Deformable Part Based Models for Pedestrian Detection

Vivardhan Kanoria
Stanford University
Management Science & Engg. Department
vkanoria@stanford.edu

Kevin Truong
Stanford University
Computer Science Department
ktruo001@stanford.edu

Abstract

We implement a modified version of a pedestrian detector using the Deformable Part Based Model approach. This paper talks about the general problem, approach, code and results. Our trained detector was tested on the PASCAL data set with various iterations, and we found that having multiple components was the most crucial aspect to the algorithm.

1. Introduction

We implement an algorithm for human detection, using a deformable parts based model, based on the work of Felzenszwalb et al. The implementation includes codes for training the detector given positive images annotated with bounding boxes for humans and negative images that do not contain any pedestrians. We modify the approach taken by Felzenszwalb et al in some ways.

2. Pedestrian Detection

This project considers the problem of detecting and localizing people in static images, based on the work of Felzenszwalb, et al (referred to as the authors in this paper) (enter citation). Pedestrian detection is a difficult problem because people can vary greatly in appearance. Variations arise not only from changes in illumination and viewpoint, but also due to non-rigidity of the human body and variations in shape, color and other visual properties. For example, people wear different clothes and take a variety of poses. The deformable parts model is an object detection system that represents highly variable objects using mixtures of multiscale deformable part models. These models are trained using a discriminative procedure. The training set consists of bounding boxes for people in a set of images. The system presented by the authors is both efficient and accurate and achieves state-of-the-art results on the PASCAL VOC benchmarks [5] and the INRIA Person data set [4].

The authors achieved an important milestone in computer vision by showing the effectiveness of part based models which had traditionally been outperformed by simpler models. This was because rich models often suffered from difficulties in training due to use of latent information. For example, while training a part-based model from images labeled with bounding boxes, the part locations are not labeled and they must be treated as latent (hidden) variables.

2.1. Model Formulation

A deformable part based model is trained on HOG feature pyramids as described by Dalal, et al. [4]. The model is defined by a root filter that approximately covers an entire object and part filters that cover smaller parts of the object. The root filter location defines a detection window (the pixels contributing to the region of the feature map covered by the filter). The part filters are placed at a pyramid level that has double the resolution of the root, so the features at that level are computed at twice the resolution of the root features. Higher resolution features for defining part filters helps to capture finer features that are localized to greater accuracy when compared to the features captured by the root filter.

A model for an object with n parts is formally defined by an (n+2)-tuple $\{F_0, P_1 \ldots P_n, b\}$ where F_0 is a root filter, P_i is a model for the i^{th} part, and b is a real-valued bias term. Each part model P_i is defined by a 3-tuple $P_i = \{F_i, v_i, d_i\}$, where F_i is a filter for the i^{th} part, v_i is a two-dimensional vector specifying an "anchor" position for part i relative to the root position, and d_i is a four-dimensional vector specifying coefficients of a quadratic function defining a deformation cost for each possible placement of the part relative to the anchor position.

An object hypothesis specifies the location of each filter in the model in a feature pyramid, $z = \{p_0 \ldots p_n\}$, where $p_i = \{x_i, y_i, l_i\}$ specifies the level and position of the ith filter. The score of an object hypothesis z is given by:

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

where:

$$(dx_i, dy_i) = (x_i, y_i) - (2(x_0, y_0) + v_i) ,$$

gives the placement of the part relative to the anchor position v_i and the root position (x_0, y_0)

3. Training Pipeline

At a high level our method is similar to Felzenszwalb et al. We discuss the specific details of the method here, highlighting differences in our approach, if any, in section 5.

3.1. Train Root Filter

The root filter is trained on the HOG features from positive bounding boxes and negative images in the training set. The problem formulation is that of a simple SVM, where the features are concatenated sets of feature vectors from the training images. This approach has also been used by others before Felzenszwalb et al and is well documented in literature.

