Object Detection with Heuristic Coarse-to-Fine Search

Ross Girshick

May 29, 2009
Abstract

We consider the task of localizing and labeling instances of a generic object class within real-world images. Our focus is on a generalized class of pictorial structure models that are defined in terms of visual grammars. In particular, we address the challenging problem of performing detection efficiently even as model complexity grows within this class. Our proposed solution is a blend of heuristic best-first search and a coarse-to-fine detection process. This paper demonstrates that our algorithm can be successfully applied to two special cases of visual grammars: multiscale star models and mixtures of multiscale star models. We show that for problems where the desired output is the local optima of a thresholded function, best-first search gives additional pruning power to coarse-to-fine processes. Unfortunately, admissible heuristics that also provide good best-first search behavior can be difficult or impossible to find in practice. To resolve this deficiency, we provide theoretical results demonstrating that inadmissible heuristics can be used to increase detection speed while only slightly increasing the likelihood of suffering mistakes. The theoretical results are bolstered by strong experimental evidence obtained by applying inadmissible heuristic coarse-to-fine detection to our object recognition system during both training and testing. We increase testing speed by a factor of 2-3 for some classes while maintaining comparable average precision scores on the challenging PASCAL 2007 dataset. Ultimately we expect to see even more significant speed gains when we explore more complex grammar models in future work.

Acknowledgements

I would like to thank everyone who has been patient with me while I’ve absorbed myself in this paper. I owe more than I can state to Meera for keeping me sane and being my companion on this journey. I’m indebted to Pedro for being my mentor and introducing me to the fascinating problem of object recognition. Many of the ideas in this paper were developed in close collaboration with him. This is also joint work with David McAllester and Deva Ramanan.

This material is based upon work supported by the National Science Foundation under Grant No. 0534820 and 0746569.
## Contents

1 Introduction .................................................. 5
   1.1 Matching Pictorial Structure Models ......................... 9
   1.2 The Model .................................................. 10
       1.2.1 The Configuration Set $L^n$ .......................... 11
       1.2.2 Multiscale Star Models .............................. 12
   1.3 Matching Pictorial Structure Models by Dynamic Programming .......... 14
   1.4 From Dynamic Programming to Heuristic Coarse-to-Fine Search ........... 16

2 Heuristic Coarse-to-Fine Matching ................................ 17
   2.1 Coarse-to-Fine Detection ...................................... 17
   2.2 Heuristic Coarse-to-Fine Detection ............................ 21
   2.3 Optimality of Heuristic Coarse-to-Fine Detection ................. 22
   2.4 Efficient Computation ....................................... 24
   2.5 Non-maximal Suppression ...................................... 25
   2.6 Related Work ................................................. 26
   2.7 Discussion and Connection to AO* Search in AND/OR Graphs ............ 28

3 Determining Thresholds, Heuristics, and Part Order ................. 30
   3.1 A Simple Statistical Framework ................................ 31
   3.2 Probably Approximately Admissible Heuristics .................. 32
   3.3 Probably Approximately Admissible Thresholds .................. 35
   3.4 Determining Heuristics and Thresholds ........................ 36
## 4 Discriminatively Trained Mixtures of Multiscale Star Models

4.1 Mixture Models .............................................. 40
4.2 Feature Map .................................................. 41
4.3 Training Mixture Models .................................... 42
   4.3.1 Model Parameterization and Feature Vectors ............... 42
   4.3.2 Weakly-Labeled Training Data ............................. 43
   4.3.3 Latent SVM ............................................ 44
   4.3.4 Initialization and Training Phases .......................... 47

## 5 Training Mixture Models with Heuristic Coarse-to-Fine Matching

5.1 Mixture Models and Heuristic Coarse-to-Fine Detection .......... 50
5.2 Initialization .................................................. 51
5.3 Updating Heuristics, Thresholds, and Part Order .................. 51
5.4 Fixed Root Filters .......................................... 53
5.5 Data Mining with Non-maximal Suppression ....................... 54

## 6 Experimental Results

6.1 Datasets ..................................................... 55
6.2 Protocol ....................................................... 56
6.3 Discussion of Findings ...................................... 57

## 7 Conclusions

62
Chapter 1

Introduction

In this paper, we will investigate algorithms that find objects in real-world images. Rather than recognizing a specific object, such as your trusty bicycle, we consider the problem of detecting entire object classes: find all bicycles in an image. This task raises fundamental questions about how we can represent the visual richness of the world. Many techniques have been developed, each placing varying degrees of emphasis on factors such as shape, texture, and part decomposition. At this point there is no clear answer to the question of what the ideal representation looks like.

Our approach to this highly active area of computer vision research originates from the pictorial structure model that was introduced more than 30 years ago in the pioneering work of Fischler and Elschlager [24]. Their framework is simple to grasp on an intuitive level: an object class is represented by some parts that are connected by springs. A model for bicycles, for example, might specify the parts as: two wheels, a seat, and a handle bar. The parts’ canonical spatial layout is enforced by placing springs between some pairs of parts. To see how we would detect bicycles with our model, we suppose that each spring has some cost associated with stretching it, and that we can measure how well each part matches an image at a particular location. Now we can drop our model onto an image and wiggle the parts around until we find a spatial configuration that strikes an optimal balance between matching the parts and stretching the springs. We call the task of finding an optimal configuration of a model in an image the matching problem for pictorial structures.

Ultimately the pictorial structure model is a bit simplistic. Returning to our running example of bicycles, if you consider how they appear in real-world images, you will find an amazing degree of variation. People
build bicycles in a wide range of sub-classes (e.g., mountain bikes, tandems, and 19th-century cycles with one big wheel and one small wheel) and view them in various poses (e.g., frontal versus side views). The phenomenon of intra-class variation is one of the most challenging aspects of generic object class detection. We believe that a natural approach for dealing with this variation is to consider a richer class of models that is defined in terms of a visual grammar. Grammar models generalize pictorial structure models in a number of important ways. In particular, they model objects where parts are defined recursively in terms of other parts, and where object structure is variable, such as varying part composition with changes in pose and sub-class. Grammar models also provide a natural way to share information and computation between object classes by allowing models to share parts, or even common substructures.

Our current research stands along the path from simple pictorial structure models to rich grammar models. Moving from simpler models to richer models, while maintaining or improving performance, is surprisingly difficult. One reason is that learning rich models often requires judicious use of hidden or latent information, whereas simple models can often be learned directly from fully-labeled data. For instance, our system learns part-based models using only positive bounding boxes – we have no direct information telling us how many parts an object class has, where the parts are located in each positive example, or how they are shaped and what they look like. In contrast, one can easily learn a rigid template detector directly from positive bounding boxes. We use a discriminative training formalism called Latent SVM (LSVM) to learn structured models using hidden information [17].

We first consider, in Chapter 1, a special form of pictorial structure model that has a star topology and integrates image evidence from multiple scales. Already this model can be viewed as a simple 1-level grammar model with a fixed part composition. Later, in Chapter 4, we will introduce mixtures of these multiscale star models, which can be thought of as 2-level grammar models, where the first production chooses between several simpler models. The mixture model system described in Chapter 4 recently tied for first place in the very difficult 2008 PASCAL Visual Object Categories Challenge [15].

Another obstacle in the path toward grammar models is the need for an efficient matching algorithm. Even in the case of simple pictorial structure models we immediately run into a problem: for arbitrary combinations of parts and springs, matching is NP-hard [5]. Fortunately, as we will see shortly, we can place a few natural restrictions on these models and in return receive an efficient algorithm. This algorithm,
which we review in Chapter 1, is due to Felzenszwalb and Huttenlocher [20], and has served the purpose of matching simple models admirably. In the context of grammar models, however, it is clear that this algorithm will be too slow. The primary efficiency issue is that for grammar models the algorithm must exhaustively test how well each part matches the image. Even though this procedure is efficient in the sense that its running time grows linearly in the number of parts, in practice there is a large constant factor associated with computing part scores, which forces us to use a small number of parts (e.g., 14) in practice.

Failure to properly overcome difficulties encountered in learning and efficient computation has resulted in the phenomenon that simpler models often achieve better results on benchmarks than models that seem conceptually stronger. We believe that in order to successfully develop rich models for object detection, we must develop them incrementally so that any challenges that arise along the way can be solved in isolation. Therefore we have adopted a research methodology of taking small steps, while maintaining the same high level of detection accuracy and computational efficiency along the way.

This paper is about taking a step in the direction of an efficient matching algorithm for grammar models. We believe that a promising approach to this problem can be found in the diverse class of heuristic search algorithms [31]. In particular, we also believe that to perform detection fast it will be necessary to relax our algorithm’s dependence on admissible heuristics. We present a heuristic coarse-to-fine matching algorithm for multiscale star models and mixtures of multiscale star models that successfully uses inadmissible heuristics at training and testing time, and achieves detection accuracy that is comparable to the exact, but slower, matching algorithm. At our current point on the path from simple to rich models we already find that the new algorithm increases detection speed by a factor of 2 to 3 for some object classes, though we don’t expect to realize the true benefits until we generalize our algorithm to work with rich grammar models. We develop our approach in several steps:

1. We formulate a coarse-to-fine algorithm for solving the matching problem in the case of multiscale star models (Chapter 2). This algorithm exploits the fact that most images contain only a few promising locations for matching a model.

2. Then we augment the coarse-to-fine algorithm with a form of heuristic search. This novel heuristic coarse-to-fine detection algorithm allows us to explore the sparse search space in best-first order.

3. To make our algorithms faster, we demonstrate a very simple method for selecting inadmissible
coarse-to-fine thresholds and heuristics (Chapter 3). Leveraging inadmissibility is potentially dangerous since we may no longer find optimal matches. We address this concern with theoretical bounds on the error rate due to inadmissibility, and provide empirical results for the challenging PASCAL 2007 dataset that demonstrate the validity of our approach.

4. In Chapter 4 we take a small step toward rich grammar models by introducing mixtures of multiscale star models. We describe them in the context of a complete object recognition system that achieves state-of-the-art results for several object classes in the PASCAL Challenge (2006, 2007 & 2008). In Chapters 5 & 6 we demonstrate that by a simple extension, our heuristic coarse-to-fine algorithm will work with mixture models and yield faster performance without significant loss in detection accuracy.

5. Throughout the paper, with an eye toward the future, we develop an explicit connection between heuristic coarse-to-fine detection and AO* search in AND/OR graphs. This connection is the algorithmic mechanism that will allow us to extend our algorithms to work with more general grammar models [21].

The heuristic coarse-to-fine detection algorithm presented in Chapter 2 has a strong connection to Amit, Fleuret, and Geman’s research on coarse-to-fine algorithms [26], [3], [2], and [25]. In their work, the overall goal of scene understanding is split into two phases. First, a non-contextual phase generates hypothesized detections for the object classes of interest. Next, a contextual phase refines the hypotheses by, for example, disambiguating uncertain detections and eliminating multiple detections of the same object. The output of the contextual phase is the most likely ensemble interpretation of the image.

Our heuristic coarse-to-fine detection algorithm blurs this two-phase distinction to some degree. We exploit the fact that (typically) only a single instance of a given object class can occupy the same region of an image. In a sense, multiple detections of the same class create an intra-class context. We incorporate this single-occupancy contextual rule, often employed as a post-processing step called non-maximal suppression, directly in our coarse-to-fine algorithm by using a heuristic function to explore partial object hypotheses in best-first order. To see when best-first search is most effective, imagine an image that is almost entirely filled with a bus or human face. Applied to such an image, heuristic coarse-to-fine search will quickly find this large-scale object and immediately eliminate nearly all other hypotheses. This may seem like a small gain (if one at all) since a typical image will not contain any instances of a fixed object class. But from a broader
perspective, our ultimate goal is scene understanding and hence we are imagining the eventual usage of heuristic coarse-to-fine search for multiclass detection, where instances from a potentially large set of object classes are detected jointly. In this situation, a typical image will contain some instances of interest, and these instances will allow the algorithm to eliminate all competing object class hypotheses according to a reasonable rule for inter-class non-maximal suppression.

1.1 Matching Pictorial Structure Models

In our discussion, $M$ will denote a pictorial structure model. We can define $M$ naturally in terms of the undirected graph $G_M = (V, E)$, where $V$ is a set of vertices labeled by $V = \{v_1, \ldots, v_n\}$, and $E \subseteq V \times V$ is a set of edges. Vertices represent parts and edges represent springs. By abuse of notation, we will use the vertex label $v_i$ to refer to the $i$-th part of $M$ as well as the $i$-th vertex of $G_M$. The set $E$ is defined so that $(v_i, v_j) \in E$ if and only if $M$ has a spring between $v_i$ and $v_j$.

In the pictorial structures literature, matching is usually formalized as an energy minimization problem. It will be more natural for us to cast this problem as score maximization, where our score function is simply the negative of the conventional energy function. Score is a function of three parameters: a model $M$, an image, and a configuration of $M$ in the image. A configuration of $M$ in an image is a list of locations $L = (l_1, \ldots, l_n) \in \mathcal{L}^n$, where $l_i$ specifies the position of $v_i$ from some finite set of valid part locations $\mathcal{L}$. The set of part locations typically depends on the size of the image. Our score function for matching $M$ in $\mathcal{L}^n$ is

$$S_M(L) = \sum_{i=1}^{n} m_i(l_i) - \sum_{(v_i, v_j) \in E_{i<j}} d_{ij}(l_i, l_j),$$

(1.1)

where $m_i(l_i)$ measures $i$'s compatibility with the image data at location $l_i$, and $d_{ij}(l_i, l_j)$ is the cost of deforming the spring between parts $i$ and $j$ when they are placed at locations $l_i$ and $l_j$. The score function’s dependence on the image is hidden in the definition of $m_i$ and the configuration set $\mathcal{L}^n$. We almost always consider the matching problem with respect to some implicit image, so this parameterization is omitted to simplify notation. A configuration that maximizes equation (1.1) is a solution to the matching problem.

Maximizing the score function for arbitrary graphs is NP-hard (see [5]), but under certain very natural
conditions for $G_M$’s topology and the form of the function $d_{ij}$, efficient algorithms are known. For instance, if the graph is acyclic and $d_{ij}$ is squared Euclidean distance, then equation (1.1) can be maximized in $O(kn)$ time using dynamic programming and an efficient algorithm for generalized distance transforms [20]. Here $n$ is the number of parts, and $k = |\mathcal{L}|$ is the number of locations that each part may occupy. Without any restrictions on $d_{ij}$, the dynamic programming algorithm runs in time quadratic in $k$. At first glance this may not seem problematic, but it is, in fact, prohibitive in practice. Even for reasonably sized images $k$ may be very large. For example, if $k$ is the number of pixels in an image pyramid, it could easily be on the order of one million.

