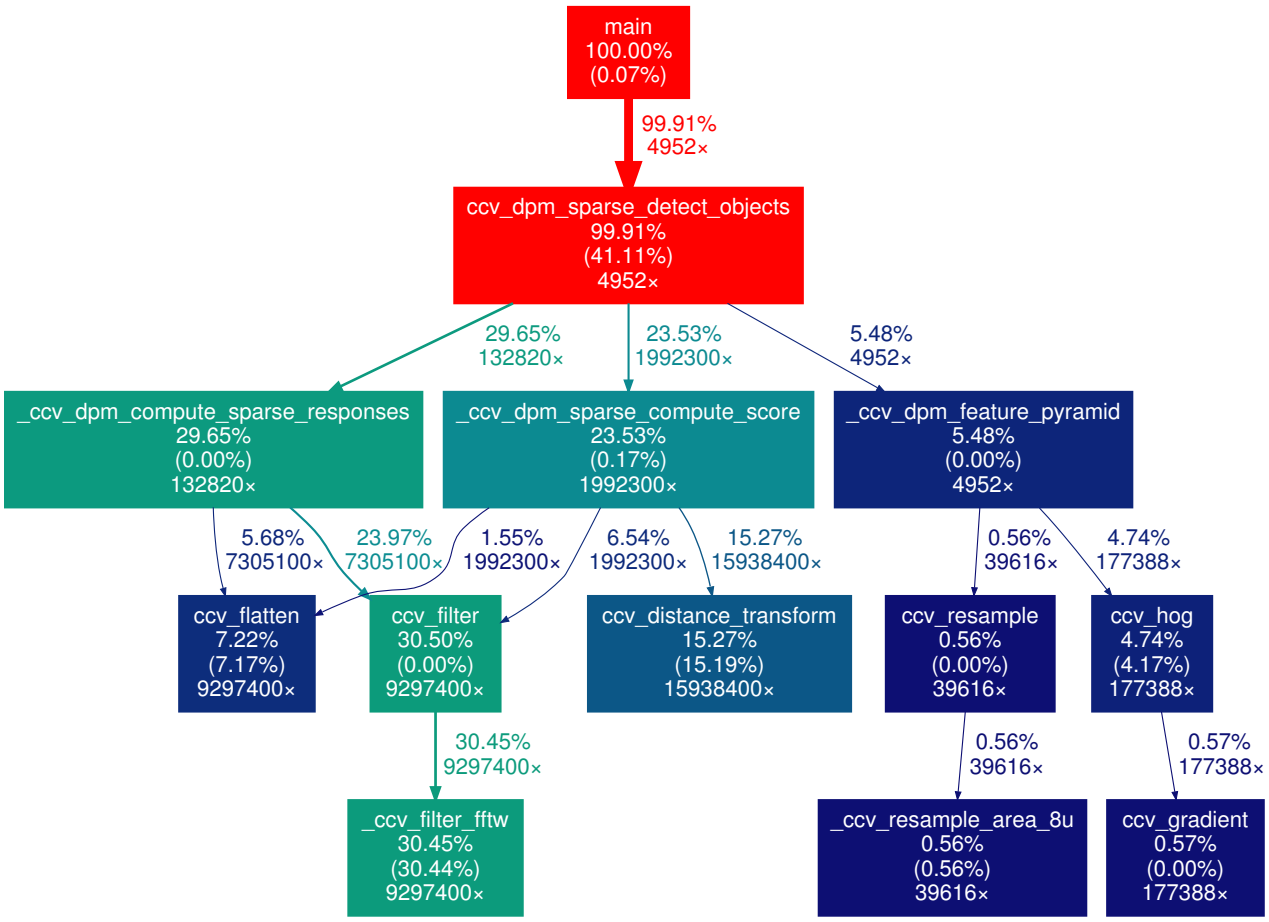


Figure 6.7: DOT graph of `dpmsparsedetect` profile data. Detection of 5 classes comprising a total of 60 sparse filters on the 4,952 test images of the VOC 2007 dataset.

the function calls to `ccv_filter` and half of the total time percentage is spent in that function.