3.2. Initialize Part Filters

Given the trained root filter, we now initialize the part filters. There are several important rules that are used to do this. These are listed below:

a. Bounds on width and height: We lower bound the width and height by 3 units of HOG feature bins. The upper bound is 2 less than twice the corresponding root feature dimension
b. Aspect Ratio: We constrain the absolute difference between the width and height to be less than 5 HOG bins
c. Root filter area overlapped: Each part filter should overlap at least the root area divided by the number of parts and at most 120% of the root area divided by the number of parts
d. Part Mutual Exclusion: We constrain all parts to be non-overlapping with each other when initialized
e. Symmetry: A part is either placed in the center (its central vertical axis coincides with the root filter's central vertical axis) or it has a symmetric counterpart with respect to the root filters vertical axis of symmetry
f. Scale of Computation: The part filters are initialized at twice the spatial resolution of the root
g. Positive Root Weights: We only use positive root weights to initialize the parts

3.3. Retrain Filter Update with Latent Information

The authors allow for three different pieces of latent information: the model component, the location of the root filter and the location of the parts relative to their anchor position in the root. Our model formulation uses a latent SVM model with the objective function:

$$f_\beta(x) = \max_z \beta.\Phi(x, z)$$

Our latent variables are our model components, root filter position and part locations. Our feature weights are the weights corresponding to the root filter, part filter, and deformation cost. Training the latent SVM formulation requires us to approach the problem in two steps: first, we optimize our latent variables, and then we optimize the weights. We repeat this cycle iteratively.

Given a model, we determine these latent positions by finding the positions that give the max score of our SVM hypothesis. Finding the max of the component is relatively straight forward, since we are just taking the max score of all components. Finding the max of the root and part positions, however, is more complicated because we have to find the max of them both simultaneously. This involves searching through every possible part positions given a root filter position, which can be computationally intensive. In section 4, where we talk about inference, we will explain more about this issue and the strategies we used to make our algorithm more efficient.

Once we determine the best root position of an example, we can update the root filter bounding box to the bounding box that we have found and train on this new bounding box. This corrects for annotations that are inaccurate, and it also normalizes the size of the annotation so we no longer need to warp the positive images. This process is called the root filter update.

3.4. Datamining hard negative examples

The training data is skewed in that for every positive example, there are many thousand negative examples in an image. This skew creates bias in the classifier if trained directly, so instead of giving our classifier every negative example, we want to intelligently select useful negatives in a size-constrained cache. This size constraint not only guarantees an upper limit to the skew, but also helps computational efforts since our memory utilization will be far less.

Initially, we do not have a classifier, and we select the negative examples randomly. After having trained this classifier from the annotated positive examples and randomly selected negative examples, we have a working classifier that helps us decide the hard negative examples that will give the most useful information. The criterion for a hard negative example is defined by:

$$H(\beta, D) = \{\left(i, \Phi(x_i, z_i)\right) \mid z_i = \operatorname{argmax}_{z \in Z(x_i)} \beta.\Phi(x_i, z_i) \text{ and } y_i\left(\beta.\Phi(x_i, z_i)\right) < 1\}$$

This is simply determining the highest scoring negative examples that are inside the margin of our SVM classifier. On the other hand, the easy negative examples are defined by:

$$E(\beta, F) = \{(i, v) \in F \mid = y_i(\beta.v) > 1\}$$

We remove all the easy examples to save space and ensure that our classifier only learns from the most useful negative examples. Examples that are neither hard nor easy are ambiguous examples, and we keep them on our cache. They are vectors that are right on our classifier's margins, i.e. they are our support vectors.

Each loop for datamining hard negatives is as follows:

1. Train classifier with current positive and negative examples.
2. With the newly trained classifier, re-score all the negative examples to find the easy negatives. Remove them from the negative cache.

3. Re-score all the negative examples to find the hard negatives. Randomly add them to the cache until size limit has reached.
4. With a new set of negative examples, we go back to step 1 and retrain the classifier on a smarter dataset.

4. Inference

Inference on a test image is fairly similar to the training procedure. We get the HOG feature pyramid and convolve each level with the root filter and corresponding levels with the part filters. The best overall score for every root position is computed using the distance transform technique for each part. The scores thus obtained are then thresholded to get detection instances. Finally, non-maximal suppression is performed.

Searching the image for the latent parameters is a fairly expensive computation owing to the large space to be searched. The authors used an approach called min-convolutions or distance transforms proposed by Felzenszwalb, et al. [3] in order to speed up this search. We attempted to do this computation in a reasonable amount of time by storing feature vectors for images, computing convolutions of filters with feature pyramids at most once and taking advantage of fast matrix and tensor algebra in MATLAB.