One approach to this problem is to keep $k$ small by only considering a small set of locations that are detected with an interest-point operator, such as in the constellation model work by Fergus, Perona, and Zisserman [22]. We focus on the case where $k$ is large because experimental results confirm that a dense set of locations is important for keeping false negative rates low [23].

### 1.2 The Model

In this paper, we focus on a more restricted class of models. Instead of allowing $G_M$ to be any acyclic graph, we require it to be a star graph (a tree with only one level beneath the root). Star graphs are natural for many object classes including cars, bicycles, and even as a rough approximation to the structure of a pedestrian. State-of-the-art object recognition systems have been built using this exact restriction [9, 18, 20, 23]. Second, we require that $d_{ij}$ is a convex quadratic function of two variables. We will appeal to the convexity of $d_{ij}$ later when we discuss the efficiency of the algorithms that we develop.

To make this discussion more concrete, we will specify additional details for the form of our model $M$ and the set of possible model configurations $\mathcal{L}^n$. It is important to note that these details are not necessary for the matching algorithms developed in this paper, but are provided in order to make the discussion more concrete and to relate the algorithms to our object recognition system, which is described in Chapter 4. The only two restrictions required by our matching algorithms were those described in the preceding paragraph (namely, $G_M$ is a star graph and $d_{ij}$ is a convex quadratic function).
1.2.1 The Configuration Set $\mathcal{L}^n$

Recall that our goal is to detect and localize instances of a target object class (instances from now on) in an image $I$. An instance may appear at any scale and any translation within $I$. Each part, therefore, may match the image at any scale and any translation. Thus we need $\mathcal{L}$ to contain every shift, at every level, of a scale pyramid constructed from $I$. We will ignore object rotations and assume that $M$ is capable of handling some range of rotation by virtue of its deformable part structure, though this range will likely be a fairly limited. We recognize that this is a significant omission that deserves treatment in future research.

Working directly with pixel data is problematic due to variations in scene illumination and object color, and due to noise introduced by the imaging process. Rather than modeling these variables directly, we choose to mitigate their influence by using a function $\eta$ that maps an image to a feature image (see [35] for an overview of feature detectors and descriptors). Whereas an image is simply a two-dimensional array of vectors, typically in $[0, 1]^3$, called pixels, a feature image is a generalization of the image structure where each pixel is a vector in $\mathbb{R}^k$. To distinguish these generalized pixels from normal pixels, we will call them features. To be useful, the features computed by $\eta$ must introduce some measure of invariance to photometric effects and, ideally, to small-scale shape deformations.

The feature-image representation makes the mapping between image coordinates and feature space coordinates easy to compute, though not necessary one-to-one (many pixels may be mapped to the same feature, for instance). To illustrate the utility of this design, imagine that we have matched a template to some location in the feature image. By converting the template’s coordinates from feature space to image space, we can easily localize the template in the image. The Histogram of Oriented Gradients (HOG) feature map is a popular example of a function that has the properties that we just described [12].

Using an input image $I$ and our feature map $\eta$, we can construct a feature pyramid $H = (H_1, \ldots, H_s)$, containing $s$ feature images called levels. The levels are sorted in decreasing order of size and the scaling factor between levels is a fixed constant $c$. Level $i$ is computed as $H_i = \eta(\text{scale}(I, c^{i-1}))$. The function $\text{scale}(I, x)$ returns an image created by scaling $I$ by a factor of $x$. The scaling factor $c$ is selected such that for some positive integer $d$ the ratio of each dimension between $\text{scale}(I, c^{i-d})$ and $\text{scale}(I, c^i)$ is exactly 2 (levels double in size every $d$ steps in the pyramid). Note that features at the top of the feature pyramid capture coarse image data while features at the bottom capture finer resolution information. Figure 1.1, A-B,
shows an example of constructing a feature pyramid from a sample image.

We can now define the set of all possible locations for a part as

\[ \mathcal{L} = \{(x, y, l) \mid l \in [s] \text{ and } (x, y) \text{ is a valid coordinate in } H_1\}, \]  

(1.2)

where we define \([s] = \{1, \ldots, s\}\). It should be noted that both \(H\) and \(\mathcal{L}\) depend on the input image \(I\) (\(\mathcal{L}\) will be different for images with different dimensions, for example). As in the score function, we omit this parameterization to simplify notation.

The set of all model configurations is simply \(\mathcal{L}^n\). At times it will be necessary to refer to a partial configuration that specifies only the first \(j\) part locations. We will use the notation \(L|_j = (l_1, \ldots, l_j) \in \mathcal{L}^j\). Additionally, we will need to discuss the extension of a partial configuration in \(\mathcal{L}^j\) to a complete configuration in \(\mathcal{L}^n\). To denote the concept of configuration completion, we will write \(L|_{jn} = (l_1, \ldots, l_j, l'_{j+1}, \ldots, l'_n) \in \mathcal{L}^n\).

1.2.2 Multiscale Star Models

We define the parts of our model \(M\) to be two-dimensional arrays of weight vectors in \(\mathbb{R}^k\), so that each vector is the same length as the features in \(H\). We call a part with this structure a filter. Indeed, a filter is
also a feature image. We use the term filter to emphasize that the filter’s “pixels” act as *weights on features*, rather than features. Again, by abuse of notation, we will use $v_i$ to refer to the $i$-th part of $M$, the $i$-th vertex of $G_M$, and the filter representing part $i$.

Note that any array containing $m$ cells, each holding a vector in $\mathbb{R}^k$ (e.g., a regular image, a feature image, or a filter), can be converted into a single vector in $\mathbb{R}^{mk}$ by concatenating all array cell vectors in a fixed order. We will refer to the result of this concatenation process as the *vector representation* of an array of vectors, and denoted it by $vec(\cdot)$. Now we can define the part match function $m_i(l_i)$ as the dot product $vec(v_i) \cdot vec(A(l_i))$, where $A(l_i)$ is the feature image formed by cropping out a sub-array of $H$, such that $A(l_i)$’s upper-left corner is located at $l_i$, and its height and width match $v_i$’s dimensions (see Figure 1.1, B-D).

Our model $M$ will be represented by the star graph $G_M = (V, E)$. We distinguish vertex $v_1$ to be the root of our tree. At this point, we make a slight departure from the original notion of a pictorial structure model and define a *multiscale* model. Rather than modeling local appearance properties of the object class, the filter $v_1$ will be a coarse-resolution rigid template for the entire object. We will refer to this filter as the *root filter*. Each of the remaining parts, $v_2, \ldots, v_n$, is specified at twice the spatial resolution of $v_1$ (i.e., they must appear in $H$ exactly $d$ levels below $v_1$) and models local appearance properties of the object class. We will refer to the non-root filters as *part filters*. To reiterate $M$’s topology in terms of filters, there is a spring between the root filter and each of the part filters, but no other springs. Figures 1.2 & 1.3 illustrate multiscale star models for the classes bicycle and person.

This formulation of $M$ adds new constraints to the set of possible configurations. A valid configuration is any configuration $L = (l_1, \ldots, l_n)$ with $l_1 = (x_1, y_1, s_1)$, where both of the following are true:

1. The root filter is placed above level $d$: $s_1 > d$,
2. All part filters are placed exactly $d$ levels below the root filter: $\forall l_i, i > 1, l_i = (x_i, y_i, s_1 - d)$.

Since our model only permits a spring between the root filter and each part filter, we drop one index from $d_{ij}$ and let $d_i(l_1, l_i) = d_{1i}(l_1, l_i)$. We assume that each spring in $M$ has a convex quadratic deformation cost.
of the form

\[ d_i(l_1, l_i) = a_{xi} \tilde{x}_i^2 + b_{xi} \tilde{x}_i + a_{yi} \tilde{y}_i^2 + b_{yi} \tilde{y}_i, \]

where \( l_1 = (x_1, y_1, s_1) \), \( l_i = (x_i, y_i, s_1 - d) \), and we define \( \tilde{x}_i = x_i - (2x_1 + u_{xi}) \), and \( \tilde{y}_i = y_i - (2y_1 + u_{yi}) \). The vector \( u_i = (u_{xi}, u_{yi}) \) defines an offset relative to the root filter’s location from which spring displacement is measured. The coefficients on the quadratic terms must be positive to ensure that this function is convex and non-linear. Note that equation (1.3) meets the requirements that we specified earlier, though \( d_i \) need not be axis aligned, in general, as we have defined it here.

Figure 1.2: A visualization of \( M \) for the class bicycle. (A) The coarse-resolution root filter; (B) six part filters with twice the spatial resolution of the root filter; (C) a visualization of the deformation cost functions where darker implies lower cost. The filters are visualized by interpreting each filter cell vector as a set of weights that votes for (positive values) or against (negative values) particular features. Here HOG features are used, so the weights can be understood as selecting preferential gradient orientations. Only the positive filter values are shown here.

### 1.3 Matching Pictorial Structure Models by Dynamic Programming

To provide motivation for our new matching algorithm, a quick review of the dynamic programming approach to matching pictorial structure models is needed. We will assume that \( M \) and \( L \) are of the form defined in the previous section. Note that this dynamic programming algorithm is applicable to any acyclic graph even though we are only considering star graphs at this point. The derivation of this algorithm in a statistical setting can be found [20].
Let \( v_r = v_1 \) be the root vertex, let \( \pi_v \) denote \( v \)'s parent vertex, and let the set of \( v \)'s children vertices be denoted by \( C_v = \{ c \in V \mid \pi_c = v \} \). We can maximize equation (1.1) using the following dynamic programming algorithm.

1. For each leaf vertex \( p \in V \), compute the score of the best location for part \( p \) as a function of part \( \pi_p \)'s location:
   \[
   B_p(l_{\pi_p}) = \max_{l_p} \left( m_p(l_p) - d_{p\pi_p}(l_p, l_{\pi_p}) \right). 
   \]
   By replacing \( \max \) with \( \text{argmax} \), we can compute the optimal location for part \( p \) as a function of part \( \pi_p \)'s location.

2. For each internal node \( p \in V \setminus \{ v_r \} \), once \( B_c \) has been computed for all \( c \in C_p \), compute the score of the best location for part \( p \) as a function of part \( \pi_p \)'s location:
   \[
   B_p(l_{\pi_p}) = \max_{l_p} \left( m_p(l_p) - d_{p\pi_p}(l_p, l_{\pi_p}) + \sum_{c \in C_p} B_c(l_p) \right). 
   \]
   Again, replace \( \max \) with \( \text{argmax} \) to compute optimal locations for part \( p \).

3. Finally, compute the maximum of equation (1.1) by solving
   \[
   \max_{l_r} \left( m_r(l_r) + \sum_{c \in C_r} B_c(l_r) \right). 
   \]
A maximum score configuration can be found by substituting arg\text{max} for \text{max}.

Because we have assumed that \(d_{ij}\) is of the form specified in (1.3), the \text{max} problem in steps 1 and 2 can be computed in time linear in \(|\mathcal{L}|\) using an efficient algorithm for generalized distance transforms [19].

### 1.4 From Dynamic Programming to Heuristic Coarse-to-Fine Search

Even though we already have an efficient algorithm for maximizing equation (1.1), in practice the number of model parts must be fairly small (e.g., 7-14 is common) in order to make training and testing these models on large datasets tractable. Clearly this algorithm will be a major performance bottleneck if used on a rich grammar model that defines an object class with variable structure and a modest number of parts. At the very least, an efficient parsing algorithm is required to avoid brute-force maximization over the score of every derived structure (see parsing with AO* search in [21]).

Even with a parsing algorithm the central efficiency problem is that it still must evaluate \(m_i\) at each valid location for each part (the cost of computing \(d_i\) is negligible, so we ignore it). From an intuitive viewpoint, we expect that most images will be sparsely populated with instances of the target object class. In particular, the number of instances should be much smaller than the number of locations in \(\mathcal{L}\). This observation provides strong motivation for seeking an algorithm that only explores promising configurations. That is, we wish to only evaluate \(m_i(l_i)\) when there is a reasonable chance that \(l_i\) will appear in a solution. Exploiting this sparsity property has guided other approaches to fast object detection such as the Viola-Jones face detector [33] and the application of branch-and-bound to bag-of-visual-word classifiers in work by Lampert, Blaschko, and Hofmann [29].

The new matching algorithm presented in the next two chapters is formed by merging best-first heuristic search with coarse-to-fine processing. Variations on these approaches have appeared numerous times in a number of areas including computer vision and natural language processing. We will discuss the relationship of some of these approaches to ours in Section 2.6. To our knowledge this the first time best-first heuristic search and coarse-to-fine processing has been applied in conjunction to the problem of localizing generic object classes.
Chapter 2

Heuristic Coarse-to-Fine Matching

In Chapter 1 we introduced the matching problem for pictorial structure models, defined the form of the object we are matching our model to (a feature pyramid), and defined the form of our model (a multiscale star model with convex quadratic deformation costs). We also reviewed an efficient dynamic programming algorithm for solving the matching problem, but observed that it processes all potential part locations rather than focussing on what we expect to be a much smaller number of promising locations.

In this chapter, we propose a new algorithm for matching star models (maximizing equation (1.1)) that combines a coarse-to-fine detection process with heuristic search. We present the algorithm as an extension to traditional coarse-to-fine methodology. At the end of the chapter, we will make the algorithm’s connection to AO* search [31] explicit.

2.1 Coarse-to-Fine Detection

The dynamic programming algorithm required computing $m_i$ at each location in $L$ for each part in $M$. One way to reduce this computational load is through a technique called coarse-to-fine (CTF) processing. In CTF processing, a hierarchy of models is used to introduce complexity gradually, rather than all at once. Each model in the hierarchy is called an abstraction of the full model. These models are designed so that more abstract (i.e., simpler, coarser) models will be computationally less expensive to evaluate than more concrete (i.e., more complex, finer) models. A CTF process is an algorithm that uses the abstraction hierarchy, and a sequence of thresholds, to filter out candidate solutions.
Applied to object detection, one can imagine a CTF process that evaluates each candidate location with each model abstraction, in coarse to fine order, but discards candidate locations as soon as a model fails its threshold test. Computational costs are reduced if the coarser models are able to prune a large amount of the search space. Ideally, the full model (the finest in the hierarchy) would only be evaluated at a small number of promising locations.

