International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:10, No:10, 2016

Faster Pedestrian Recognition Using Deformable Part Models

Alessandro Preziosi, Antonio Prioletti, Luca Castangia

Abstract—Deformable part models achieve high precision in pedestrian recognition, but all publicly available implementations are too slow for real-time applications. We implemented a deformable part model algorithm fast enough for real-time use by exploiting information about the camera position and orientation. This implementation is both faster and more precise than alternative DPM implementations. These results are obtained by computing convolutions in the frequency domain and using lookup tables to speed up feature computation. This approach is almost an order of magnitude faster than the reference DPM implementation, with no loss in precision. Knowing the position of the camera with respect to horizon it is also possible prune many hypotheses based on their size and location. The range of acceptable sizes and positions is set by looking at the statistical distribution of bounding boxes in labelled images. With this approach it is not needed to compute the entire feature pyramid: for example higher resolution features are only needed near the horizon. This results in an increase in mean average precision of 5% and an increase in speed by a factor of two. Furthermore, to reduce misdetections involving small pedestrians near the horizon, input images are supersampled near the horizon. Supersampling the image at 1.5 times the original scale, results in an increase in precision of about 4%. The implementation was tested against the public KITTI dataset, obtaining an 8% improvement in mean average precision over the best performing DPM-based method. By allowing for a small loss in precision computational time can be easily brought down to our target of 100ms per image, reaching a solution that is faster and still more precise than all publicly available DPM implementations.

Keywords—Autonomous vehicles, deformable part model, dpm, pedestrian recognition.

I. Introduction

PEDESTRIAN recognition is an important task in the automotive field, both for driving assistance technologies and completely autonomous vehicles. Several algorithms achieve high precision in this task, but their execution time is not suitable for realtime applications. In this paper, we survey publicly available implementations of the "Deformable Part Model" algorithm and explore different methods to speed up detection for realtime applications. We obtain our best results using lookup tables, computing convolutions in the frequency domain, limiting our search using perspective information, and supersampling the original images near the horizon.

In Section II, we compare publicly available implementations of DPM. In Section III, we describe our implementation. We show our results in Section IV, singling out how different features improve speed and detection accuracy.

Alessandro Preziosi is with the Polytechnic University of Turin (e-mail: alessandro.preziosi@polito.it).
Antonio Prioletti and Luca Castangia are with the University of Parma (http://vislab.it).

All tests were run on a laptop equipped with an Intel Core i7-4700MQ cpu, consisting of four cores @ 2.40 GHz with 6MB of cache memory.

II. State of the Art

There are many open source implementations of DPM available online as a standalone executable or as part of a library, most are implemented in Matlab or C++. We tested most publicly available implementations of DPM against the KITTI dataset, with a particular attention to execution speed. The following sections describe these different implementations and their relative results in terms of speed.

A. Vanilla DPM

OpenCV 2.4.9 offers an implementation of a DPM classifier in the class `LatentSvmDetector`. It is based on Felzenszwalb's original paper [1], but the training part is not implemented. Models can be trained using Felzenszwalb's software [1], and imported as an xml file. We tested this implementation using a model trained on the INRIA person dataset. The results were very poor in terms of speed, OpenCV takes on average 4.38 seconds ($\pm 168ms$) to analyze a single frame form the KITTI dataset, making it unusable in a realtime scenario.

It is possible to increase the number of threads to use, but there is no speed difference for single-class detection.

B. Cascade DPM

Many detectors can be accelerated using cascade techniques, which allow pruning unpromising detection areas before computing their score entirely. This technique can be applied to DPM, as shown in [2]. We can evaluate root locations hypotheses under a simpler model, and only evaluate the high scoring ones with a richer model. These simplified models can be generated starting with the sole root filter, and sequentially adding parts.

For a model of $n + 1$ parts (n parts + 1 root filter), [2] generated a hierarchy of $2(n + 1)$ models. The first $n + 1$ models are generated by sequentially adding simplified parts, while the second $n + 1$ models are generated by sequentially replacing simplified parts with the full-resolution ones.

Other than training the model itself, we also need to train the pruning thresholds for each simplified model. Admissible thresholds would not prune any partial hypothesis that leads to a complete detection scoring above the global threshold. We

can do this by looking at the statistics of partial hypotheses scores over positive examples. This leads to a simple method of picking thresholds depending on how big an error we are willing to accept.

At any stage i of the partial score computation, we can prune the hypothesis using two methods:

- *Hypothesis pruning*: We add to the current partial score the score of additional part i, and if the result is lower than a threshold t_i we prune the hypothesis
- *Deformation pruning*: For each location of additional part i, we calculate the deformation cost and subtract it form the partial score. If the result is lower than a threshold t_i^1 then we avoid calculating the part score at that specific location.

This means we can prune hypotheses earlier without calculating part scores if the part would be too far away from its ideal location, resulting in a lower score.

Felzenszwalb et al. [2] report an average time for person recognition on the PASCAL dataset of $682ms$ ($+459ms$ to compute the HOG features), versus 8.5 seconds required by their reference DPM implementation [3].

C. OpenCV's Cascade DPM

Version 3.0 of the OpenCV library includes a faster version of `LatentSvmDetector`, based on the cascade approach. The new algorithm is the result of work by the NNSU team and it is based on Felzenszwalb's cascade DPM [2]. In theory this, cascade approach could speed-up execution by about one order of magnitude at a negligible loss of precision, as shown in [2]. In our tests with the KITTI dataset execution time was 1 second $\pm 113ms$, almost five times faster than the previous openCV implementation, but still not enough for realtime applications.

D. Vector Quantization

With a deformable part model, the majority of time during detection is spent calculating convolutions of the filters with the features of the input image. Much work has been done trying to speed-up this process with different methods. Sadeghi and Forsyth [4] achieved good results in speeding up template evaluation using a technique called vector quantization. Vector quantization offers speed-ups in situations where arithmetic accuracy is not crucial, and it can offer a good increase in speed at a negligible loss of precision.