The number of calls to `ccv_filter` in the `dpmsparsedetect` test corresponds to that of `ccv_flatten`, because these are called together in the same context (`_ccv_dpm_compute_sparse_responses`). If instead we look at the number of calls to almost any other function (`ccv_distance_transform`, `_ccv_dpm_feature_pyramid`, `ccv_hog`, `ccv_resample`) we notice that they are the same in both tests, because they all concern operations that are independent from the reconstruction of sparse filters, and hence have to be executed before or after it.

The number of calls to the `_ccv_dpm_sparse_compute_score` function also corresponds to that of `_ccv_dpm_compute_score`, because these are both functions in which the detection scores are computed, and they are computed the same number of time in both tests. The difference, as shown by their percentage time values, is that the first is much lighter, since the part responses have been already pre-computed, so only the root ones and distance transforms are needed, while the second has to compute everything and in fact it represents the heaviest computational block of the whole process.

The runtime tests were made on a 3.40GHz Intel Core i7-4930K CPU, and the code was compiled using `gcc` 4.7.3 with the following compilation flags:

```
CFLAGS := -O3 -ffast-math -Wall -msse2
```

The same settings were used for the profiling tests, except for the optimisation flag which was set to `-O0` to get more accurate data, and of course with the addition of the `-pg` flag.

6.6 Summary

We have described the tools used to evaluate the detection accuracy of DPM in CCV and `voc-release5`. Using some of them, we have carried out an investigation to measure the efficiency of an integration of sparse coding into CCV's DPM detection code.

We have selected a set of 10 models and tested them on the VOC 2007 test set, comparing the average precision results obtained from the original implementation (`dpmdetect`) with those of our sparse coding one (`dpmsparsedetect`), with dictionaries of 40 and 20 bases to approximate a total of 122 part filters.

Finally, we have assessed the speed gain achieved by the sparse coding approach, which is evident even in our CPU implementation, and could be exponentially greater in a GPU one, as reported in [40].

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Analysing the results of our integration of sparse coding for the DPM implementation included in the CCV library, we can summarise the contributions of this project in the following points:

- We have described the state-of-the-art variants of DPM, emphasising their strong points and their drawbacks with regard to accuracy and speed. We have shown how the DPM framework has evolved through the years to finally achieve real-time performance on GPU with the sparselet method of [40] (chapter 3).
- We have analysed the traits of the DPM implementation in CCV, carrying out a detailed breakdown of the model training and object detection processes. We have explained the importance of this analysis in view of an integration of sparse coding in this framework (chapter 4).
- We have managed to use the matrix factorisation and decomposition functions provided in SPAMS in the DPM object detection implementation included in CCV by merging the two code bases in a neat and efficient way (chapter 5).
- We have evaluated our sparse coding implementation in the CCV context, showing that with a sample set of 10 models we managed to maintain the AP, and that the speed of this implementation is substantially higher than that of the original detection program (chapter 6).

7.2 Further Work

Our implementation could be improved from several points of view. In the following subsections, we list some things that could be done to advance the work presented in this report.

7.2.1 Parallel Implementation

The DPM framework is well-suited to a parallel implementation, because there are several computational blocks that are independent from each other. For example, these operations could be executed independently:

- computation and search over different scales of a feature pyramid;
- search over different sub-windows of the image;
- evaluation of different models;
- evaluation of different model components.

Similarly, other blocks can be executed in parallel and synchronised at the end of the subsection of the process in which they are involved. For instance, part filters responses can be reconstructed from sparse filters in parallel, and then when all the needed responses are ready (synchronisation) they can be used to compute distance transforms.

Essentially, almost every part of the detection process can be parallelised. As we have seen in section 3.5, the GPU implementation of [40] has succeeded in achieving real-time performance.