In some applications, the abstraction hierarchy is designed by hand. We prefer to seek an automated method for generating abstractions. To accomplish this goal, first we fix an ordering of the parts \( V_o = (v_1, \ldots, v_n) \). The first part \( v_1 \) is always the root filter \( v_1 \), and \( v_2, \ldots, v_n \) are the part filters taken in some order. Later we will address the problem of selecting this ordering, but for now assume that it is arbitrary. To simplify notation, we assume that the ordering \( V_o \) is the same as the natural indexing of \( V \) (i.e., \( M \) was built with \( V_o \) in mind). Now we can drop the cumbersome parentheses and simply write \( V_o = (v_1, \ldots, v_n) \).

Having fixed the part order, let \( M_{v_i v_j} \) denote the model \( M \) restricted to parts \( v_i, \ldots, v_j \). We now define our CTF abstraction hierarchy as

\[
M_i = M_{v_1 v_i}, \quad \text{for } i \in [n].
\]

The most abstract model \( M_1 \) consists of only one part: the root filter. As we move down the hierarchy (increasing \( i \)), we add in parts in the order of \( V_o \) (see Figure 2.1). This interpretation is especially appealing in the case of a multiscale representation of the object class. In our setting, \( v_1 \) acts as an inexpensive low-resolution filter for the entire object (also recall that it only operates on the pyramid above level \( d \)). We expect that evaluating this filter at every valid location will be very fast. At each location that survives the root filter test, we apply the next least abstract model, which uses one new part to look for more evidence in the image near the root filter location. At locations where more evidence is found, the next least abstract model is applied, and so on. Root filter locations that seem sufficiently hopeless, as determined by a threshold, are pruned from the search.

In addition to being automatically defined, this coarse-to-fine hierarchy has the significant advantage that it does not require any additional information beyond what is already contained in \( M \). Therefore, given a procedure that learns \( M \), no additional training is required to learn the abstract models.
Figure 2.1: Moving down the coarse-to-fine abstraction hierarchy, from left to right. At each step we add one additional part to refine the model.

Another appealing property of our definition is that computing the score function for a more complex model requires only a slight modification to the score of a simpler model. To see this we parameterize the score function by a model abstraction, yielding the equation

$$S_{M_j}(L|_j) = \sum_{i=1}^{j} m_i(l_i) - \sum_{i=2}^{j} d_i(l_1, l_i).$$

(2.2)

Recall that $L|_j$ is a partial configuration $L|_j = (l_1, \ldots, l_j)$. Note the recursive relationship between $S_{M_j}$ and $S_{M_{j-1}}$:

$$S_{M_j}(L|_j) = S_{M_{j-1}}(L|_{j-1}) + m_j(l_j) - d_j(l_1, l_j).$$

(2.3)

Equation (2.3) implies that we can efficiently compute the score of a more concrete model by adding and subtracting the values computed from $m_i$ and $d_i$, respectively, to the score of next coarsest model.

The last ingredient required to specify a CTF detection algorithm is a sequence of thresholds that each model abstraction must pass. The notion of thresholding a detector response is natural, even when not using a CTF detection process. A threshold on the full model provides a tunable nob that adjusts the tradeoff between the model’s precision and recall performance on test images. Furthermore, in object detection applications we are often not just interested in maximizing equation (1.1), but rather we desire to find all configuration of $M$ that are above a threshold so that we can explicitly control precision and recall according
to the application at hand. This leads to the following problem. Find the set of configurations

$$L_T = \{ L = (l_1, \ldots, l_n) \mid S_M(L) \geq T \text{ and } (l_2, \ldots, l_n) = \arg\max_{l'_2, \ldots, l'_n} S_M(l_1, l'_2, \ldots, l'_n) \}, \quad (2.4)$$

for some real number threshold $T$. Intuitively, the set $L_T$ includes the best scoring model configuration for each root location provided that the score is above the threshold $T$. Using this threshold sensitive formulation of the matching problem (the thresholded matching problem), we state a CTF detection algorithm:

**Algorithm 1** Coarse-to-Fine Detection

```plaintext
1: procedure CTF_DETECT($M, H, T$)
2:   select a threshold sequence $t_1, \ldots, t_n$
3:   $\hat{L}_T \leftarrow []$ \Comment{the solution set}
4:   for each $l_1 \in H$ do \Comment{visit each valid root location}
5:     $S[l_1] \leftarrow m_1(l_1)$ \Comment{accumulate $M$’s score when root at $l_1$}
6:     $L \leftarrow (l_1)$ \Comment{the current configuration}
7:     if $S[l_1] < t_1$ then \Comment{$M_1$ failed threshold test}
8:       continue
9:   end if
10:  for $i = 2$ to $n$ do \Comment{compute best location for $v_i$}
11:     $l_i \leftarrow \arg\max_{l \in \mathcal{R}_i(l_1)} (m_i(l) - d_i(l_1, l))$
12:     $S[l_1] \leftarrow S[l_1] + m_i(l_i) - d_i(l_1, l_i)$ \Comment{update score}
13:     if $S[l_1] < t_i$ then \Comment{$M_i$ failed threshold test}
14:       continue outer loop
15:     end if
16:     append $l_i$ to $L$ \Comment{extend $L$ toward completion}
17:   end for
18:   append $L$ to $\hat{L}_T$ \Comment{$L$ is a complete solution}
19: end procedure
```

At termination $\text{CTFDetect}$ will return every element in the set $L_T$ provided that the thresholds $t_1, \ldots, t_n$ are selected to satisfy the following condition

$$(\forall L \in L_T)(\forall j \in [n])(S_{M_j}(L|_j) \geq t_j). \quad (2.5)$$

We call any sequence $t_1, \ldots, t_n$ that satisfies this condition a sequence of admissible thresholds. Condition
(2.5) is trivially satisfied if we select the thresholds using

\[ t_j = \min_{L \in \mathcal{L}_r} S_{M_j}(L), \text{ for } j \in [n]. \quad (2.6) \]

Note that equation (2.6) is the tightest sequence of admissible thresholds that we can achieve (if any threshold were tightened, it would reject a valid solution), but it does not allow us to completely specify the algorithm because it depends on knowing the solution set that \texttt{CTFDetect} returns! For now we will assume that an oracle hands us admissible thresholds. We will address the problem of selecting the thresholds in chapter 3.

In \texttt{CTFDetect} we also limit the set that the \texttt{argmax} on line 11 is computed over. This confinement is necessary to avoid computing \( m_i \) at every location. We will discuss how to select \( \mathcal{R}_i(l) \) in Section 2.4.

### 2.2 Heuristic Coarse-to-Fine Detection

Another approach that can reduce the computational expense of solving the thresholded matching problem is to use a heuristic function that orders exploration of the search space, as is done in A* and AO* search.

Let \( L|_{j^*} = (l_1, \ldots, l_j, l_{j+1}^*, \ldots, l_n^*) \in \mathcal{L}^n \) be a completion of the partial configuration \( L|_j \) such that \( S_M(L|_{j^*}) \geq S_M(L|_{jn}) \), where \( L|_{jn} \in \mathcal{L}^n \) is any other completion of \( L|_j \). We call \( L|_{j^*} \) an optimal completion of \( L|_j \). A heuristic function \( h_j : \mathcal{L}^j \rightarrow \mathbb{R} \) is an upper bound on the difference between the score of \( L|_{j^*} \) and the score of \( L|_j \):

\[
(\forall j \in [n])(\forall L|_j \in \mathcal{L}^j)(h_j(L|_j) \geq S_M(L|_{j^*}) - S_M(L|_j)),
\]

\[
h_j(L|_j) = 0, \text{ if } j = n. \quad (2.7)
\]

The heuristic \( h_j(L|_j) \) is an estimate of how much the score of the partial configuration \( L|_j \) can improve as it is extended to completion. Since \( h_j \) is an upper bound, the estimate is always optimistic and never underestimates how much the score can change. The requirement that \( h_j(L|_j) = 0 \) if \( j = n \) is natural since when \( j = n \) the difference to estimate is known to be exactly zero. We call any function that satisfies conditions (2.7) and (2.8) an admissible heuristic.
The motivation for using a heuristic function is twofold. First, it allows us to apply the best-first search strategy: we will use the value \( S_{Mj}(L|j) + h_j(L|j) \) to determine the next partial configuration to extend toward completion. If the estimates are exact, that is, if we have equality in equation (2.7), then the best-first strategy will propel us directly to a configuration maximizing equation (1.1) (see [31]). Best-first search also yields the advantage that solutions to the thresholded matching problem, equation (2.4), are discovered in decreasing score order. Later we will show that this property allows us to apply a filtering step called non-maximal suppression directly inside the detection algorithm. In contrast, non-maximal suppression can usually be applied only after detecting all solution configurations. Second, while we do not expect our heuristic to be exact in practice, if it is a reasonably tight upper bound, then we expect it to prevent us from exploring most bad configurations. This property follows precisely from the fact that the best-first strategy requires that we inspect the most promising configurations first. Algorithm 2 \texttt{HCTFDetect} presents a heuristic coarse-to-fine detection procedure.

### 2.3 Optimality of Heuristic Coarse-to-Fine Detection

\texttt{HCTFDetect} is optimal in the sense that if the \( h_j \) are admissible heuristic functions and if the CTF thresholds \( t_j \) are admissible, then \texttt{HCTFDetect} will always find a solution to the thresholded matching problem (if and only if one exists) before finding any other solution or terminating without a solution. In the following discussion we assume the following properties of the inputs to \texttt{HCTFDetect} are true:

1. A solution to the thresholded matching problem exists: \( \max_{L \in \mathcal{L}^n} S_M(L) \geq T \).
2. \( h_1, \ldots, h_n \) are admissible heuristic functions.
3. \( t_1, \ldots, t_n \) is a sequence of admissible thresholds.

**Lemma 1.** Let \( L^* = (l_1^*, \ldots, l_n^*) = \arg\max_{L \in \mathcal{L}^n} S_M(L) \) be an optimal configuration, and let \( S^* = S_M(L^*) \) be the score of that configuration. At all times before \texttt{HCTFDetect} adds the first complete configuration to the solution list \( \mathcal{L}_T \), there exists a pair \((L|j, s) \in Q\) such that \( s \geq S^* \).

**Proof.** Let \( L|j^* = (l_1^*, \ldots, l_j^*) \) denote a partial configuration that contains the first \( j \) elements of \( L^* \). \( Q \) is initialized to contain every single-element partial configuration that passes threshold \( t_1 \). From the definition of a sequence of admissible thresholds, we know that \texttt{HCTFDetect} will never prune a partial optimal
Algorithm 2 Heuristic Coarse-to-Fine Detection

1: procedure HCTFDETECT($M$, $H$, $T$)
2:   determine threshold sequence $t_1, \ldots, t_n$ satisfying equation (2.4)
3:   determine heuristic functions $h_1, \ldots, h_n$ satisfying equation (2.7) and (2.8)
4:   $t_{n+1} \leftarrow T$
5:   $h_{n+1} \leftarrow 0$
6:   $\hat{L}_T \leftarrow []$
7:   $Q \leftarrow []$
8:   ▶ Initialize
9:   for each $l_1 \in H$ do ▶ visit each valid root location
10:      $S[l_1] \leftarrow m_1(l_1) + h_1(l_1)$ ▶ accumulate $M$’s score when root at $l_1$
11:      if $S[l_1] - h_1(l_1) \geq t_1$ then ▶ root threshold test
12:         $Q \leftarrow ((l_1), S[l_1])$ ▶ insert partial configuration $Q$
13:         $A[l_1] \leftarrow 2$ ▶ next abstraction level for $l_1$
14:   end if
15: end for
16: ▶ Best-first search with CTF threshold tests
17:   while $(L, s) \leftarrow Q$ and $s \geq T$ do
18:      $l_1 \leftarrow L[1]$ ▶ root location
19:      $a \leftarrow A[l_1]$ ▶ abstraction level
20:      if $a > n$ then ▶ check for complete solution
21:         append $L$ to $\hat{L}_T$
22:         NMS($Q, L$) ▶ non-maximal suppression
23:         continue
24:   end if
25:      $l_{a+1} \leftarrow \arg\max_{l \in R_{a+1}(l_1)} (m_{a+1}(l) - d_{a+1}(l_1, l))$ ▶ compute best location for part $a + 1$
26:      $S[l_1] \leftarrow S[l_1] + m_{a+1}(l_{a+1}) - d_{a+1}(l_1, l_{a+1}) - h_a(L) + h_{a+1}(L, l_{a+1})$ ▶ update score
27:      if $S[l_1] - h_{a+1}(L, l_{a+1}) < t_i$ then ▶ threshold test
28:         continue ▶ prune partial configuration
29:   end if
31:      append $l_{a+1}$ to $L$ ▶ extend $L$ toward completion
32:      update $(L, S[l_1])$ in $Q$
33:   end while
34: return $\hat{L}_T$
35: end procedure
configuration. Furthermore, we are assuming that a solution above the threshold exists (given by $S^* \geq T$). Therefore immediately after initialization, $Q$ contains $L|_1^*$. At all times after initialization, and before adding the first solution to $L_T$, $Q$ must contain a partial optimal solution derived by extending $L|_1^*$ toward completion. That is: there exists $j \in [n]$ such that $(L|_j^*, s) \in Q$. We conclude by considering the score of $L|_j^*$:

$$s = S_{M_j}(L|_j^*) + h_j(L|_j^*) \geq S_{M_j}(L|_j^*) + (S_M(L^*) - S_{M_j}(L|_j^*)) = S^*.$$  \hspace{1cm} (2.9)

\begin{proof}

Theorem 2 (HCTFDetect is optimal). The first complete configuration $L$ that HCTFDetect adds to the solution list $L_T$ has $S_M(L) = S^*$.

Proof. For a contradiction, suppose that $S_M(L) < S^*$. We know from lemma 1 that just prior to removing $L$ from $Q$ and adding it to $L_T$ that $Q$ contained an element $(L', s)$ with score $s \geq S^*$. In accordance with the best-first strategy, HCTFDetect removes elements from $Q$ in order of decreasing score, so $L$ must have the highest score in $Q$. This implies that $S_M(L) \geq s \geq S^*$, which contradicts the assumption. Since $S^*$ is the unique maximum value of $S_M$, it follows that $S_M(L) = S^*$.