Jegou et al. [5] first introduced vector quantization as a technique to approximate nearest neighbour search. They quantize space into a finite number of indexed regions and represent any vector as a short code composed of a number of subspace quantization indices. Based on this codification, they can efficiently estimate the euclidean distance between two vectors from their codes. This method can efficiently estimate the score of a template at a certain location by looking-up a number of tables, instead of multiplying vectors. Their work has been very successful as it offers two orders of magnitude speed-up with a reasonable accuracy.

In the case of DPM, if we have an $m \times n$ filter, the score of the filter at location (x, y) is given by:

$$S(x,y) = \sum_{\Delta y=1}^{m} \sum_{\Delta x=1}^{n} \mathbf{w}(\Delta x, \Delta y) \cdot \mathbf{h}(x + \Delta x - 1, y + \Delta y - 1)$$

where **w** and **h** are d dimentional vectors, containing respectively a set of weights, and the features of the source image at location *(x,y)*. We want to compute an approximation of this score where the accuracy of the approximation can be easily manipulated, so that we can trade-off speed with performance. To do so, we quantize the features in each cell **h**(*x,y*) into *c* clusters, using a basic k-means procedure, and we encode each quantized cell *q(x,y)*, using its cluster ID (from 1 to c). In order to speed-up the scoring process, we pre-compute the partial dot product of each template cell $\mathbf{w}(\Delta x, \Delta y)$ with all possible c centroids and store them in a lookup table $\mathbf{T}(\Delta x, \Delta y, i)$. We can then approximate the dot product by looking up the table:

$$S(x,y) = \sum_{\Delta y=1}^{m} \sum_{\Delta x=1}^{n} \mathbf{T}(\Delta x, \Delta y, q(x + \Delta x - 1, y + \Delta y - 1))$$

This reduces the computational complexity of exhaustive search from $\Theta(mnd)$ to $\Theta(mn)$, where *d* is typically 32. In practice 32 multiplications and 32 additions are replaced with one table lookup and one addition. This can potentially speed-up the process by a factor of 32, although in practice, table lookup is often slower than multiplication.

The vector quantization technique is compatible with a cascade approach and can be easily incorporated into a cascade detection algorithm to speed it up. While the time for calculating convolutions is reduced (so testing a huge number of templates is cheap), we pay a per-image penalty, because, after computing the HOG features, we need an extra step to compute the vector quantization.

We tested Sadeghi and Forsyth's implementation [4] against images of the KITTI dataset, using a pyramid of features with 10 intervals. The average execution time for a frame from the KITTI dataset is $1.24s$ divided as shown in Table I.

TABLE I
TIME SPENT IN EACH STEP OF THE DETECTION PROCESS FOR VECTOR-QUANTIZED DPM

Task	**Time**
HOG feature computation	854ms
Feature quantization	290ms
Detection (person)	10ms
Total:	**1.24s**

The detection time is really small thanks to the use of lookup tables. This means this technique is very advantageous if we are trying to score a lot of classes at the same time (or, using many models for a single class). Unfortunately there is a non-negligible increase in computational time required to quantize the HOG features (almost 300 milliseconds). This means that if we're trying to detect a single class, like in our case, this technique can be disadvantageous because it introduces an additional overhead in the feature computation phase.

E. Fourier Accelerated DPM

The sliding window approach requires convolving a filter in every possible position of the source image, and computing all the convolutions is a major bottleneck for DPM. There is an easy way to speed-up this process, considering that convolutions are equivalent to products in the frequency domain we can compute a Fourier Transform of the input and afterwards calculate cheap dot products instead of convolutions, as suggested by Doubout and Fleuret [6].

We do a Fourier Transform of both the feature planes, and the filter planes. Then, these are pointwise multiplied, which is equivalent to a convolution in the spatial domain. Finally, instead of computing an inverse transform for every plane, and later adding up the score of each plane, we can sum these scores in the frequency domain, thanks to the linearity of the Fourier Transform, and then compute only one inverse transform. This saves a lot of computational time, especially if the filters are big. The transform of the filters can be done offline, so for each frame we only need $L + 1$ transforms, where L is the number of HOG planes.

Charles Dubout's implementation, without modifications, takes $465 \pm 33ms$ for an image of the KITTI dataset.

The computation of the HOG features is also faster compared to OpenCV's implementation, as it uses a lookup table to calculate the gradient angles. This is based on the fact that pixels of an image have a limited range of values, from 0 to 255, therefore a gradient in the x or y direction is limited to 511 integers $[-255, 255]$. We can therefore precompute a lookup table of 511x511 elements, to speed-up the computation of the gradient orientation.

The calculation of HOG features in openCV's implementation takes on average $85 \pm 7ms$. Using lookup tables it takes $51 \pm 14ms$. If we parallelize the calculation of features for different scales we can get it down to $38 \pm 12ms$.

According to [7], it is possible to further speed-up HOG features computation and get it down to about $7ms$, by using lookup tables also for additional steps of the computation (a total of three lookup tables).

III. Our Implementation

Our final implementation is largely based on Dubout's version [6], using Fourier transform to compute convolutions. FFTW [8] is used to calculate Fast Fourier Transforms, and the Eigen [9] library is used for basic math (mostly matrix multiplications). The models are loaded from a file and transformed into the frequency domain. This is done only once, at startup. Then for every input frame we detect pedestrians using this cached model. The per-frame computation can be divided into 4 main steps:

1) Feature computation;
2) Patchwork convolution;
3) Scoring;
4) Output.

In the rest of this section we will analyze all the different steps in detail.