7.2.2 Updates to CCV's DPM Implementation

As we have mentioned before, the DPM implementation in CCV is based on the older `voc-release3` version of classical DPM. Upgrading it to make it compliant with `voc-release5` should be taken into account for a future version of the library. For example, the following things could be improved:

- The algorithm used to do NMS should be substituted with that used in `voc-release5`, which grants better AP results.
- The user should be allowed to set certain parameters that are currently hard-coded, like the threshold that determines if a detection is accepted or discarded, or the flag that specifies if and when NMS should be done.
- The drawing and validation tools should allow more user customisation too, since in the current implementation they have a very limited range of application.
- The training process should be updated following the structure used in `voc-release5`, which makes also use of `parallel for` (`parfor`) loop constructs, which are part of the MATLAB parallel computing toolbox and are similar to those used in the OpenMP API [3]. It should be analysed how they are employed in the MATLAB code with the aim of replicating the same kind of parallelisation in CCV. This could improve significantly the training speed.
- Model files should include more information, like the optimal detection threshold and the class label. Again, `voc-release5` models and their structure could be taken as a reference point.

The last point is particularly important in a multi-class detection context. Object labelling implies that we can specify which object classes we want to detect using class names, instead of the model index in the list as we were forced to do in our implementation (see section 5.3.1). This would reduce the level of abstraction for the user.

7.2.3 Evaluation with Accurate Models

As we have explained in section 6.3, we could not employ very good models for our tests, mainly due to lack of time. Therefore, to make a better assessment of this implementation, it should be tested using accurate models. This should confirm that the AP is successfully maintained when using sparse coding for DPM detection, thus giving more confidence about the advantages of this framework.

The models could also be trained and tested on newer, richer and more challenging datasets than the VOC ones, like the ImageNet database [7].

7.3 Final Remarks

Through this investigation, we have learnt that a deep and complex framework like DPM has still room for improvement, from both an accuracy and a speed point of view. We think that further research should be conducted in this field, because it has the potential to achieve an efficient, reliable and real-time object detection.

In conclusion, we hope that what we have implemented in this project provides a useful base that can be used in research contexts aiming to enhance the performance of computer vision applications, like the PAMELA research programme [37].

Appendix A

Data Structure Definitions in `ccv.h`

The definitions of data structures that are often mentioned in this report are listed in the following subsections. The first groups data structures of general interest that are not used only in DPM but also in other algorithms implemented in CCV. The second comprises data structures that are used only in the DPM context.

A.1 General-Purpose Data Structures Used by DPM

```
typedef union {
   unsigned char* u8;
   int* i32;
   float* f32;
   int64_t* i64;
   double* f64;
} ccv_matrix_cell_t;

typedef struct {
   int type;
   uint64_t sig;
   int refcount;
   int rows;
   int cols;
   int step;
   union {
      unsigned char u8;
      int i32;
      float f32;
      int64_t i64;
      double f64;
      void* p;
   } tag;
   ccv_matrix_cell_t data;
} ccv_dense_matrix_t;

typedef struct {
   int width;
   int height;
} ccv_size_t;
```

```
typedef struct {
   int x;
   int y;
   int width;
   int height;
} ccv_rect_t;

typedef struct {
   int type;
   uint64_t sig;
   int refcount;
   int rnum;
   int size;
   int rsize;
   void* data;
} ccv_array_t;
```

A.2 Specific Data Structures Pertaining to DPM

```
typedef struct {
   int id;
   float confidence;
} ccv_classification_t;

typedef struct {
   ccv_rect_t rect;
   int neighbors;
   ccv_classification_t classification;
} ccv_comp_t;

typedef struct {
   ccv_rect_t rect;
   int neighbors;
   ccv_classification_t classification;
   int pnum;
   ccv_comp_t part[CCV_DPM_PART_MAX];
} ccv_root_comp_t;

typedef struct {
   ccv_dense_matrix_t* w;
   double dx, dy, dxx, dyy;
   int x, y, z;
   int counterpart;
   float alpha[6];
} ccv_dpm_part_classifier_t;

typedef struct {
   int count;
   ccv_dpm_part_classifier_t root;
   ccv_dpm_part_classifier_t* part;
   float alpha[3], beta;
} ccv_dpm_root_classifier_t;
```

```
typedef struct {
   int count;
   ccv_dpm_root_classifier_t* root;
} ccv_dpm_mixture_model_t;

typedef struct {
   int interval;
   int min_neighbors;
   int flags;
   float threshold;
} ccv_dpm_param_t;

typedef struct {
   int components;
   int parts;
   int grayscale;
   int symmetric;
   int min_area; // 3000
   int max_area; // 5000
   int iterations;
   int data_minings;
   int root_relabels;
   int relabels;
   int discard_estimating_constant; // 1
   int negative_cache_size; // 1000
   int square_filter_size; // 0
   double include_overlap; // 0.7
   double alpha;
   double alpha_ratio; // 0.85
   double balance; // 1.5
   double C;
   double percentile_breakdown; // 0.05
   ccv_dpm_param_t detector;
} ccv_dpm_new_param_t;
```

Appendix B

Comparison Tables

These tables are put here for completeness and for reference in the context of the evaluation presented in chapter 6.