\end{proof}

2.4 Efficient Computation

The argmax that appears in HCTFDetect on line 25:

$$l_j^* = \arg\max_l (m_j(l) - d_j(l_1, l)),$$

requires careful attention in order to make HCTFDetect efficient. We will outline a strategy for computing equation (2.10). First, we note that since $d_j$ is a convex quadratic function, we may impose an artificial limit on the range of $l$. We justify this reduction of the search space by noting that for all locations outside of a relatively small window centered on $v_j$'s resting position the deformation costs will always be too high. Suppose that the root is located at $l_1 = (x_1, y_1, s_1)$. We will limit the range of the argmax optimization to the set of locations on pyramid level $s_1 - d$: $\mathcal{R}_j(l_1) = ((2x_1, 2y_1) + u_j + \mathcal{W}_r) \times \{s_1 - d\}$. The vector $u_j$
is part $j$’s offset, relative to $l_1$, defined in the deformation cost function (equation (1.3)). The set $\mathcal{W}_r$ is the set of all two-dimensional grid points defining a square window of side length $2r + 1$:

$$\mathcal{W}_r = \{v_r = (v_{rx}, v_{ry}) \in \mathbb{Z}^2 : |v_{rx}| \leq r, |v_{ry}| \leq r, \text{ and } r \in \mathbb{Z}^+\}. \tag{2.11}$$

The side length of the search window is typically fairly small, about the size as a large part (e.g., 11 feature image cells). The new optimization problem is

$$l_j^* = \arg\max_{l \in \mathcal{R}_j(l_1)} (m_j(l) - d_j(l_1, l)). \tag{2.12}$$

Second, we note that the value of $m_j(l)$ may be needed by multiple instances of this problem. Imagine two placements of the root filter, $l_1$ and $l'_1$, that are adjacent in the same level of the feature pyramid. Part $j$’s search region for each of these root placements, $\mathcal{R}_j(l_1)$ and $\mathcal{R}_j(l'_1)$, will largely overlap, and as a result most of the values of $m_j$ can be shared if we cache $m_j(l)$ for each part $j$ and each location $l$. Conceptually one can think of the function $m_j$ as a pyramid of lazily computed 2D lookup tables.

Finally, because we must visit each cell of $m_j$ within the optimization region $\mathcal{R}_j(l_1)$ to retrieve its value (and possibly compute it), as we iterate over the cells we can maintain a pointer to the cell the maximizes equation (2.12). This scheme for sharing computation between instance of the maximization problem is essential for making HCTFDetect fast.

### 2.5 Non-maximal Suppression

The thresholded matching problem allows us to detect multiple instances of an object class in a single image. Detecting multiple instances, unfortunately, introduces a new difficulty: a pictorial structure model will often find many high scoring configurations near each instance. A solution to this problem, called non-maximal suppression (NMS), is to only report local maxima of $S_M$ on $\mathcal{L}^n$ as true detections. Using the dynamic programming algorithm reviewed in section 1.3, NMS can only be implemented as a post-processing step. In contrast, HCTFDetect will always find a local maxima before finding any local sub-optima because it discovers solutions in decreasing score order. This property allows us to incorporate NMS.
directly into the search algorithm. Furthermore, HCTFDetect can benefit from performing NMS on the fly because it may allow the procedure to further reduce the size of the search space. The performance increase becomes more evident as the depth of the feature pyramid increases.

Non-maximal suppression is applied each time a new solution $L$ is found (HCTFDetect line 21). The NMS procedure executes the following steps:

1. Project the root filter’s coordinates, placed according to $L$, into image coordinates. Call the rectangle formed by this projection $R$.
2. For each valid root location $l' \in L$, if $R$ has large overlap with the projection into image coordinates of the root filter placed according to $l'$, then remove all partial configurations that start with $l'$ from the priority queue.

### 2.6 Related Work

Coarse-to-fine methods have appeared numerous times in computer vision research. Our detection algorithm is most strongly connected to the body of coarse-to-fine research developed by Amit, Fleuret, and Geman over the last decade [26], [3], [2], and [25]. Aside from our use of best-first heuristic search we differ from their approach in that we assume discriminatively trained models, rather than generative models, and we define our thresholds and heuristics in terms of classifier scores rather than an estimated background probability distribution. The techniques developed in this paper also bear a strong relationship to the work of Viola and Jones [33], in which they propose using a cascade of weak, discriminatively-trained classifiers, each with a pruning threshold tuned to minimize false negative rates and maximize false positive rates. Our approach differs in at least three important respects.

First, we learn classifiers for generic object classes rather than just frontal faces, as is done in their work. For frontal faces, simple Haar-like features appear to be sufficient. Furthermore, a significant source of their detector’s speed relies on using these features, which can be computed in constant time using integral images – a representation introduced in their work. Dalal shows in [11] that an even more general set of Haar-like features is inferior to HOG features for detecting people with SVM-trained rigid templates.

Second, our framework incorporates a deformable spatial model. The Viola-Jones system, in contrast,
uses a fixed spatial model that is defined implicitly by the features chosen by each weak classifier.

Finally, we move away from the traditional sliding window approach – the common object detection practice of exhaustively sliding a detector over all translations and scales in a feature pyramid – to a best-first search method. Best-first search is made possible by introducing heuristics functions.

Using a heuristic to order evaluation of candidate solutions is similar in spirit to the object localization approach employed by Lampert, Blaschko, and Hofmann in [29]. They develop a general branch-and-bound algorithm and show how it can be applied to the popular bag-of-visual-words image representation using SVM classifiers with a variety of kernels. Their approach is not directly applicable to our framework because it relies on accumulating information over relatively large rectangular regions, using integral images, and then weighting each accumulated sum by a single constant from their classifier. In our setting, we require the learned classifier weights to vary over fairly small square regions (e.g., 8x8 pixels), and therefore we cannot aggregate information in the same way. Earlier work by Huttenlocher, Klanderman, and Rucklidge used branch-and-bound to match ridge templates defined by sets of points to binary images [13].

The efficiently computed upper bounds used by Lampert’s the branch-and-bound algorithm allow their system to be exact. One potential downside to our system, in comparison, is that we ultimately require using inadmissible heuristics. Our experimental results show that the loss detection accuracy due to inadmissibility is very small (typically 1-2%), and therefore we believe that our approximate method offers a favorable tradeoff between detector speed and accuracy.

Best-first heuristic search and coarse-to-fine processing are often viewed as orthogonal approaches. The widely practiced rule of thumb is that best-first search should be applied to problems that require returning a single best solution, whereas coarse-to-fine processing is appropriate for problems where all good solutions must be found. Coughlan and Yuille added pruning to best-first search in the application of tracking a single road in an aerial photograph [8]. In their application, pruning prevented exponential search space growth. The converse – adding best-first search to a coarse-to-fine detection algorithm – is rarely, if ever, done. A contribution of this paper is to note that when a problem requires returning all solutions that are good and locally maximal, then best-first search enables additional pruning of the search space.
Figure 2.2: The implicit AND/OR search graph for a model with 3 parts. The graph is drawn using the hypergraph notation from Nilsson [30]. Each node is an OR of zero or more connectors. A connector is either a single edge, or a group of edges joined by an arc that represents an AND of the connector’s children. At each node, a score is computed as the max over the scores of the outgoing connectors plus the node’s intrinsic score (either $m_i$, $d_i$, or 0). The score of a connector is the sum of its children’s scores. To clarify the illustration most nodes and connectors are omitted. The ones left visible highlight the shared sub-problems (computing $m_i$ and $d_i$) that make efficient computation with our model possible.

2.7 Discussion and Connection to AO* Search in AND/OR Graphs

We have presented HCTFDetect as an extension to a typical coarse-to-fine detection process that allows use of a heuristic function to order the application of the abstraction hierarchy. An equivalent formulation is to consider HCTFDetect as an extension to AO* search. In Figure 2.2 we show a fragment of the implicit AND/OR search graph. Our algorithm extends AO* by associating model abstractions and pruning thresholds to a subset of graph nodes (e.g., the bold nodes in Figure 2.2). When a node’s score falls below the pruning threshold, it is immediately set to $-\infty$. In our setting, the structure of the model simplifies the AND/OR search algorithm. The traditional AO* search algorithm given by Nilsson [30] has two steps: (1) top-down node expansion followed by (2) a bottom-up phase that bubbles updated scores back up the search graph. Instead of a general-purpose bottom-up phase, HCTFDetect only needs a single priority queue to propagate information from level 3 to level 0.

To see why, first note that we compute the value of each level 2 node as a single optimization problem
(line 25 of HCTFDetect), so we never have update a level 2 node after its value has been computed, even though it has more than one outgoing connector. Next, note that level 1 nodes only have a single outgoing connector (AND over part filters), so immediately after a level 2 node is computed, its parent in level 1 can be directly updated (line 26 of HCTFDetect) without taking a max. This leaves only one node left to updated – the single node in level 0. We use a priority queue to maintain the max over the freshly updated level 1 nodes, and hence the top of the priority queue is the level 0 node. In future work, we will extend this algorithm to handle more general acyclic AND/OR graphs by incorporating a true bottom-up phase and expanding our notion of model abstractions and heuristics. This will allow us to apply heuristic coarse-to-fine detection to grammar models (see [21]).

Orthogonal to the AO* connection, an interesting observation is that CTFDetect and HCTFDetect both perform exactly the same quantity of computational work (ignoring overhead from the priority queue). The real difference between these algorithms is how they order that work. This ordering has significant consequences that can make one algorithm more appropriate than the other depending on the task to which it is applied. Based on experimental results, we found that CTFDetect is faster than HCTFDetect when NMS is not used (typically by a factor of 1.3). This result is entirely the effect of CPU cache performance. As one might expect, because CTFDetect visits neighboring locations sequentially, it has much better cache coherence properties. By randomizing the visitation order, we can destroy the cache coherence – an experiment we tried – and in doing so we verified that CTFDetect performs almost exactly the same as HCTFDetect. This experimental test demonstrates that the overhead from the priority queue is negligible compared to the other computational costs. In contrast, if NMS is used during detection, then CTFDetect plus NMS as a post-processing step is slightly slower than HCTFDetect with embedded NMS. This difference is accentuated as more levels are added to the feature pyramid.

If the goal of a particular application is simply to find the best configuration, then HCTFDetect has a significant advantage over CTFDetect. HCTFDetect can terminate as soon as it finds the first (and best) solution, which typically leads to much better performance than CTFDetect on the same task.
Chapter 3

Determining Thresholds, Heuristics, and Part Order

In the preceding discussion of heuristic coarse-to-fine detection, we assumed that the heuristic functions $h_j$ and the sequence of thresholds $t_j$ were available to us through some oracle. The real situation is much murkier – we have to determine $h_j$ and $t_j$ bearing in mind that the performance of HCTFDetect is highly dependent on making good choices for these quantities.

State-of-the-art performance in object class detection on real-world images (such as in the PASCAL dataset) typically falls near 50% precision at 50% recall. Given the current state of performance, choosing admissible thresholds that yield a wide range of recall is not likely to produce a matching algorithm that is much faster than the dynamic programming approach. For this reason, we believe that it is necessary to explore inadmissible heuristics and thresholds. In this chapter we develop a framework for assessing the consequences of this choice.

An important hypothesis is that we expect the relative speed improvement of heuristic coarse-to-fine detection to increase as our models become more complex. For example, using mixture models with more components allows the filters in each component to become more specialized and therefore more discriminating. Discriminating filters have greater pruning power, which should reduce the proportion of filters that actually have to be evaluated. Another view is to consider a model with infinitely many part filters. If each part is able to prune a constant fraction of the candidate configurations, then model evaluation will neces-
sary terminate after evaluating a finite number of parts because the set of root filter locations is finite. In this
sense, coarse-to-fine detection allows the matching process to be somewhat independent of the number of
parts.

3.1 A Simple Statistical Framework

The structure of some problems allows for the formulation of nice heuristics, such as using minimum span-
ning trees to derive heuristic estimates in the traveling salesman problem [31]. In our setting, it is clear what
we want the heuristic to upper bound, but it is not clear if there is a simple sub-problem (such as finding a
minimum spanning tree) that will give us a reasonable estimate.

In this paper, we will assume that computing a good upper bound for use as the function $h_j(L|j)$ using
information specific to the partial configuration $L|j$ is computationally too expensive (i.e., all known meth-
ods of computing an upper bound are costly enough to eliminate any gains made by using heuristic search).
This assumption allows us to simplify the form of the heuristic function. Rather than considering heuristic
functions on the domain of partial configurations, we will restrict each $h_j$ to be a constant function. Thus
we have

$$\forall j \in [n], \forall L|j \in L^j[h_j(L|j) = \hat{h}_j].$$

That is, at each level of abstraction $j$ we use the same constant $\hat{h}_j$ as the estimate of how much the score can
change for any partial configuration in $L^j$. This simplifies matters, but we still need some way of selecting
the constants $\{\hat{h}_j\}$.

At one extreme, if the features in each pyramid cell of $H$ have a bounded norm, then for a fixed model
$M$ we know that at each level of abstraction the amount by which the score can improve is bounded by
a constant $h_j^{\max}$. If we choose $\hat{h}_j \geq h_j^{\max}$, then our heuristic will be admissible, but HCTFDetect
will be dominated by breadth-first search behavior and we will likely gain very little from using it. On the
other hand, if we select $\hat{h}_j < h_j^{\max}$, then we are not guaranteed an admissible heuristic. Suppose that
we are willing to accept an inadmissible heuristic. Clearly this choice will lead to an algorithm that is
not guaranteed to produce optimal solutions, but if we are able to reasonably bound the chance of finding
sub-optimal results, then the tradeoff should be worthwhile.