A. Initialization

At start up, a function loads a model mixture from a file into memory. The models in the mixture are then transformed with a fast Fourier transform, and cached in memory. The models are trained using Felzenszwalb's code, release 4 [3].

A mixture is a group of models representing a specific category (in our case pedestrians), because a single model is usually not enough to capture the different ways an object can appear. For our tests we used mixtures of 6 models, which is a typical number for pedestrian recognition.

Lowering the number of models in the mixture alone decreases computational time, but it does not affect the time required to compute HOG features of the input images and transform them.

An alternative approach, suggested by [10] is to increase the number of models in order to have models at several resolutions but compute HOG features of the source image at fewer scales. This is more time efficient than calculating a higher density feature pyramid but using fewer models. Dollar et al. [11] propose a method to generate a dense feature pyramid in a fraction of the time, by interpolating HOG features computed at few different resolutions. The same approach could be applied to models, generating a higher number of interpolated models at different resolutions.

B. Feature Computation

The parameter **interval** defines the number of scales at each octave of our feature pyramid as shown in Fig. 1. The scale of the i-th level of the pyramid is given by:

$$scale(i) = 2^{1-(i/interval)}$$

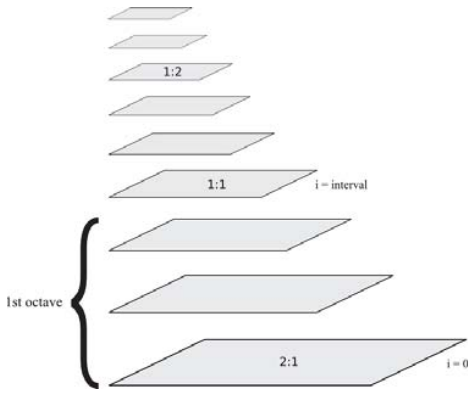

Fig. 1 A pyramid of features with **interval** set to 3. This means we will have three levels for each octave. Every octave is halved in size compared to the previous one

Because part filters are double the resolution of root filters, when we are calculating the score of the model at level i, we will compute the part scores at level $i - interval$. Therefore, the first octave of the pyramid (the one at the highest resolution) is only used for part scoring and not for root filters. We compute this first octave, at double resolution, not by doubling the image in size, but computing the HOG features in cells of 4x4 instead of 8x8. Other octaves are calculated

resizing the original image with a bilinear interpolation, and using 8x8 cells.

The total number of octaves depends on the dimension of the original image. We keep creating smaller levels as long as the image is at least 40 pixels wide (5 cells). For RGB images, the gradients are calculated for each of the three channels, and the one with the highest magnitude is used for the feature computation.

HOG features are normalized in 4x4 blocks with the same normalization described by Dalal-Triggs [12]. Additionally, in order to speed-up the feature computation, we use a lookup table as described in [7]. The lookup table is calculated the first time a feature pyramid is created, then stored in memory and reused for every feature computation.

C. Patchwork Convolution

The FFTW library is optimized to compute Fourier transforms of planes of a predefined size. Because of this, instead of transforming every scale of the feature pyramid on its own, we pack them into bigger planes of the same size, making the computation faster. Fitting the pyramid scales into a single patchwork layer is an optimization problem known in the literature as the $packing$ problem. In our case we use a simple Bottom-Left algorithm [13] (complexity of $\mathcal{O}(n^2)$). Each rectangle, starting from the biggest, is placed as close to the bottom as will fit, and as closed to the left as possible, without overlapping with previously placed rectangles. If not enough space is available, a new patchwork plane is created.

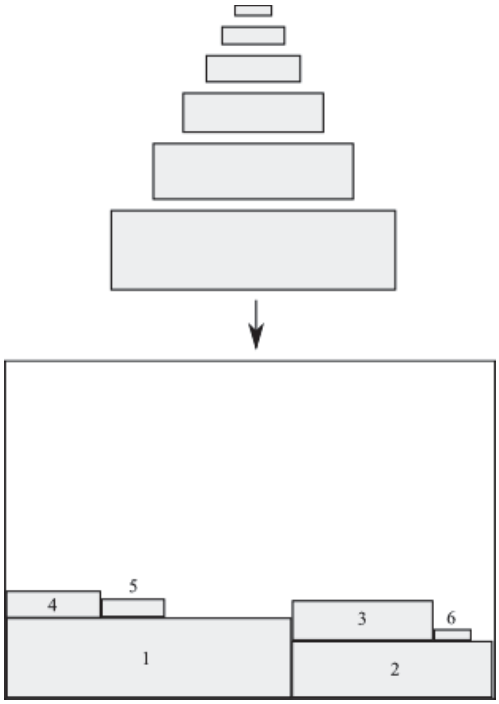

Fig. 2 Patchwork creation with a BLF algorithm. The pyramid layers are inserted from the biggest to the smallest, as low as it will fit and as close to left as possible

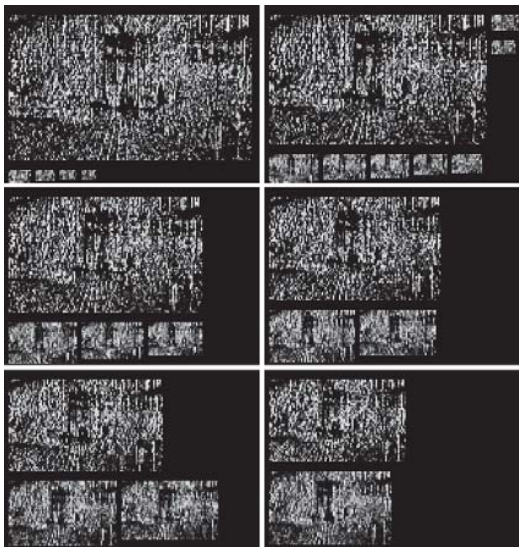

Fig. 3 An actual patchwork created by the program, ready for transformation. We create the minimum number of planes necessary to store all pyramid layers (in this case 6 planes were required). Plane size is fixed at initialization because the FFTW class is initialized and optimized to transform images of a specific size for maximum speed

D. Scoring

At this point, we take the matrices representing the transformed patchwork planes (p) and pointwise multiply them with each transformed root and part filter (f). The results of these Hadamard products is then transformed back into the spacial domain and stored in a $(p * f)$ vector named *convolutions*, which is used to calculate the score of each model. For each model, in the mixture we loop over all parts to calculate deformation costs. At the end of this phase we have two vectors: $Scores$ and $positions$. $Scores$ will be used to find out the highest scoring points of the original image, and, for each of these points, $positions$ will tell us the optimal part positions given a certain root position (x, y).