Model	**voc-release5**	**CCV**
bicycle (2007)	17.50	15.22
bus (2012)	4.68	1.72
car (2007)	27.71	20.37
person (2007)	1.90	1.34

Table B.1: Comparison of the NMS impact on AP when performed in CCV and in `voc-release5`. The models in this table are very weak ones that were discarded during the evaluation tests.

Name	$C \times P$	**Baseline AP**	**Sparse40 AP**	**Sparse20 AP**
`aeroplane07`	1×8	9.30	9.20	0.48
`bicycle07_1`	1×6	22.76	17.41	11.87
`bicycle07_2`	2×6	19.62	16.56	11.16
`bus07`	1×8	6.64	2.47	2.79
`car07_1`	1×6	30.81	23.71	24.09
`car07_2`	2×6	27.10	23.71	19.83
`motorbike12`	2×8	9.82	9.60	9.33
`person07_1`	1×6	11.72	4.67	4.03
`person12_3`	3×8	10.82	10.52	4.03
`tvmonitor12`	2×8	16.39	9.65	5.97

Table B.2: AP comparison of baseline and sparse detections on the 10 models selected for our evaluation. C: number of model components; P: number of parts per component.

Class	**Recall**	**FA**	I	P
aeroplane	3.22	201	205	331
bicycle	35.73	590	250	418
bird	16.32	3468	289	599
boat	10.18	814	176	398
bottle	3.04	870	240	634
bus	21.65	486	183	272
car	33.42	1429	775	1644
cat	8.65	1611	332	389
cow	4.68	868	127	356
dog	14.34	1724	433	538
horse	10.13	239	279	406
motorbike	8.94	288	233	390
pottedplant	23.31	2711	254	625
sofa	7.07	407	355	425
train	7.28	359	259	328
tvmonitor	17.17	549	255	367

Table B.3: Comparison of the results obtained by some models evaluated using CCV metrics (`dpmvldtr`), which comprises the recall percentage and the number of false positives (FA). These models were trained on the VOC 2007 `trainval` set and evaluated on a subset of the VOC 2007 `test` set, comprising only images containing instances of the class of interest. The models comprise either a single component or 2 components, and 6 parts per component. I: total number of images used in the test. P: total number of positive instances in all the images considered.

Bibliography

[1] Yotam Abramson, Bruno Steux, and Hicham Ghorayeb. Yet Even Faster (YEF) real-time object detection. *International Journal of Intelligent Systems Technologies and Applications*, 2(2/3):102–112, February 2007.

[2] Rodrigo Benenson, Markus Mathias, Tinne Tuytelaars, and Luc Van Gool. Seeking the strongest rigid detector. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3666–3673. IEEE, 2013.

[3] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://openmp.org/. Accessed September 1, 2014.

[4] Mark Borgerding. Kiss FFT. http://kissfft.sourceforge.net/. Accessed September 1, 2014.

[5] Gary Bradski. The OpenCV library. *Dr. Dobb's Journal of Software Tools*, 25(11):120–126, 2000.

[6] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In Cordelia Schmid, Stefano Soatto, and Carlo Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition*, volume 2, pages 886–893, INRIA Rhône-Alpes, ZIRST-655, av. de l'Europe, Montbonnot-38334, June 2005.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[8] Piotr Dollár, Zhuowen Tu, Pietro Perona, and Serge Belongie. Integral channel features. In *BMVC*. British Machine Vision Association, 2009.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[10] Boris Epshtein, Eyal Ofek, and Yonatan Wexler. Detecting text in natural scenes with stroke width transform. 0:2963–2970, 2010.