In order to analyze this question we need a framework that allows us to select \( \hat{h}_j \). Let \((I, L)\) be an (image, configuration) pair. We can define a simple statistical framework in which we assume that there exists an unknown distribution \( D \) over an arbitrary set of \((I, L)\) pairs. For each pair \((I, L)\) drawn from \( D \), we can compute the score \( S_{M_j}(L|j) \) of each partial configuration. These partial configuration scores are functions of random configurations from \( D \), and are themselves random variables. We define a new random variable \( h^*_j \) in terms of them as follows

\[
h^*_j(L) = S_M(L) - S_{M_j}(L|j).
\] (3.2)

The random variable \( h^*_j \) is exactly the quantity for which we seek an upper bound. We denote the induced distribution that generates \( h^*_j \) by \( D^*_j \).

Note that the following technique makes no assumptions about the set of \((I, L)\) pairs and their distribution \( D \). In practice, will are only interested in picking heuristics for the specific set of \((I, L)\) pairs that we care about (i.e., those corresponding to object instances). In section 3.4 we will describe the set of \((I, L)\) pairs that we consider in practice.

### 3.2 Probably Approximately Admissible Heuristics

Suppose that we have a model \( M \) and a sample drawn I.I.D. from \( D \). Our goal is to exploit the information contained in this sample in order to choose a good constant heuristic function \( \hat{h}_j \). We define good in a way that is analogous to a well learned concept in PAC learning (see [27]): a heuristic is good when the probability that its error is greater than \( \epsilon \) is less than the confidence \( \delta \).

We define an inadmissible heuristic to be in error exactly when it is not admissible with respect to a complete configuration \( L \). This can happen if the heuristic fails to be admissible with respect to any partial configuration formed by truncating \( L \):

\[
err(\hat{h}_1, \ldots, \hat{h}_n) = P_{(I,L) \sim D}(\hat{h}_1 < h^*_1(L) \lor \ldots \lor \hat{h}_n < h^*_n(L)).
\] (3.3)
Taking the union bound of this error we get

\[ \text{err}(\hat{h}_1, \ldots, \hat{h}_n) \leq \sum_{j=1}^{n} P_{(I,L) \sim D} (\hat{h}_j < h_j^*(L)). \] (3.4)

Our goal is to find an algorithm for selecting \( \hat{h}_1, \ldots, \hat{h}_n \) such that

\[ P(\text{err}(\hat{h}_1, \ldots, \hat{h}_n) > \epsilon) < \delta, \] (3.5)

for user defined values of \( \epsilon \) and \( \delta \).

From another perspective, with probability at least \( 1 - \delta \) such a heuristic causes best-first search to find an optimal configuration (from \( D \)) before finding any solution with a lower score, with probability greater than \( 1 - \epsilon \). Because we have high confidence that such a heuristic is “approximately” admissible – in the sense that it is usually admissible, but may make an arbitrarily large mistake with low probability – we call it a \textit{probably approximately admissible} heuristic.

Let \( \mathcal{H}_j \) be a set of \( h_j^* \) values sampled I.I.D. from \( D_j^* \). We propose the following simple rule for selecting our heuristics:

\[ \hat{h}_j = \max (\mathcal{H}_j). \] (3.6)

Given this rule, what can we say about the relationship between \( \epsilon, \delta, \) and \( \mathcal{H}_j \)?

**Theorem 3** (Lower bound on \(|\mathcal{H}_j|\)). Given a collection of samples \( \{\mathcal{H}_j \sim D_j^*\}, j \in [n] \), each of size \( m \), if \( \hat{h}_j = \max (\mathcal{H}_j) \) and \( m > \frac{n}{\epsilon} \ln \frac{n}{\delta} \) for some \( 0 \leq \epsilon, \delta \ll 1 \), then \( P(\text{err}(\hat{h}_1, \ldots, \hat{h}_n) > \epsilon) < \delta. \)

**Proof.** For each \( j \in [n] \) there is a \( \frac{\epsilon}{n} \)-probability interval \([\beta_{j\epsilon}, \infty)\) given by

\[ P_{x \sim D_j^*} (x \geq \beta_{j\epsilon}) = \frac{\epsilon}{n}. \] (3.7)

Denote these intervals by \( T_j = [\beta_{j\epsilon}, \infty) \) and their complementary intervals by \( \bar{T}_j = (-\infty, \beta_{j\epsilon}) \). The probability that a single sample point drawn from \( D_j^* \) is in \( \bar{T}_j \) is exactly \( (1 - \frac{\epsilon}{n}) \). Furthermore, the probability that \( n \) I.I.D. samples from \( D_j^* \) are all in \( \bar{T}_j \) is \( (1 - \frac{\epsilon}{n})^n \).
If there exists a \( j \in [n] \) such that all of the \( m \) samples in \( \mathcal{H}_j \) are in \( \bar{T}_j \), then by the rule in equation (3.6) \( \hat{h}_j < \beta_j \epsilon \), which implies that \( P_{x \sim D_j^*}(x \geq \hat{h}_j) \geq \frac{\epsilon}{n} \). In order for the union bound in equation (3.4) to be greater than or equal to \( \epsilon \) such a \( j \) must exist.

We bound the probability that all \( m \) samples fall in \( \bar{T}_j \), for any \( j \), by \( \delta \) to bound the probability that the union bound will exceed \( \epsilon \).

\[
P(\text{err}(\hat{h}_1, \ldots, \hat{h}_n)) > \epsilon) \leq n \left(1 - \frac{\epsilon}{n}\right)^m < \delta.\tag{3.8}
\]

Using the approximation \((1 - x) \leq e^{-x}\) for small \( x \), we have:

\[
m > \frac{n}{\epsilon} \ln \frac{n}{\delta}.\tag{3.9}
\]

The \text{max} rule (equation (3.6)) for picking the heuristic functions is efficient in the sense that the sample size needed to estimate \( \hat{h}_j \) grows as a linear function of \( \frac{n}{\epsilon} \) and a logarithmic function of \( \frac{n}{\delta} \). This theoretical result justifies why our very simple approach is reasonable. For example, with \( n = 6 \) (a typical value for our class of models) we can achieve with at least 99% confidence an inadmissibility rate of at most 1% using about 3800 examples. Receiving 4000 positive examples for an object class used to be a completely unreasonable requirement. However, modern datasets for object detection, such as PASCAL 2007 & 2008 [14, 15], are starting to provide datasets this large for some classes, though it is still uncommon. Given this situation, it is important to note that in practice far fewer positive examples, around 300 for example, suffice for producing heuristic functions that yields detection accuracy results that are comparable with the exact dynamic programming algorithm. This may be attributed to slackness in the somewhat conservative bounds given in the theoretical results and to the fact that given a positive example, finding a configuration that is within some small distance from the optimal solution is usually good enough for object detection applications. This point is especially relevant when using deformable part models where the difference between an optimal configuration and a next best configuration could amount to nothing more than a slight shift of a single part.
3.3 Probably Approximately Admissible Thresholds

The problem of determining the sequence of coarse-to-fine pruning thresholds is very similar in spirit to the problem of determining the heuristic function. We will begin from the same premise: (1) that there is no known method for determining good thresholds from some general property of the problem; (2) that selecting thresholds that are provably admissible, based on a fixed model and features with bounded norm, is too permissive; and so, (3) we are willing to relax the admissibility requirement if we can find a threshold that is good in the sense of being probably approximately admissible.

We will use the same statistical framework that was introduced in the previous section to develop a method for selecting thresholds. To simplify the following discussion we will assume that $S_M(L) \geq T$ for all $(I, L) \sim D$. As previously discussed, the distribution $D$ induces a distribution on the scores of partial configurations obtained from that distribution. Given a random pair $(I, L) \sim D$, we seek a sequence of thresholds $t_j$ that are estimates of the lower bounds of the random variables $S_{M_j}(L|_j) \sim S_j^*$. We proceed in the same vein as when we determined the heuristic functions.

A threshold $t_j$ is in error with respect to a partial configuration $L|_j$ exactly when $S_{M_j}(L|_j) \geq T$, but $S_{M_j}(L|_j) < t_j$. By natural extension, a sequence of thresholds $t_1, \ldots, t_n$ is in error with respect to a configuration $L$ exactly when at least one of the thresholds $t_j$ is in error. The error of a sequence of thresholds can be bounded by

$$\text{err}(t_1, \ldots, t_n) \leq \sum_{j=1}^{n} P_{(I,L)\sim D}(t_j > S_{M_j}(L|_j)). \tag{3.10}$$

In order to obtain another probably approximately admissible result for the threshold error rate, suppose that we have a sample $S_j \sim S_j^*$ for each $j \in [n]$. We propose the following simple rule for selecting the sequence of thresholds from the training sample:

$$t_j = \min (S_j). \tag{3.11}$$

Using this rule we can obtain the same probably approximately admissible result for thresholds as for the heuristic functions.
Theorem 4 (Lower bound on $|S_j|$). Given a collection of samples $\{S_j \sim S^*_j\}$, $j \in [n]$, each of size $m$, if

$t_j = \min (S_j)$ and and $m > \frac{n}{\epsilon} \ln \frac{n}{\delta}$ for some $0 \leq \epsilon, \delta \ll 1$, then $P(\text{err}(t_1, \ldots, t_n) > \epsilon) < \delta$.

Proof. The proof follows exactly the same line of reasoning as in Theorem 3.

This method for selecting coarse-to-fine thresholds can be seen as a principled derivation of the standard trick: select thresholds that yield low false negative rates on training data. Our result simply states a generalization bound for this approach.

3.4 Determining Heuristics and Thresholds

In practice we are only interested in selecting heuristics and thresholds that are probably approximately admissible with respect to locally optimal detections of object instances. Pruning partial configurations outside of this set is always advantageous. Furthermore, the application of an inadmissible best-first heuristic to a sub-optimal configuration does not break the optimality of the search as long as the heuristic is admissible with respect to an optimal configuration also in the queue. We define the set of $(I, L)$ pairs of interest as those for which the following conditions are true:

1. $I$ is an image containing one instance specified by a ground-truth bounding box,
2. $L$ is a locally optimal (in the sense of NMS) configuration of $M$ in the feature pyramid constructed from $I$,
3. $M$‘s root filter, placed according to $L$, projected from feature pyramid coordinates to image coordinates has large overlap (e.g., at least 70%) with the ground-truth bounding box for the instance in $I$ (overlap can be measured by intersection area over union area),
4. and finally, the score must be above threshold: $S_M(L) \geq T$.

In the following chapters we assume that the distribution $D$ is over this set.

Given this set of interest, our approach for selecting thresholds is equivalent to the multiple-instance pruning method developed by Zhang and Viola in [34]. There are, however, some important differences in our derivation and use of this threshold rule. Zhang and Viola consider the scale and location of detection windows as the only latent information, whereas we treat all filter locations and scales (and eventually
mixture component labels) as latent variables. Since thresholds are determined by taking the minimum over a training set of the maximum score over latent variable settings for each element in that set, introducing more latent information should allow for computing more aggressive thresholds.

We arrive at our threshold selection rule from a very different perspective than that of Zhang and Viola. We derive the threshold rule by first finding a good method for selecting a best-first heuristic function that is approximately admissible with high probability, and then recognizing that selecting thresholds can be formulated as the same type of problem.

Finally, we show in our experimental results that these inadmissible thresholds and heuristics can be used for training and testing detectors for a diverse set of object classes. Zhang and Viola only demonstrate their approach on frontal face detection.

3.5 Choosing a Part Order

Up until now we have assumed that $M$’s parts were given in some fixed order $V_o$. It is clear that this order may be crucial to the performance of HCTFDetect. For instance, if the first part in $V_o$ is uninformative, that is if it scores almost exactly the same everywhere in the feature pyramid, then it will not be very useful. More specifically, an uninformative part will be very poor at targeting the best-first search in on a solution, and at pruning partial configurations that are below threshold. Hence the computational time spent evaluating an uninformative part is largely wasted. This example is extreme. No part will be truly uninformative, otherwise we would simply remove it from the model. We do expect, however, that there will be a spectrum of how informative each part is. Our goal in picking a part ordering is to place the more informative parts before the less informative parts.

We have experimented with a number of approaches that all work almost equally well in practice, and perform much better than selecting a random part order. This problem is addressed by Bourdev and Brandt in [4] where they reorder a cascade of weak classifiers in order to improve detection efficiency. They suggest greedily selecting the classifier that has the largest gap between its mean score on negative examples and its mean score on positive examples. In addition to their method, we have also looked at greedily selecting the part that has the highest (estimated) likelihood of scoring higher on a positive example than on a negative example.
One method, which is especially simple, greedily selects the next part by minimizing the expected difference between the heuristic and the random variable that the heuristic is upper bounding. Suppose that we have already ordered the parts $v_1, \ldots, v_i$. Greedily select the next part by solving

$$
\min_{j=i+1, \ldots, n} \left( \max_{H_j} (H_j) - \frac{1}{|H_j|} \sum_{x \in H_j} x \right), \quad (3.12)
$$

where $H_j$ is a sample from $D_j^*$. There are three keys to understanding why this method works. First, note that equation (3.12) only uses differences of scores from optimal configurations on positive image data (recall how $D_j^*$ was defined). Second, we are assuming that we have trained a reasonably good classifier that has discriminating parts. These parts should therefore reliably score higher on positive examples than on background data. Finally, when estimated from real data $D_j^*$ appears strongly unimodal and resembles a low-variance Gaussian distribution. Given these factors, the part that minimizes equation (3.12) is more likely to make a good partial configuration stand out against the background noise than the other parts are. In contrast, consider a part that has a large expected difference. This part is less informative because it leads to a score that more easily confuses positive data with background data. Intuitively, this method attempts to select a part order that prioritizes heuristics that are tight for good partial configurations.
Chapter 4

Discriminatively Trained Mixtures of Multiscale Star Models

In the first three chapters we described a new heuristic coarse-to-fine algorithm for approximately solving the thresholded matching problem. In this chapter, we will set that algorithm aside momentarily and describe a complete object recognition system that is based on, and extends, the model described in section 1.2. This work is an extension of the system based on multiscale star models described in [18], and it recently tied for first place in the 2008 PASCAL VOC Challenge [16]. The model presented in this chapter can be seen as a small step toward grammar models. We introduce a mixture model formulation that is equivalent to a 2-level grammar where a derivation of a model configuration is the selection of one multiscale star model from a pool of mixture components, followed by the derivation of that model.