Fig. 4 shows the $scores$ vector at each scale of the pyramid. A maximum can be easily seen at many scales, in correspondence to the pedestrian in the source image. This goes to show that the part-based representation does not require an extremely dense feature pyramid. Even if the root filter is not exactly equal in size to the pedestrian, pedestrians will still score highly. But to obtain detection windows that are as precise as possible, it is good to use a high number of scales.

E. Output

Once the $scores$ and $positions$ vectors have been populated, we scan the $scores$ vector for each level of the pyramid, looking for detections above the user-defined threshold. Overlapping detections are removed (the detection with the lower score is removed). The overlap criteria is the same used for the PASCAL and KITTI challenges: Two boxes are considered overlapping if the area of the intersection over the area of the union is greater than 50% (or another user-defined threshold).

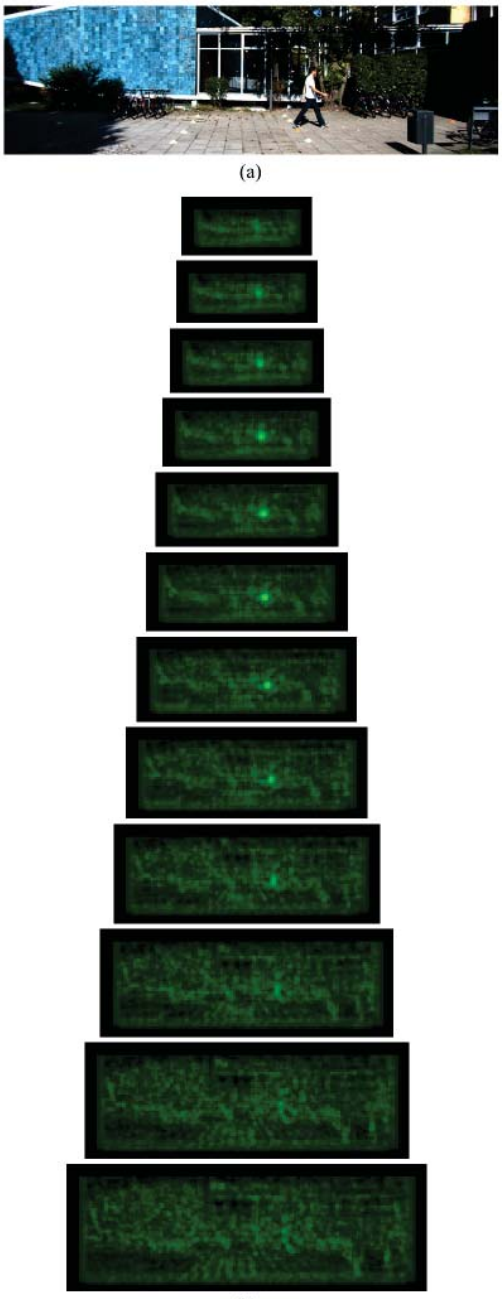

Fig. 4 (a) Response of the filter (root+parts) at various levels of the pyramid generated from (a). We can see that the size of the filter does not have to match precisely the size of the pedestrian to obtain a high filter response. The pedestrian represents a maximum at most scales of the pyramid

F. Search Ranges

We manage to further speed-up and improve the precision of our classifier by making some considerations regarding perspective. With the method explained so far, we are searching for pedestrians of all sizes in every possible position of the source image. If we know where the horizon of the image is, it is possible to discard many hypotheses based on their size. This not only allows us to speed-up computation, but also to decrease false positives.

Using perspective information as an additional constraint is useful for any multi-scale, sliding-window approach, and it can reduce execution time considerably, depending on how strict the constraint is.

Our objective is to avoid calculating the filter convolution in areas of the image where the filter size is not congruent with perspective information. In our case, different considerations must be made, as we calculate convolutions in the frequency domain, and we use different resolutions for the root and part features.

We use the details of the camera lens and its position to learn the perspective center, and use two parameters, W0 and W1, defining the minimum and maximum pedestrian width in meters. From these parameters, for each line of the source image, we can compute the minimum and maximum plausible width of a pedestrian laying on that specific line, as shown in Fig. 5.

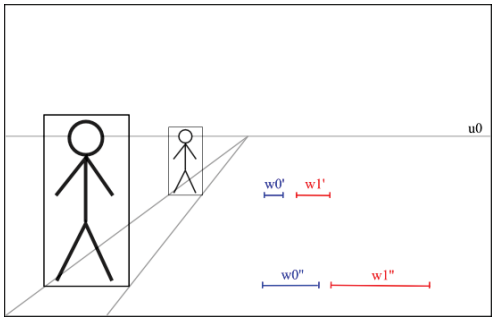

Fig. 5 Given the information about the horizon position (u0), for each scanline of the image we can estimate a minimum ($W0^1$) and maximum ($W1^1$) plausible width of a pedestrian laying on that line, based on absolute parameters W0 and W1 representing the minimum and maximum width of pedestrians in meters

In our case, the filter size is constant, but we operate on a pyramid of features at different scales, so higher resolution scales are used to detect smaller pedestrians, while lower resolution scales are used to detect pedestrians near the camera. With this information, for every level of the feature pyramid we compute features only in the area of the image where pedestrians of that specific size can be found.