[11] Mark Everingham, Luc Van Gool, C. K. I. Williams, J. Winn, and Andrew Zisserman. The PASCAL Visual Object Classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[12] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, June 2008.

[13] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[14] Pedro F. Felzenszwalb, Ross B. Girshick, and David A. McAllester. Cascade object detection with deformable part models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2241–2248. IEEE, June 2010.

[15] Pedro F. Felzenszwalb, Ross B. Girshick, David A. McAllester, and Deva Ramanan. Discriminatively trained deformable part models, release 3. http://cs.brown.edu/~pff/latent-release3/. Accessed September 1, 2014.

[16] Pedro F. Felzenszwalb, Ross B. Girshick, David A. McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[17] Pedro F. Felzenszwalb, David A. McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.

[18] Martin A. Fischler and Robert A. Elschlager. The representation and matching of pictorial structures. *IEEE Transactions on Computers*, C-22(1):67–92, January 1973.

[19] José Fonseca. Gprof2Dot: Convert profiling output to a dot graph. https://code.google.com/p/jrfonseca/wiki/Gprof2Dot. Accessed September 1, 2014.

[20] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. volume 93, pages 216–231, Feb 2005.

[21] Free Software Foundation (FSF). GNU Scientific Library. http://www.gnu.org/software/gsl/. Accessed September 1, 2014.

[22] Free Software Foundation (FSF). GOMP: An OpenMP implementation for GCC. https://gcc.gnu.org/projects/gomp/. Accessed September 1, 2014.

[23] Ross B. Girshick, Pedro F. Felzenszwalb, and David A. McAllester. Discriminatively trained deformable part models, release 5. http://people.cs.uchicago.edu/~rbg/latent-release5/. Accessed September 1, 2014.

[24] Luc Van Gool, Markus Mathias, Radu Timofte, and Rodrigo Benenson. Pedestrian detection at 100 frames per second. 0:2903–2910, 2012.

[25] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2010.

[26] Chang Huang, Haizhou Ai, Yuan Li, and Shihong Lao. High-performance rotation invariant multiview face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):671–686, 2007.

[27] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1409–1422, July 2012.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, pages 1106–1114, 2012.

[29] Liu Liu. Application driven philosophy. http://libccv.org/post/application-driven-philosophy/. Accessed September 1, 2014.

[30] Liu Liu. C-based/Cached/Core Computer Vision library. http://libccv.org/. Accessed September 1, 2014.

[31] Liu Liu. Call for a new, lightweight, c-based computer vision library. http://libccv.org/post/call-for-a-new-lightweight-c-based-computer-vision-library/. Accessed September 1, 2014.

[32] Liu Liu. Dpm: Deformable parts model. http://libccv.org/doc/doc-dpm/. Accessed September 1, 2014.

[33] Liu Liu. An elephant in the room. http://libccv.org/post/an-elephant-in-the-room/. Accessed September 1, 2014.

[34] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[35] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research (JMLR)*, 11:19–60, March 2010.

[36] Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. *International Conference on Pattern Recognition*, 3:850–855, 2006.

[37] The University of Manchester. PAMELA project webpage. http://apt.cs.manchester.ac.uk/projects/PAMELA/. Accessed September 1, 2014.

[38] Marco Pedersoli, Andrea Vedaldi, and Jordi Gonzàlez. A coarse-to-fine approach for fast deformable object detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1353–1360. IEEE, 2011.

[39] Alan F. Smeaton, Paul Over, and Wessel Kraaij. Evaluation campaigns and TRECVid. In *MIR '06: Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval*, pages 321–330, New York, NY, USA, 2006. ACM Press.

[40] Hyun Oh Song, Stefan Zickler, Tim Althoff, Ross B. Girshick, Mario Fritz, Christopher Geyer, Pedro F. Felzenszwalb, and Trevor Darrell. Sparselet models for efficient multiclass object detection. In Andrew W. Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid, editors, *ECCV (2)*, volume 7573 of *Lecture Notes in Computer Science*, pages 802–815. Springer, 2012.

[41] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *Computing Research Repository (CoRR)*, abs/1311.2901, 2013.

[42] Mu Zhu. Recall, precision and average precision. Technical report, Department of Statistics & Actuarial Science, University of Waterloo, 2004.