We will cover the extensions to our model and describe the LSVM based training algorithm. In the next chapter, we will return to heuristic coarse-to-fine detection and show how our system can be modified to use HCTFDetect for model training and detection at test time. Finally, we will present experimental results that compare the performance of models learned with HCTFDetect to models learned with the slower, but exact, dynamic programming algorithm.
4.1 Mixture Models

The power of pictorial structure models lies in their ability to capture intra-class variation by allowing for a range of shape deformation. Nevertheless this degree of flexibility is clearly not enough. Intra-class variation can be so significant that two instances of the same class will not share the same parts at all. The differences can arise from a class having a large range of structural forms or it can be due, quite simply, to viewpoint changes that render two instances completely unrecognizable by a single pictorial structure model (see Figure 4.1).

![Figure 4.1: Three bicycles viewed from two different viewpoints. The two poses share no common parts. The bicycles also exhibit significant structural differences such as the ratio of the wheel size to the frame size, the handle bar shape, and the bicycle frame geometry.](image)

We propose using a mixture of the multiscale star models (a mixture model from here on) that were introduced in Chapter 1 as a means of capturing extreme intra-class variation (see Figure 4.2). A mixture model is defined as a collection of $C \geq 1$ multiscale star models, each of which is referred to as a component of its mixture model:

$$\mathcal{M} = (M^{(1)}, \ldots, M^{(C)}).$$

(4.1)

Note that mixture models potentially extend the range of object rotations that we can handle – offering a partial solution the problem raised earlier.

A detection with a mixture model specifies the configuration of exactly one component. We can naturally accommodate this within our framework by treating the component label as a new dimension in an extended configuration space. Let $[C] \times \mathcal{L}^n$ be the set of all configurations for a mixture model $\mathcal{M}$. We can define

40
the score function for a mixture configuration $L' = (k, l_1, \ldots, l_n) \in [C] \times \mathcal{L}^n$ as the original score function parameterized by the selected mixture component $M^{(k)}$:

$$S'_{M}(L') = S_{M^{(k)}}(L) = \sum_{i=1}^{n} m_i(l_i) - \sum_{i=2}^{n} d_i(l_1, l_i),$$

(4.2)

where $L = (l_1, \ldots, l_n) \in \mathcal{L}^n$. The score of a mixture model abstraction is simply the original score function parameterized by the abstraction of the selected mixture component.

The matching problem for our mixture models can be restated as

$$L'^* = \arg\max_{k \in [C], L \in \mathcal{L}^n} S_{M^{(k)}}(L).$$

(4.3)

Figure 4.2: A two component mixture model for the class bicycle. The top row is the model pictured in Figure 1.2 and the bottom row is a new model that captures near frontal and rear views. This mixture model is capable of detecting all bicycle instances in Figure 4.1.

### 4.2 Feature Map

We use a modified version of the Histogram of Oriented Gradients (HOG) feature maps. In contrast to the original 36-dimensional features presented in [12], we found a 13-dimensional representation, capturing 9 orientations under a single normalization plus 4 features capturing texture gradients, that yields the exact
same detection accuracy as the original 36-dimensional features. The reduced feature set can be seen as a linear projection of the original features that preserves most of their information. An expansion to 18 contrast sensitive orientations and 4 texture gradients leads to improved performance on some classes. In our system, we use 31-dimensional HOG features that include 18 contrast sensitive and 9 contrast insensitive orientations, and 4 texture gradients. See [17] for additional technical details.

4.3 Training Mixture Models

We still have to address the important question of how to learn the mixture models described in this paper. We will only cover this topic at a high level and refer to the reader to [17] for full details. In the following sections we will discuss the setting of our training problem and the algorithm used to learn the mixture models.

4.3.1 Model Parameterization and Feature Vectors

We can view a mixture model as a single vector of model parameters $\beta$. The vector $\beta$ is formed by concatenating the vector representation of each mixture component. A mixture component is represented as the vector formed by concatenating the vectorized filters, $\{\text{vec}(v_i)\}, i \in [n]$, followed by the set of deformation function parameters, $\{a_{xi}, a_{yi}, b_{xi}, b_{yi}\}, i \in [n]$.

We call a vector $q$, with the same dimension as $\beta$, that encodes training data a feature vector. We form a vector $q$ from a feature pyramid $H$ and a configuration $L = (k, l_1, \ldots, l_n)$ using the following procedure.

1. Initialize $q$, of length equal to the dimension of $\beta$, to 0.
2. Let $i_k$ be the index in $\beta$ where the vector representation of component $k$ starts.
3. Crop out the sub-arrays $\{A(l_i)\}, i \in [n]$, from $H$ that are underneath each filter. Write $\{\text{vec}(A(l_i))\}, i \in [n]$, into $q$ starting at index $i_k$.
4. Each part filter is placed at some offset $(\tilde{x}_i, \tilde{y}_i)$ from its resting position (see equation (1.3)). Write the set $\{-((\tilde{x}_i^2, \tilde{y}_i^2, \tilde{x}_i, \tilde{y}_i))\}$, $i = 2, \ldots, n$, into $q$, continuing from where the previous step ended.

The resulting feature vector is somewhat sparse in the sense that all entries corresponding to the components other than $k$ are zero. We will denote the function that computes the feature vector $q$ from $H$ and $L$.
by \( q = \psi(H, L) \).

With this vector representation of the mixture model and training data, combined with our definitions of \( m_i \) and \( d_i \), we can compute the score of a model configuration simply by taking the dot product of two vectors. This formulation of the score function, as \( \beta \cdot q \), would allow us to learn the model parameters using any technique that is suitable for learning linear classifiers, if we were presented with a training set of \((q_i, y_i)\) pairs, where \( q_i \) is a feature vector and \( y_i \in \{-1, 1\} \) is each feature vector’s class label.

We choose not to assume that we have access to fully labeled training data. This assumption is practical and principled. Traditionally it has been expensive to manually label data at the level of detail that we would require (component labels and filter location labels). Even though this starting to change with the advent of services such as Amazon.com’s Mechanical Turk, which provides a marketplace for hiring inexpensive human labor for tasks such as labeling images, in principle it is much better handle weakly-labeled data. One fundamental reason is that requiring fully labeled data fixes model structure – the labels predetermine how many components may be used and what the model parts are. This predetermination eliminates the ability for the training algorithm to determine a potentially better structure. Furthermore, label consistency may deteriorate if labels are created through a large-scale, decentralized process, rather than by a small group of experts.

### 4.3.2 Weakly-Labeled Training Data

We learn mixture models from incomplete training data. We assume the following input is available for the target class.

1. A set of positive images and ground-truth bounding boxes. Each positive image contains one or more instances of the target class and a (possibly noisy) ground-truth bounding box for each instance.
2. A set of negative images. Each negative image must contain no instances of the target class.

Clearly the positive information is weak in comparison to what our models require. We approach this problem by treating the mixture component label, (rectangular) part shape, and the part configuration for each positive example as latent variables that we must solve for during model training. We fix the number of mixture components at two and the number of part filters at six per component. Part shape is initialized early in the training procedure and then fixed (see [18]). The remaining variables (mixture component label
and part locations) remain latent throughout the entire training procedure. Note that a mixture configuration completely specifies a setting of our latent variables.

### 4.3.3 Latent SVM

We use the Latent SVM (LSVM) formalism, described in [18] and [17], to learn the model parameters $\beta$. We will start by defining the LSVM objective function in the general setting, and then show how to map our problem into an LSVM problem.

For training we are given a dataset $D = \{\langle x_1, y_1 \rangle, \ldots, \langle x_N, y_N \rangle\}$ of $N$ training examples. The labels are binary with $y_i = 1$ for positive examples and $y_i = -1$ for negative examples. The examples may be a heterogeneous mix of highly structured objects. Furthermore, we assume there is a set of latent variables that may be used to extract feature vectors from the examples. Ultimately, the LSVM procedure will learn a linear classifier, $\beta$, from extracted feature vectors. First, define the score of an example $x_i$ as

$$f_\beta(x_i) = \max_{z \in \mathcal{Z}(x_i)} \beta \cdot \phi(x_i, z). \quad (4.4)$$

Here $\mathcal{Z}(x_i)$ is the set of all valid values that the latent variables are allowed to take on for example $x_i$. The function $\phi(x_i, z)$ takes a particular setting of the latent variables $z$ and computes a feature vector of the same dimension as $\beta$ from the example $x_i$. Now we can define the LSVM objective function as

$$\beta^{*}(D) = \arg\min_{\beta} \frac{\lambda}{2} \|\beta\|^2 + \sum_{(x,y) \in D} \max(0, 1 - y f_\beta(x)) \quad (4.5)$$

The LSVM objective function has a key property called semi-convexity: if we restrict $\mathcal{Z}(x_i)$ to contain only one setting of the latent variables for each positive example $x_i$, then equation (4.5) becomes convex in $\beta$. This gives rise to a coordinate descent algorithm in which we alternate between the following two steps:

1. Holding the model $\beta$ fixed, solve the $\max$ problem in equation (4.4) for each positive example in $D$.
2. Holding the positive latent variables fixed, optimize the LSVM objective function, which is now convex in $\beta$.

This iterative procedure converges to a strong local minima of the LSVM objective function as described...
in [18]. We proceed by mapping the problem of training a mixture of multiscale star models into an LSVM problem in the following way.

1. For positive examples, \( x_i = (I_{x_i}, B_{x_i}) \), where \( I_{x_i} \) is a positive image and \( B_{x_i} \) is a ground-truth bounding box for an instance of the target class in \( I_{x_i} \) (see Figure 4.3 A).

2. For negative examples, \( x_i = (I_{x_i}, l^{(1)}_{x_i}, \ldots, l^{(C)}_{x_i}) \), where \( I_{x_i} \) is a negative image and \( l^{(1)}_{x_i}, \ldots, l^{(C)}_{x_i} \) are root filter locations, one for each component, such that when the root filters are placed according to these locations their geometric centers coincide (see Figure 4.3 C). Note that there are many ways to define negative examples. We believe that it is important to select a definition that causes the components to compete against each other in the maximization of equation (4.4).

3. Our latent variables are: the mixture component label and the part locations. A setting of our latent variables is equivalent to a mixture configuration and therefore \( Z(x_i) \subseteq [C] \times \mathcal{L}^n \).

4. For positive examples, the set \( Z(x_i) \) is defined such that for each configuration \( L \in Z(x_i) \), the root filter, placed according to \( L \), has significant overlap (70% or more) with \( x_i \)'s ground-truth bounding box (see Figure 4.3 B).

5. For negative examples, the set \( Z(x_i) \) is defined such that for each configuration \( L \in Z(x_i) \), the component label and root filter location are restricted to the set \( \{(1, l^{(1)}_{x_i}), \ldots, (C, l^{(C)}_{x_i})\} \). All other part locations are unrestricted (though they must yield a valid configuration with respect to the root filter location, see section 1.2).

6. For an example \( x_i \) and a configuration \( L \), we define \( \phi(x_i, L) = \psi(H(x_i), L) \), where \( H(x_i) \) is a function that returns the feature pyramid computed from \( I_{x_i} \).

7. The max problem in equation (4.4) is exactly the matching problem for mixtures of multiscale star models and can be solved using the algorithms discussed in this paper.

This completes the mapping from our model-training problem into the LSVM framework. It is important to note that for each negative image there is a very large number of negative examples – essentially one for each location in \( H(x_i) \). Furthermore, for each positive and negative example, the set \( Z(x_i) \) is exponentially large in \( n \). As a consequence, in practice there are far too many feature vectors to execute step 2 as a simple convex optimization procedure.
Figure 4.3: Example training instances. (A) A positive example $x^+$ for the class bicycle; (B) a valid configuration from $Z(x^+)$; (C) the bounding boxes corresponding to a negative example (with respect to the class bicycle) where each component’s root filter is placed such that they share the same center.

To overcome this obstacle, we use an iterative caching procedure that is guaranteed to converge to the global optimum in step 2. This procedure, which is given in detail along with a convergence proof in [17], builds a cache of feature vectors by carefully selecting configurations from the available pool $\{Z(x_i)\}$.

For each positive example, the solitary feature vector selected in step 1 is added to the cache (this will make the objective function convex in $\beta$). For negative examples, hard negative features vectors are found by data mining. Given a cache of finite size, the data mining procedure loops over the negative images and harvests feature vectors $q$ (one per negative example) that maximize equation (4.4) and score on, or within, the margin of the current classifier $\beta$ (i.e., $\beta \cdot q \geq -1$). This procedure is exactly the application of the thresholded matching problem to each negative image with $T = -1$.

Once the cache is full (or there are no more hard negative feature vectors to data mine), we use stochastic gradient descent to minimize the LSVM objective function limited to the feature vectors in the current cache. We then use the new model returned by the gradient descent procedure to score the negative feature vectors in the cache. We shrink the cache by removing correctly classified negative feature vectors that score strictly outside of the margin. Following shrinking, we attempt to grow the cache by performing another
round of data mining. This process is guaranteed to terminate and converge to the global minimum of the LSVM objective function for the full dataset restricted to the current fixed positive latent values (see [17]). In practice we set \( T = -(1 + \epsilon) \) to avoid missing support vectors due to optimization error. We also add a regularized bias parameter to each mixture component (in gradient descent the bias parameters are regularized with a very low learning rate). This per component bias term is very important as it allows components to learn comparable scores.

4.3.4 Initialization and Training Phases

We need an initial mixture model before we can use LSVM. Our initialization method and the subsequent two phases of LSVM training are described below. Together these three phases comprise our complete training algorithm.

Phase 1: Root filter initialization. We start by splitting the positive instances into two groups, those with a bounding box aspect ratio below the median and those with a bounding box aspect ratio above the median. For initialization, the first group is assigned to component 1 and the second group is assigned to component 2. We use aspect ratio as a simple indicator of extreme intra-class variation.

For each component, we select the assigned group’s median aspect ratio to be the root filter’s aspect ratio, and select the filter’s size to be the largest size not larger than 80\% of the assigned bounding boxes. To train the initial root filters, we collect positive feature vectors by warping the image regions under each ground-truth bounding box to match the size and shape of the root filters. We compute the feature image of each warped image and use its vector representations as a positive feature vector. We randomly select negative feature vectors from negative images. The initial root filters are trained independently using the standard SVM objective function and no latent variables. We force root filters to be symmetric about the vertical axis.

Phase 2: Merging components. We combine the initial root filters into a mixture-of-root-filters model and retrain the mixture using LSVM with the root location and component label as the only latent variables.