Given a specific filter, we can find out its size in the image space by multiplying the filter width with the width of HOG cells (8 in our case). For every scale of the pyramid we find out which rows are useful according to parameters w0 and w1 (for example, the highest resolution features will only be needed near the horizon, to look for small pedestrians in the distance). Now for each layer of the pyramid we have two values *minRow* and *maxRow*, limiting our search at that level of the feature pyramid. It is important to remember that every layer of the pyramid is also used to search for parts when the root filter is placed an octave above, because part filters have double the resolution of root filters. So when using a root filter at layer i, we will scan for parts at layer $i - interval$.

The useful area at level i, $A(i)$ only represents the area for root filters placed at that level, but we will also need features at

that resolution to score the part filters at level $i+interval$. For this reason at level i, the area for which we compute features $F(i)$ is:

$$F(i) = A(i) \cup A(i + interval)$$

This is true for all layers except the first $interval$ layers, which are only used for part scoring, so the area needed is only $A(i + interval)$. Pruning detections that are not congruent with perspective information gives an increase in mean average precision of 5%. They also allow us to further reduce the total computational time to about $105ms$, which is an acceptable speed for a real time scenario.

IV. Results

We tested our algorithm on the KITTI dataset, which provides labeled images as well as c++ code to calculate precision/recall performance. Pedestrian from the KITTI dataset are divided into three categories:

- **Easy**: Min. bounding box height: 40 Px, Max. occlusion level: Fully visible, Max. truncation: 15%
- **Moderate**: Min. bounding box height: 25 Px, Max. occlusion level: Partly occluded, Max. truncation: 30%
- **Hard**: Min. bounding box height: 25 Px, Max. occlusion level: Difficult to see, Max. truncation: 50%

In Sections IV-A to IV-E we describe the tuning we applied in different areas in order to obtain the best results.

A. Scaling

While resizing the source image to calculate HOG features at different scales of the pyramid we noticed that using bilinear interpolation instead of a simple nearest neighbour approach gave us an increase in mAP from 23% to 27% for the moderate pedestrian class.

B. Training

Training a model on a different dataset can make a huge difference in detector performance and, as shown in Fig. 6, the best results are achieved when training a model on a dataset similar to the one used for testing.

Test set \ Training set	INRIA	Caltech-USA	KITTI
INRIA	*17.42 %*	60.50 %	**55.83 %**
Caltech-USA	**50.17 %**	*34.81 %*	61.19 %
KITTI	**38.61 %**	28.65 %	*44.42 %*
ETH	**56.27 %**	76.11%	61.19 %

Fig. 6 Effect of training set on the detection quality (lower is better for INRIA, Caltech-USA and ETH, higher is better for KITTI). Bold indicates the best result using a different training set

For our initial tests, we used a DPM mixture trained on the Pascal dataset, we then switched to training out models on the KITTI dataset, leading to a huge improvement in precision-recall rates. We train mixtures of six models, three of which are generated by the latent SVM training, the remaining three are generated by mirroring the first three with respect to the vertical axis.

Fig, 7 shows two mixtures, the one on the top row was trained on the KITTI dataset, while the one on the bottom row was trained on the Pascal 2007 dataset. Models trained on the KITTI dataset give much better results in our tests, because the images in the dataset are all captured from a car-mounted camera, in urban settings. The Pascal challenge includes images of humans in many different settings (on a beach, at a restaurant, on a motorcycle, and so on) so, the models trained on this dataset are more generic, but less performing for our use case.

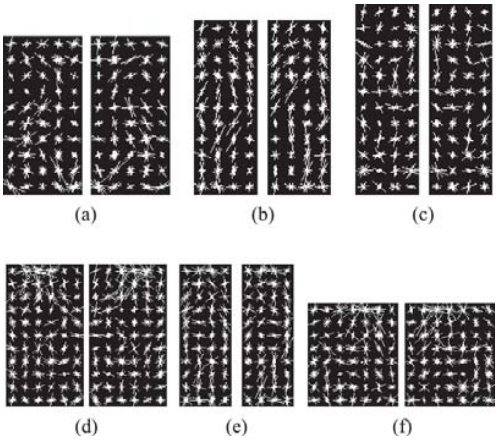

Fig. 7 A mixture trained on the KITTI dataset (first row) and one trained on the more generic PASCAL dataset (second row). Models (a) and (d) represent a pedestrian walking left and right. Models (b) and (e) represent a pedestrian walking towards the camera. Model (c) represents an upright pedestrian, while model (f) represents a person sitting at a table (where only the chest and head are showing). The mixture in the first row is much more effective in our case

Training a 6 models mixture gives us a pretty generic model, and there is no sign of overfitting. In fact, even if we train the mixture on the same images used for testing, there is no improvement in precision-recall. We also tried training a 12-component mixture, without any improvement in results, on the contrary, we experienced a small increase in false positives. By training a model on the KITTI dataset (different images than the ones used for testing) we obtain a big increase in performance. Mean average precision for the $moderate$ class goes from 27% to 46,89%. Fig. 8 shows the difference in precision-recall for $moderate$ pedestrians using a model trained on Pascal versus one trained on KITTI.

C. Optimal Search Ranges

Search ranges can greatly speed-up computation and prune false positive detections, but it is important to choose them carefully to avoid pruning useful detections.

The best analytical approach to choose the minimum and maximum pedestrian width W0 and W1, is to take the ground truth labels and analyze their distribution.

For all the labeled images in the dataset we stored, for each row of the source image, the width of pedestrians that can be

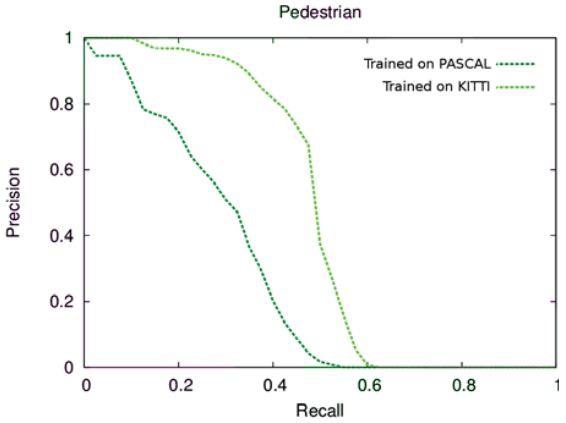

Fig. 8 Improvement in precision-recall for the moderate class using a model trained on KITTI. Light green curve is trained on KITTI, dark green is trained on Pascal

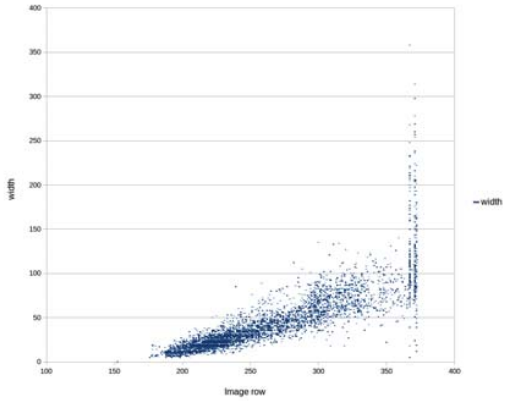

Fig. 9 Scatter plot of pedestrian width versus the row of the source image where they lay

found at that row. Fig. 9 shows the results of this analysis as a scatter plot.

We can clearly see from the plot that that no pedestrians are positioned above the horizon (rows 0-150), and for each row of the source image we know what size of pedestrians to expect. Detections outside this range are likely false positives.

From the statistics of pedestrian width at different distances extrapolated from labelled KITTI data, we can safely choose the search ranges parameters W0 and W1. The exact values maximizing average precision are $W0 = 0.346m$ and $W1 = 1.39m$.

D. Detecting Small Pedestrians

As we can see from Fig. 10, the majority of pedestrians that we miss are the small ones, placed near the horizon. This is not surprising given that the minimum width of our root filters is 4 HOG cells, corresponding to $4 \times 8 = 32$ pixels (see Fig. 7). Most of the pedestrians that we do not detect are below 35 pixels in width, as shown from Fig. 10. In order to allow us to detect these smaller instances we introduce a new parameter called $scale$.

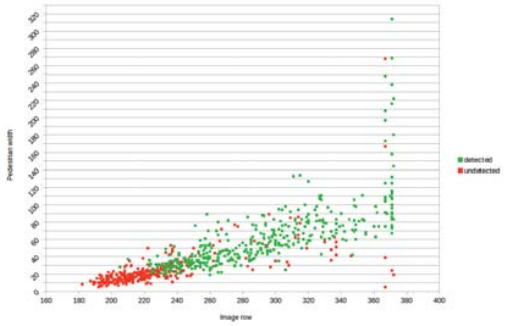

Fig. 10 Undetected (in red) and detected (in green) pedestrian instances, for each image row and pedestrian width

Before creating the HOG pyramid, the source image is scaled based on the $scale$ parameter (using bilinear interpolation). If we use a a scale parameter of 2.0 our 32 pixels filter will effectively work as a 16 pixels filter.

Using a scale parameter of 1.5 brings a good improvement in average precision. Fig. 11 shows our precision-recall results using a scale of 1.5, for the $Moderate$ class we obtain an increased mean average precision of 0.52% versus 0.478%.

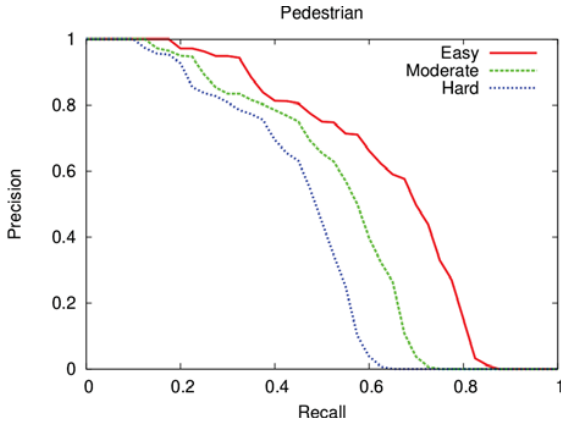

Fig. 11 Precision-recall using scale=1.5, 10 intervals, search ranges disabled

This increase in precision comes at a cost in computational time, but the increase is not linear, because by using search ranges the highest resolution image is only used to search for pedestrians on a small strip near the horizon.

E. Overlap

As described in Section III-E, we use an $overlap$ parameter for the final non-maximum suppression step.

To understand which $overlap$ value gives the best results, we calculated mean average precision over overlap values from 0.1 to 0.9, the results are shown in Fig. 12

A value of 0.4 gives the best results for the $easy$ and $moderate$ class. For the $hard$ class, while 0.4 is a local maximum, slightly better results are obtained using higher overlap values. This could be explained by the fact that the $hard$ class is constituted by smaller and sometimes overlapping pedestrians, therefore, using a smaller $overlap$

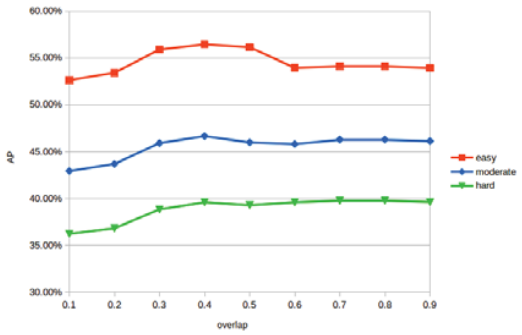

Fig. 12 Change in mean average precision for the $easy$, $moderate$ and $hard$ class, when changing the $overlap$ parameter. (tests were done with $interval = 2$ and search ranges disabled)

value would lead to the elimination of some detections that were correct, but overlapping (in that case the less visible detection with the smaller score is eliminated).