Phase 3: Retraining with parts. We use the part initialization scheme from [18] to add parts to each mixture component. Then we retrain the full mixture model using LSVM, this time with the full set of latent variables: component label, root location, and part locations. Parts that are centered on the vertical axis of a
root filter are vertically symmetric. Parts that are placed off center always have a vertically mirrored partner part (see Figure 4.2).

We have found that limiting the number of negative images to 100 during all but the last iteration of the coordinate descent procedure produces models that perform as well as if all negative images were available during all of the iterations. This significantly speeds up training. We enforce the positively constraint for the deformation parameters $a_{xi}$ and $a_{yi}$ by lower-bounding them by 0.01 during gradient descent.
Chapter 5

Training Mixture Models with Heuristic Coarse-to-Fine Matching

We seek to prove that heuristic coarse-to-fine matching with inadmissible heuristics and thresholds can be used successfully in practice. We also wish to quantify the effects of our new matching algorithm on training and testing speed.

There are four main issues with using HCTFDetect in the system presented in the previous chapter. First of all, we must extend HCTFDetect to work with mixture models. Second, our system will need to initialize the heuristics, thresholds, and part order (referred to as the HCTF parameters from here on) prior to having a model with which score statistics can be collected. The third issue is how to update the HCTF parameters efficiently during the training procedure. Finally, we consider some opportunities that switching to HCTFDetect provides for further improving performance in terms of speed and detection accuracy. We address these issues with the following modifications:

1. HCTFDetect can be extended to work with mixture models by maintaining separate HCTF parameters for each component and by expanding the configuration space, as was done in the previous chapter.
2. Initialize the heuristic functions and pruning thresholds to admissible values prior to entering phase 3 of training.
3. After running gradient descent on the current training cache, use the cache to compute the statistics
that are necessary for updating the HCTF parameters for the next training iteration.

4. Freeze the root filter after phase 2 of training.

5. Use non-maximal suppression during phase 3 of training.

The first three modifications are necessary to handle initialization and maintenance of the heuristics, thresholds, and part ordering. The third and fourth changes are introduced to see if they will speed up HCTFDetect or improve detection accuracy.

5.1 Mixture Models and Heuristic Coarse-to-Fine Detection

Our detection algorithm, HCTFDetect, can be easily extended to work with the mixture models introduced in the previous chapter. Only two small changes are required, which we sketch here. First, since a mixture model is just a collection of $C$ models of the type described in section 1.2, we can simply maintain a separate set of HCTF parameters for each mixture component. Second, we work with configurations in $[C] \times L^n$ instead of just $L^n$: change the initialization loop (line 8 of HCTFDetect) to iterate over component, root-location pairs $(k, l_1)$, and throughout the remainder of the algorithm use the component specific thresholds and heuristics as appropriate.

It is interesting to note that HCTFDetect will perform non-maximal suppression between mixture components during detection. This is an example of competition between different derivations from the simple grammar model. We see that one derivation can win before others have been completed, further reducing computation.

To keep the AND/OR graph interpretation in sync with mixture models, we must make two small changes: (1) replicate the graph in Figure 2.2 once for each mixture component, and (2) add a new root node and attach it with a single edge to the root of each component’s graph. Note that even though we are adding an additional level there are no new AND connectors in our construction. This implies that we can continue using a single priority queue to compute the max at the new root (which takes a max over maxes).
5.2 Initialization

Our training algorithm benefits greatly from using two distinct sets of HCTF parameters. The first set is used when updating the latent positive variables in step 1 of the LSVM coordinate descent algorithm. The second set is used for hard negative data mining. These two problems involve very different data, so it is natural to expect that specializing the HCTF parameters for each task is important.

At the end of phase 2, the training algorithm outputs a model that is a mixture of one root filter per component. At the start of phase 3, the training algorithm initializes each component’s parts to produce a mixture model where each component is a star graph of the form detailed in section 1.2. In order to proceed with HCTFDetect, we need to give each component initial values for $h_j$, $t_j$, and a part ordering.

We initialize by selecting values for $h_j$ and $t_j$ that are known to be admissible for all components and for both sets of parameters.

\begin{align}
    h_j &= \infty \text{ if } j \in [n - 1], \text{ and } 0 \text{ if } j = n, \quad (5.1) \\
    t_j &= -\infty \text{ for } j \in [n]. \quad (5.2)
\end{align}

The initial part order for each component is: root filter first, followed by an arbitrary ordering of the component’s part filters. Since we are starting with a heuristic that will lead to breadth-first search and a pruning threshold that will not remove any partial configurations, the part ordering has no effect on performance.

This initialization is used when computing the first cache of positive feature vectors. A small change is required for the first round of hard negative data mining: set $t_n = -(1 + \epsilon)$ in order to capture only hard negative feature vectors. In practice, running HCTFDetect with these initial values is slightly slower than using the dynamic programming algorithm. If desired, one could use the dynamic programming approach in place of HCTFDetect for this step.

5.3 Updating Heuristics, Thresholds, and Part Order

At the end of each execution of the gradient descent procedure, we compute updated values for $t_j$, $h_j$, and a new part ordering for all components and for both sets of parameters. Recall that the gradient descent
procedure operates on a cache of positive and negative feature vectors. When gradient descent terminates, we are given a new model. Using this model, we can score each feature vector in the cache. Because the feature vectors have internal structure, we can record the total score as well as the individual part scores, including deformation costs: \( m_i(l_i) - d_i(l_1, l_i) \).

To update the parameters for latent positive detection, we consider only the positive cache entries. We view these feature vectors as quantities derived from a sample generated by \( D \). Using the part scores associated with each positive feature vector, we can create samples from \( D^*_j \). Using these samples, we fix a new part ordering for each model component. Experimentally we found that the two proposed methods of ordering parts yield substantially equivalent results. Given this finding, we chose to use the simpler method in which parts are greedily selected to yield the tightest expected heuristic. After fixing the part order, we compute new values of \( h_j \) and \( t_j \) using equations (3.6) and (3.11).

We follow the same procedure to update the parameters for hard negative data mining. The only difference is the sample that we extract from the cache. In this case, we use all negative feature vectors that score greater than \(- (1 + \epsilon)\) with respect to the new model.

At the end of the phase 3, we select the final heuristics, thresholds, and part ordering that will be used on test images. We use the same method as when we set the HCTF parameters for latent positive detection, except that we shrink the sample slightly. Instead of using all positive feature vector from the cache, we select the top scoring 98%. In general, the training cache will contain a few positive feature vectors on which our model scores very poorly. If we include these examples in the sample that we use to generate the final parameters, then the pruning thresholds will be far too low, which will lead to a slow detection algorithm with a very high false positive rate. To compare the performance of HCTFDetect to the dynamic programming algorithm, we use the same top scoring 98% of positive feature vectors as the criterion for determining the recall threshold of the final detector. It should be noted that even on the best performing classes state-of-the-art models typically only reach about 70-75% recall on the PASCAL 2007 & 2008 test data. This suggests that we could lower the top-98% criterion without affecting the AP score of the classifier by very much. In the case of HCTFDetect, this change will make the algorithm even faster. In contrast, this change will have no effect on the performance of the dynamic programming algorithm. During training, in contrast, we select all positive feature vectors for our sample because we want to train with as much positive data as
possible.

A final consideration is to note that we are using training data to determine the HCTF parameters. In general this is not a safe approach. Ideally we would like to use a hold-out validation set for this task. We avoid using a validation set for two reasons. The first is that some classes have very few positive examples (about 300) and carving a validation set out of the training data could produce sets that are too small for both training the model and estimating good HCTF parameters. Second, our goal is to use HCTFDetect without adding much overhead. The existing system already scores the entire cache at the end of gradient descent (which is a very fast operation), by using a slight extension to this cache-scoring step (i.e., extracting part scores), we avoid adding the overhead of executing HCTFDetect on separate positive and negative validation sets. The concern here is that the detection parameters might be poor due to model over-fitting, but this concern has been abated by experimental results which suggest that there is very little over-fitting in practice.

### 5.4 Fixed Root Filters

We would like to test the hypothesis: can we improve the speed of HCTFDetect by freezing the root filters after phase 2 of the training algorithm? The intuition behind this question is that after phase 2, we should have learned a mixture of root filters that is optimal (or within optimization error of optimal) for the mixture-of-root-filters family of LSVM classifiers. This suggests that our most abstract model, the one that contains only root filters, may be better at filter candidate configurations than an abstract model that was modified during phase 3 of training.

To test this hypothesis we introduce the following change. After phase 2 of training, we consider the root filter for each component to be an immutable object. Then we add a new parameter, to each model component, that acts as a weight controlling how much each root filter contributes to the overall model score.

To learn this new model structure, in phase 3 we must correspondingly change the structure of our feature vectors. Instead of each vector containing the HOG features under the placement of a root filter, the new feature vectors contain a new scalar computed by the dot product of the frozen root filter with the HOG features under it. One way to think about this new formulation is that for each frozen root filter, we
generate a new feature pyramid that stores the score of the frozen root filter at each translation and scale of the original feature pyramid. We then build a new classifier using all of these feature pyramids. Freezing the root filter also has the advantage of reducing the number of parameters in the model during phase 3, which makes gradient descent slightly faster.

5.5 Data Mining with Non-maximal Suppression

The final modification that we make is to use non-maximal suppression during phase 3 of training. The motivation behind this idea is that during the initial hard negative data mining, nearly every valid root location in a negative image will give rise to a hard negative feature vector, but the feature vectors found at neighboring root locations will be very similar. It seems more natural to only use hard negative feature vectors that are locally optimal, which would allow us to fill the training cache with a more diverse sample of examples. We test this hypothesis in the experimental results chapter to see if it produces better models.
Chapter 6

Experimental Results

We use the PASCAL 2007 VOC dataset to evaluate the effectiveness of applying heuristic coarse-to-fine search to our object recognition system. Our primary goal is to demonstrate that HCTFDetect can be used successfully with our existing class of mixture models. We also hope to see incremental improvements in detection speed, though we expect that the true benefit of our new matching algorithm will be when it is applied to richer grammar model.

6.1 Datasets

At the time of this writing, the 2007 version of the PASCAL VOC dataset is the most recent release that includes ground truth for the test set (ground truth for the 2008 test set is not available to researchers so that it may be reused in future challenges). When comparing results obtained using the PASCAL VOC dataset it is important to note that the 2007 dataset is significantly more difficult than either the 2005 or 2006 releases. Scores cannot be compared directly across these datasets. Using this data, we are interested in answering the following three questions:

1. How is performance (computation time and average precision) affected by replacing the dynamic programming matching algorithm with HCTFDetect?
2. What effects can we observe if the root filters are frozen prior to phase 3 of the training algorithm?
3. Does training with non-maximal suppression yield better performance?
It is important to remember that our models are random functions. Randomness enters from several sources: (i) our training data is a random sample; (ii) we initialize our models with random negative feature vectors in phase 1; (iii) we use stochastic gradient descent to optimize our objective function. The first source of randomness is irrelevant to our experimental protocol because the training data is the same for all experiments. We control the effect of the second source of randomness – random initialization – by executing training phases 1 & 2 once for each class, and then we use those initial models as the starting point for phase 3 in all of our experiments. The third source of randomness – stochastic gradient descent – is left as a random effect in our experiments.

### 6.2 Protocol

Our experimental setup is:

1. For each PASCAL 2007 object class, execute training phases 1 & 2 and save the resulting initial models.

2. Construct four versions of our system:

   (a) **DP**: the baseline system that uses the dynamic programming algorithm and distance transforms for solving the thresholded matching problem.

   (b) **HCTF**: the previous system modified to use *HCTFDetect* for solving the (thresholded) matching problem for fixing positive latent variables and data mining.

   (c) **HCTF + Frozen**: the previous system modified to use frozen root filters.

   (d) **HCTF + NMS**: HCTF modified to use non-maximal suppression during hard negative data mining.

3. For each PASCAL 2007 object class, starting with the initial model, execute phase 3 of the training algorithm using each of the four systems.

4. For each PASCAL 2007 object class, using the final model learned in the previous step, evaluate average precision and computation time on the PASCAL 2007 VOC test dataset. We use the PASCAL *comp3* testing protocol for computing average precision.
6.3 Discussion of Findings

The results for a subset of these classes are presented in tables 6.1, 6.2, and 6.5. For these timings, training and testing were performed using a single 2.66GHz Intel Xeon CPU. The feature-vector cache used in gradient descent was limited to 512MB.

Our most significant finding, answering the first question, is that by using system HCTF testing time can be reduced by a factor of 2 to 3 for many classes while maintaining comparable AP scores. Training time can also be reduced, but the effects are somewhat limited in comparison to the results found during the testing phase. This is due to two factors. The first is that HCTFDetect performs slightly slower than the the baseline matching algorithm when computing positive latent labels. The reason for this is that the bounding box overlap requirement allows us to shrink the set of valid part locations to a very small size (by cropping the input image). In this regime, the additional overhead associated with HCTFDetect and fact that it does not use distance transforms, dwarfs any performance gains that we would hope to see. This issue is most apparent for the class person, which has by far the most positive examples at more than 9000 (we use each positive example and its vertical mirror). Given this, it is reasonable to argue that our system should rely on both matching algorithms, so that during training time we are using an exact algorithm for computing positive latent labels. The second factor is that the gradient descent procedure, which is not affected by HCTFDetect, consumes approximately one-third of the total training time on average. The time consumption dampens the visible effect of speeding up hard negative data mining.