A lower overlap value decreases the number of false positives, leading to better mean average precision. But, as shown in figure, if we want high recall and do not care too much about precision, then using a higher overlap value can be a better choice (increasing value over 0.6 does not lead to better results).

F. Comparison with the State of the Art

We obtained our best results using 10 pyramid intervals, a scale of 1.5 and search ranges. Enabling search ranges gives us an additional improvement in precision-recall, allowing us to obtain an average precision of 54.79% on the $moderate$ class. As we can see from Table II, our method outperforms all other DPM-based methods on the KITTI dataset, even those exploiting additional information, such as fusion-DPM, which uses dense point clouds generated by a lidar sensor, or DPM-C8B1, which uses stereo information.

TABLE II
MEAN AVERAGE PRECISION ON THE KITTI DATASET FOR PEDESTRIAN CLASS OF METHODS BASED ON DPM

Method	Moderate	Easy	Hard
Ours	**54.79 %**	**66.76 %**	**46.55 %**
Fusion-DPM	46.67 %	59.51 %	42.05 %
DA-DPM	45.51 %	56.36 %	41.08 %
LSVM-MDPM-sv	39.36 %	47.74 %	35.95 %
LSVM-MDPM-us	38.35 %	45.50 %	34.78 %
DPM-C8B1	29.03 %	38.96 %	25.61 %

Fusion-DPM [14] uses lidar data in addition to RGB information. DA-DPM [15] uses Domain Adaptation. DPM-C8B1 [16] uses stereo information.

G. Speed

Computing convolutions in the frequency domain as in [6] makes our algorithm faster than most state-of-the-art DPM implementations, without any loss in precision. Using a 10 interval pyramid and no search ranges, the average computational time for a frame in the KITTI dataset is $520 \pm 26ms$. This is good for a DPM detector, but not enough for our use case. By allowing for a small loss in precision we can further speed-up our algorithm. The norm for challenges that only require high precision, but have no speed requirements, is to construct a feature pyramid with 10 intervals. The problem is that this requires calculating the HOG features many times, and in our case also doing many Fourier transforms. As explored in [10], for real time detection, it is more efficient to use fewer scales of the input image, and use a greater number of filters representing multiple scales. This reduces the computational time required for feature extraction.

Increasing the number of intervals allows us to produce more precise bounding boxes which is desirable to score highly on challenges, but not necessary for all applications.

As shown in Fig. 13, there are diminishing returns in terms of average precision when using additional pyramid levels, while time increases linearly.

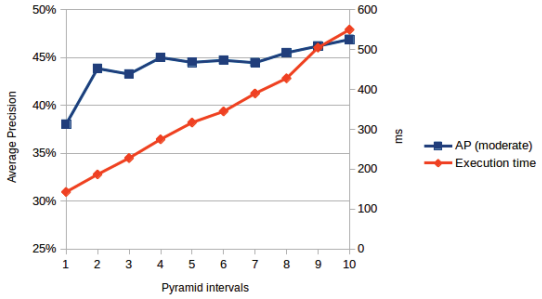

Fig. 13 Change in average precision and computational time when increasing the number of pyramid intervals

Lowering the number of pyramid intervals does not lower average precision too much as long as we use more than 1 interval. The decrease in precision from using 2 intervals instead of 10 is about 3%, while computational time is more than halved (we go from $550ms$ to $180ms$), the precision-recall curves are very similar.

Even with pedestrians of many different scales, an interval value of 2 is usually enough to detect all pedestrians in the image, as shown in Fig. 14.

The introduction of search ranges allows us to further speed-up the algorithm. For our test data we found $W0 = 0.346$ and $W1 = 1.39$ to be the optimal parameters maximizing average precision. There is a tradeoff between speed and average precision when choosing these parameters. For example, if we increase W0 execution time decreases linearly, but precision will decrease as well, as we are going to miss some of the smallest pedestrians. If we are not interested in detecting small pedestrians far away and prefer a faster detection this kind of tradeoff can be taken into consideration, depending on the application. For our results, we used the optimal W0 and W1 parameters, as the increase in speed by using stricter limits did not seem enough to justify the loss in precision.

H. Speed Results

Calculating convolutions in the frequency domain allows us to decrease computational time from about 3 seconds to

Fig. 14 Detections using interval=1,2,10. Increasing interval from 2 to 10 does not lead to any additional detection

$550ms$, with no loss in average precision. Decreasing the pyramid intervals from 10 to 2 allows us to bring execution time down to $180ms$ per image, with a loss in average precision of about 2%. Enabling search ranges allows us to further reduce execution time to $105ms$, with no decrease in average precision. By using different $W0$ and $W1$ parameters we can bring execution time even further down, but at a loss of average precision. Fig. 15 shows our precision-recall results using this $fast$ configuration. We achieve an average precision of 49.6%, which is still better than all DPM-based solutions published on the KITTI benchmark website.