Regarding our second question, using a frozen root filter nearly always improves HCTFDetect’s speed, but often comes with a significant cost in terms of reduced AP performance. Surprisingly, using NMS during training does not appear to improve results. One possible explanation for this finding is that we are running a sufficient number of coordinate descent iterations to converge to a local minima of our objective function. It remains to be seen if using NMS will allow us to reduce the number of coordinate descent iterations (converge faster) while maintaining the same AP performance level.
<table>
<thead>
<tr>
<th>class</th>
<th>DP</th>
<th>HCTF</th>
<th>HCTF + Frozen</th>
<th>HCTF + NMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>aeroplane</td>
<td>4.47h</td>
<td>3.71h</td>
<td>3.21h</td>
<td>3.84h</td>
</tr>
<tr>
<td>bicycle</td>
<td>4.64h</td>
<td>3.76h</td>
<td>3.33h</td>
<td>3.81h</td>
</tr>
<tr>
<td>bottle</td>
<td>1.76h</td>
<td>2.25h</td>
<td>1.66h</td>
<td>2.48h</td>
</tr>
<tr>
<td>bus</td>
<td>4.81h</td>
<td>3.92h</td>
<td>3.56h</td>
<td>3.78h</td>
</tr>
<tr>
<td>car</td>
<td>4.25h</td>
<td>4.74h</td>
<td>4.44h</td>
<td>4.64h</td>
</tr>
<tr>
<td>cow</td>
<td>3.57h</td>
<td>3.05h</td>
<td>2.75h</td>
<td>3.05h</td>
</tr>
<tr>
<td>horse</td>
<td>3.81h</td>
<td>3.44h</td>
<td>3.00h</td>
<td>3.63h</td>
</tr>
<tr>
<td>motorbike</td>
<td>4.50h</td>
<td>3.95h</td>
<td>3.83h</td>
<td>4.24h</td>
</tr>
<tr>
<td>person</td>
<td>6.12h</td>
<td>6.48h</td>
<td>6.74h</td>
<td>7.10h</td>
</tr>
<tr>
<td>sheep</td>
<td>3.00h</td>
<td>2.76h</td>
<td>2.69h</td>
<td>3.08h</td>
</tr>
<tr>
<td>train</td>
<td>5.16h</td>
<td>4.42h</td>
<td>3.97h</td>
<td>4.47h</td>
</tr>
<tr>
<td>tvmonitor</td>
<td>3.25h</td>
<td>2.95h</td>
<td>2.58h</td>
<td>3.14h</td>
</tr>
</tbody>
</table>

Table 6.1: Time spent in phase 3 of the training algorithm for a subset of the PASCAL 2007 object classes. For each class, we report data for four versions of our system: **DP** = dynamic programming and distance transforms for matching; **HCTF** = hierarchical coarse-to-fine detection; **Frozen** = root filters are frozen after phase 2; **NMS** = hard-negative data mining uses non-maximal suppression. For each class/version, the associated pair of cells reports the phase 3 training time in hours and the speedup factor relative to the baseline system **DP**.

<table>
<thead>
<tr>
<th>class</th>
<th>DP</th>
<th>HCTF</th>
<th>HCTF + Frozen</th>
<th>HCTF + NMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>aeroplane</td>
<td>5.70h</td>
<td>3.86h</td>
<td>2.22h</td>
<td>3.47h</td>
</tr>
<tr>
<td>bicycle</td>
<td>5.79h</td>
<td>2.37h</td>
<td>2.03h</td>
<td>2.57h</td>
</tr>
<tr>
<td>bottle</td>
<td>4.54h</td>
<td>2.28h</td>
<td>1.84h</td>
<td>2.62h</td>
</tr>
<tr>
<td>bus</td>
<td>5.75h</td>
<td>2.85h</td>
<td>2.85h</td>
<td>2.84h</td>
</tr>
<tr>
<td>car</td>
<td>4.37h</td>
<td>3.82h</td>
<td>4.23h</td>
<td>4.25h</td>
</tr>
<tr>
<td>cow</td>
<td>5.43h</td>
<td>3.36h</td>
<td>3.53h</td>
<td>2.79h</td>
</tr>
<tr>
<td>horse</td>
<td>6.00h</td>
<td>4.27h</td>
<td>2.93h</td>
<td>3.35h</td>
</tr>
<tr>
<td>motorbike</td>
<td>6.01h</td>
<td>2.21h</td>
<td>2.12h</td>
<td>2.33h</td>
</tr>
<tr>
<td>person</td>
<td>4.95h</td>
<td>4.45h</td>
<td>4.72h</td>
<td>4.83h</td>
</tr>
<tr>
<td>sheep</td>
<td>4.81h</td>
<td>2.85h</td>
<td>2.52h</td>
<td>3.51h</td>
</tr>
<tr>
<td>train</td>
<td>6.59h</td>
<td>2.54h</td>
<td>2.49h</td>
<td>2.58h</td>
</tr>
<tr>
<td>tvmonitor</td>
<td>9.63h</td>
<td>3.07h</td>
<td>2.32h</td>
<td>4.14h</td>
</tr>
</tbody>
</table>

Table 6.2: Time spent processing 4952 test images for a subset of the PASCAL 2007 object classes. For each class and each version of the system (see Table 6.1 caption), the associated pair of cells reports the total testing time in hours and the speedup factor relative to the baseline system **DP**.
### PASCAL 2007 Average Precision (without Bounding Box Prediction)

<table>
<thead>
<tr>
<th>class</th>
<th>DP</th>
<th>HCTF</th>
<th>HCTF + Frozen</th>
<th>HCTF + NMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>aeroplane</td>
<td>0.281</td>
<td>0.285</td>
<td>0.250</td>
<td>0.272</td>
</tr>
<tr>
<td>bicycle</td>
<td>0.564</td>
<td>0.539</td>
<td>0.535</td>
<td>0.529</td>
</tr>
<tr>
<td>bottle</td>
<td>0.269</td>
<td>0.261</td>
<td>0.254</td>
<td>0.274</td>
</tr>
<tr>
<td>bus</td>
<td>0.437</td>
<td>0.443</td>
<td>0.460</td>
<td>0.446</td>
</tr>
<tr>
<td>car</td>
<td>0.465</td>
<td>0.464</td>
<td>0.459</td>
<td>0.466</td>
</tr>
<tr>
<td>cow</td>
<td>0.176</td>
<td>0.165</td>
<td>0.163</td>
<td>0.188</td>
</tr>
<tr>
<td>horse</td>
<td>0.438</td>
<td>0.432</td>
<td>0.442</td>
<td>0.433</td>
</tr>
<tr>
<td>motorbike</td>
<td>0.384</td>
<td>0.397</td>
<td>0.355</td>
<td>0.387</td>
</tr>
<tr>
<td>person</td>
<td>0.332</td>
<td>0.336</td>
<td>0.353</td>
<td>0.348</td>
</tr>
<tr>
<td>sheep</td>
<td>0.195</td>
<td>0.190</td>
<td>0.159</td>
<td>0.189</td>
</tr>
<tr>
<td>train</td>
<td>0.340</td>
<td>0.371</td>
<td>0.319</td>
<td>0.371</td>
</tr>
<tr>
<td>tvmonitor</td>
<td>0.384</td>
<td>0.370</td>
<td>0.363</td>
<td>0.360</td>
</tr>
</tbody>
</table>

Table 6.3: Average precision score for a subset of the PASCAL 2007 object classes on the PASCAL 2007 VOC test dataset. For each class and each version of the system (see Table 6.1 caption), the associated pair of cells reports the average precision (AP) score and the percent change in AP relative to the baseline system DP.

### PASCAL 2007 Average Precision (with Bounding Box Prediction)

<table>
<thead>
<tr>
<th>class</th>
<th>DP</th>
<th>HCTF</th>
<th>HCTF + Frozen</th>
<th>HCTF + NMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>aeroplane</td>
<td>0.277</td>
<td>0.272</td>
<td>0.250</td>
<td>0.264</td>
</tr>
<tr>
<td>bicycle</td>
<td>0.567</td>
<td>0.554</td>
<td>0.553</td>
<td>0.542</td>
</tr>
<tr>
<td>bottle</td>
<td>0.272</td>
<td>0.263</td>
<td>0.260</td>
<td>0.288</td>
</tr>
<tr>
<td>bus</td>
<td>0.443</td>
<td>0.451</td>
<td>0.456</td>
<td>0.452</td>
</tr>
<tr>
<td>car</td>
<td>0.485</td>
<td>0.477</td>
<td>0.488</td>
<td>0.479</td>
</tr>
<tr>
<td>cow</td>
<td>0.172</td>
<td>0.148</td>
<td>0.158</td>
<td>0.187</td>
</tr>
<tr>
<td>horse</td>
<td>0.432</td>
<td>0.451</td>
<td>0.443</td>
<td>0.432</td>
</tr>
<tr>
<td>motorbike</td>
<td>0.399</td>
<td>0.408</td>
<td>0.366</td>
<td>0.394</td>
</tr>
<tr>
<td>person</td>
<td>0.343</td>
<td>0.345</td>
<td>0.358</td>
<td>0.354</td>
</tr>
<tr>
<td>sheep</td>
<td>0.201</td>
<td>0.182</td>
<td>0.159</td>
<td>0.185</td>
</tr>
<tr>
<td>train</td>
<td>0.347</td>
<td>0.383</td>
<td>0.320</td>
<td>0.375</td>
</tr>
<tr>
<td>tvmonitor</td>
<td>0.386</td>
<td>0.372</td>
<td>0.363</td>
<td>0.371</td>
</tr>
</tbody>
</table>

Table 6.4: Scores from Table 6.5 after applying the bounding-box prediction procedure described in [17].
Figure 6.1: Precision-recall curve for the INRIA Person dataset [10] using system HCTF. System DP produced an AP score of 0.878 with a testing time of 40.2 minutes on 288 test images. The testing time for system HCTF was 2.67 times faster, testing all 288 images in 15.0 minutes. Testing was conducted using the PASCAL comp3 testing protocol with the original INRIA bounding boxes.

<table>
<thead>
<tr>
<th>$M_i$</th>
<th>aero</th>
<th>bike</th>
<th>bottle</th>
<th>bus</th>
<th>car</th>
<th>cow</th>
<th>horse</th>
<th>mbike</th>
<th>person</th>
<th>sheep</th>
<th>train</th>
<th>tv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55.8%</td>
<td>88.4%</td>
<td>58.5%</td>
<td>67.5%</td>
<td>35.3%</td>
<td>72.9%</td>
<td>51.3%</td>
<td>86.1%</td>
<td>32.8%</td>
<td>38.4%</td>
<td>75.5%</td>
<td>60.3%</td>
</tr>
<tr>
<td>2</td>
<td>11.0%</td>
<td>8.9%</td>
<td>29.2%</td>
<td>28.2%</td>
<td>39.7%</td>
<td>14.7%</td>
<td>22.9%</td>
<td>11.6%</td>
<td>19.7%</td>
<td>41.8%</td>
<td>20.2%</td>
<td>17.7%</td>
</tr>
<tr>
<td>3</td>
<td>9.7%</td>
<td>1.4%</td>
<td>9.7%</td>
<td>2.3%</td>
<td>10.9%</td>
<td>9.7%</td>
<td>16.8%</td>
<td>1.4%</td>
<td>16.3%</td>
<td>17.3%</td>
<td>2.9%</td>
<td>15.0%</td>
</tr>
<tr>
<td>4</td>
<td>9.8%</td>
<td>1.0%</td>
<td>1.6%</td>
<td>0.7%</td>
<td>6.2%</td>
<td>2.6%</td>
<td>6.7%</td>
<td>0.6%</td>
<td>14.6%</td>
<td>2.1%</td>
<td>0.9%</td>
<td>5.9%</td>
</tr>
<tr>
<td>5</td>
<td>5.3%</td>
<td>0.2%</td>
<td>0.9%</td>
<td>0.9%</td>
<td>6.4%</td>
<td>0.1%</td>
<td>1.7%</td>
<td>0.3%</td>
<td>11.6%</td>
<td>0.2%</td>
<td>0.3%</td>
<td>0.5%</td>
</tr>
<tr>
<td>6</td>
<td>8.4%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.4%</td>
<td>0.8%</td>
<td>0.0%</td>
<td>0.5%</td>
<td>0.0%</td>
<td>3.7%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>7</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.1%</td>
<td>0.6%</td>
<td>0.0%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>1.3%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Table 6.5: Pruning efficiency by abstraction level for the system HCTF. For a given test image, a detector may report a detection at any of the valid (root location, component label) pairs. Each table cell reports the average percentage of candidate detection pairs that are pruned at each abstraction level for a given class. For example, on average the bicycle model will prune 88.4% of all candidate detection locations after applying the model’s root filters. An additional 8.9% of the total is pruned after applying the first part from each mixture component – this leaves less than 3% of the candidate locations unpruned after evaluating only 4 (of 14) filters.
Figure 6.2: Visualization of locations where part filter scores ($m_i(l)$) were computed for one level of the feature pyramid for image (a). Pane (b) shows the top scoring detection of the train model. Panes (c) - (h) show the locations where $m_i(l)$ was computed for each part filter for the first train mixture component. White cells represent areas where $m_i(l)$ was computed and black cells areas where the score was not computed. For reference, note that the square block in (g) is 11x11 HOG cells. Panes (i) - (n) show the same for the second component. These pictures show information for the scale in which the detection in (b) was found.
Chapter 7

Conclusions

We have presented a new algorithm for approximately solving the thresholded matching problem for a highly relevant class models. Our algorithm is a blend of heuristic search, in the style of A* and AO*, and a traditional coarse-to-fine process. We present our algorithm in an exact form that assumes the choice of admissible thresholds and heuristics. In practice, however, choosing admissible parameters is too conservative and, in fact, we show a very simple method for automatically selecting inadmissible heuristics and thresholds that significantly increases detection speed (by a factor of 3 for some classes). We give theoretical results that justify the selection of these inadmissible heuristics and thresholds in the form of a PAC-style guarantee on the error rate due to inadmissibility. Furthermore, we provide experimental results on the PASCAL 2007 VOC Challenge dataset demonstrating how our algorithm performs on real-world images. We show that even when using inadmissible heuristics, training and testing times can be reduced while maintaining state-of-the-art average precision performance on rigid and semi-rigid object classes. We also note that the heuristic coarse-to-fine approach, driven by sample-based heuristics and thresholds, is a general algorithmic framework that may be applied to other interesting problems.

Our contribution is significant in that it leads the way forward on the path from simple pictorial structure models to rich visual grammars. In future work we plan to develop the next iteration of model enrichment, which will likely include mixture models with more components and shared parts (imagine a single wheel part shared between left, right, and 45-degree rotated side views of cars). Later we will investigate models where each part is a mixture model, and where each part may be composed of sub-parts.
It is also clear that we need to spend more time perfecting the application of HCTFDetect to our object recognition system. The method of selecting thresholds based on the top-98% of positives tends to produce thresholds that are too conservative. For example, by reducing the criterion from 98% to 85% for car we decreased the testing time from 3.82h to 3.06h in conjunction with a mild AP decrease of 0.477 to 0.465. Furthermore, this loss in AP is almost entirely from the low-precision, high-recall region of the PR curve, which is the least useful regime from the viewpoint of application.
Bibliography