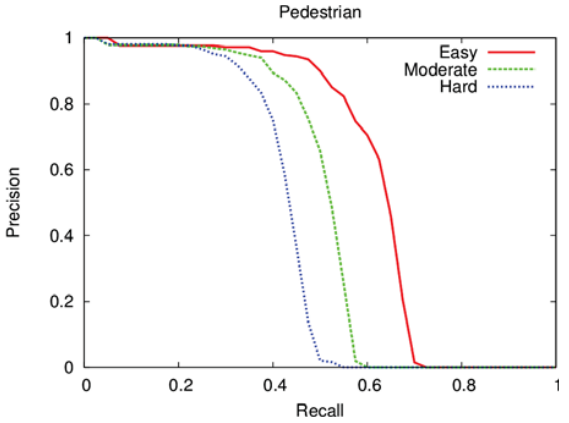

Fig. 15 Precision-recall results using our $fast$ configuration (runtime = $0.1s$)

Our execution time is divided as shown in Table III.

TABLE III
TIME SPENT IN EACH STEP OF THE DETECTION PROCESS

Task	Time
HOG feature computation	51ms
Patchwork creation and transformation	11ms
Convolution and distance transform	43ms

About 25% of the time is spent in the function calculating the matrix pointwise product between the transformed filters and the transformed patchwork planes. It could be possible to speed-up this part of the process by calculating products on the GPU. In Table IV, we compare our timing results with alternative DPM implementations tested by us and results of DPM based methods published on KITTI's evaluation website.

TABLE IV
EXECUTION TIME OF DIFFERENT DPM BASED METHODS FOR ONE IMAGE OF THE KITTI DATASET (*OBTAINED FROM [17])

Method	Time	Environment
Fusion DPM	* 30s	1 core @ 3.5 Ghz
DA-DPM	* 21s	1 core @ 3.5 Ghz
LSVM-DPM	* 10s	4 cores @ 3.0 Ghz
OpenCV vanilla	4.38s	4 cores @ 2.4 Ghz
OpenCV cascade	1.00s	4 cores @ 2.4 Ghz
FTVQ	1.24s	4 cores @ 2.4 Ghz
FFT	0.55s	4 cores @ 2.4 Ghz
Ours (best settings)	**0.41s**	**4 cores @ 2.4 Ghz**
Ours (fast settings)	**0.10s**	**4 cores @ 2.4 Ghz**

Our implementation is faster than all the alternative DPM-based methods that we tested on the KITTI dataset. Images from the KITTI dataset are 1242×375, for smaller images (e.g. 640×480) our method can perform at 20 or more frames per second. This results are compatible with a real-time usage scenario, which is our main concern. It may be possible to further speed-up our method by computing matrix multiplications on a GPU, or using a cascade method to avoid computing all the distance transforms for parts in unpromising areas.

V. CONCLUSIONS

We implemented a fast deformable part model detector based on the work by Dubout et al. [6], which speeds up the process by computing convolutions in the frequency domain. We managed to increase average precision by pruning false positives based on perspective information ($+2.8\%$), and supersampling images using bilinear interpolation ($+4\%$). We reach a mean average precision of 55% for the $moderate$ pedestrian class on the KITTI dataset, outperforming all DPM-based methods, including those using extra information such as lidar point clouds.

By lowering the number of pyramid intervals and exploiting prospective information, we can bring down computational time for an image in the KITTI dataset to about 100 milliseconds. Our implementation is suitable for real-time applications and it is faster than all publicly available DPM implementations.

Most of our missed detections are occluded or very small pedestrians. For future work, finding a method to model occlusions could decrease miss rate. In order to increase execution speed pointwise matrix multiplications could be computed on a GPU, and a cascade approach may be implemented to avoid scoring part deformation costs for all points of the image.

REFERENCES

[1] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 9, pp. 1627–1645, 2010.

[2] P. F. Felzenszwalb, R. B. Girshick, and D. McAllester, "Cascade object detection with deformable part models," in *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*. IEEE, 2010, pp. 2241–2248.

[3] ——, "Discriminatively trained deformable part models, release 4," http://people.cs.uchicago.edu/ pff/latent-release4/, (Accessed: 2015-09-30).

[4] M. A. Sadeghi and D. Forsyth, "Fast template evaluation with vector quantization," in *Advances in Neural Information Processing Systems*, 2013, pp. 2949–2957.

[5] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 1, pp. 117–128, 2011.

[6] C. Dubout and F. Fleuret, "Exact acceleration of linear object detectors," in *Computer Vision–ECCV 2012*. Springer, 2012, pp. 301–311.

[7] J. Yan, Z. Lei, L. Wen, and S. Z. Li, "The fastest deformable part model for object detection."

[8] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[9] G. G. Benot Jacob, "Eigen library v3.1," http://eigen.tuxfamily.org, 2014, (Accessed: 2015-09-30).

[10] M. Sadeghi and D. Forsyth, "30hz object detection with dpm v5," in *Computer Vision ECCV 2014*, ser. Lecture Notes in Computer Science, 2014, vol. 8689, pp. 65–79.

[11] P. Dollár, R. Appel, S. Belongie, and P. Perona, "Fast feature pyramids for object detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, no. 8, pp. 1532–1545, 2014.

[12] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[13] S. Imahori, M. Yagiura, and H. Nagamochi, "Practical algorithms for two-dimensional packing," *Handbook of Approximation Algorithms and Metaheuristics. Chapman & Hall/CRC Computer & Information Science Series*, vol. 13, 2007.

[14] C. Premebida, J. Carreira, J. Batista, and U. Nunes, "Pedestrian detection combining rgb and dense lidar data," in *IROS*, 2014.

[15] J. Xu, S. Ramos, D. Vzquez, and A. M. Lpez, "Hierarchical Adaptive Structural SVM for Domain Adaptation," in *arXiv:1408.5400*, 2014.

[16] J. Yebes, L. M. Bergasa, R. Arroyo, and A. Lzaro, "Supervised learning and evaluation of KITTI's cars detector with DPM," in *IV*, Detroit, USA, June 2014, pp. 768–773.

[17] A. Geiger, "Kitti object detection evaluation," http://www.cvlibs.net/datasets/kitti/eval_object.php, 2014, (Accessed: 2015-09-30).