However, after timing our runs using this brute force method, we also implemented our own version of the distance transform. The min-convolution or distance transform is an elegant approach for finding the extrema of sampled function under deformation costs given by a specified deformation function. The method relies on formulating the overall score as the maximum envelope specified by a set of paraboloids, each starting at the original function value and decreasing with distance from the canonical position as determined by the deformation parameters.

5. Beyond the Standard Implementation

We have implemented parts of the project that are beyond the minimal expectations. Most of these are just performance optimization to improve the training and detection time, as well as allowing us the power to use the multi-component model.

5.1. Feature Cache

During training, our training examples are referenced by the image path and bounding box. Throughout the training, we have to load the image from the image path and compute feature pyramid multiple times per iteration. To speed things up, we cached all the feature pyramids onto disk for every training instance so that we only have to load the features to obtain them. Loading the image and getting features takes ~0.3 seconds on our environment, while loading the feature pyramid directly from disk takes 0.03 seconds. This was a huge time saving given that we have thousands of training examples that we need features from, twice for every one of about a dozen iterations. Although our disk space is more strained (15+ GB used), disk space is cheap relative to CPU performance, so we believe this tradeoff is optimal for our set up.

5.2. Dynamic programming/Distance transforms

The distance transforms method although not required for this project was crucial in speeding up latent position computation. It is particularly fast because of dynamic programming. We can perform two independent one dimensional computations in the x and y directions. The algorithm then runs in time that is linear in each of these dimensions.

The results were encouraging, as we got about a 25% improvement in running time per image for finding its latent values over the brute force approach. However, the real speed up was achieved when writing a MEX file to do the computation. This was the final version used and allowed to us calculate a two dimensional distance transform for a ~100x100 convolution matrix in about 0.05 seconds.

The method implemented by us differs slightly from the one used by the authors. The authors explicitly mention in their code that their implementation runs in time that is O(nlogn), where n x n is the approximate size of the convolved score matrix. However, we apply the method stated in [3], which has a theoretical running time that is O(n).

5.3. Multithreading

We have written a function, detectParallel.m that detects batches of images in a parallel loop. On our machines with 4 cores, the time spent detecting the entire data set increased by roughly a factor of 2-3.

We have also allowed multithreading for our pascal_test.m function, which speeds up our evaluation.

5.4. Mixture model

Our code allows the flexible use of a multi-component model. We have reused the author's code mergemodels.m, which merges multiple models into one multi-component model, and split.m, which splits the training data according to aspect ratio to feed into the respective components. We will analyze the usefulness of multi-component models in section 7.

6. Code

Our codes were implemented with the following structure. There were 4 main training subroutines:

a. **train.m –** the main training function
b. **initParts.m** – function that initializes the part filters and deformation weights given the root filter
c. **detect.m –** function that finds the latent values and corresponding overall scores, given a training or test image
d. **updateNegCache.m –** function that updates our negative cache by removing all the easy examples and randomly adding hard ones

6.1. Training: train.m

We serialize the features and save them to disk. We then prepare the SVM header files, parameter lower bounds, the model information file and the model parameters file for feeding to learn.cc to learn the SVM model. We have train.m and train_old.m; train_old.m is the starter code that learns using poswarp and randneg. We only use these functions at first, and then use latent root position update and the hard negative cache afterwards. Train.m is configured to do latent root positioning while serializing the feature vector, and is expected to take in a hard neg cache updated from updateNegCache.m.

6.2. Part Initialization: initParts.m

We achieve a part filter initialization that obeys the rules mentioned earlier as follows. We first interpolate the root filter to twice its size by bicubic interpolation. We then reduce this double sized filter by changing all the negative weights to 0. We then compute the l2 norm of each 31 dimensional vector, flip the resulting matrix left to right and add it to itself to get a vertically symmetric "energy matrix".

Given this energy matrix, we now need to find the part filters and their canonical positions relative to the root filter. We initialize a predetermined number of parts (we have used 6 parts based on the authors' recommendations). We continue to add parts until this number has been added.

First, we find all valid shapes of the parts by looping over all possibilities of width and height combinations that satisfy the conditions mentioned earlier. Now for each of these shapes, we convolve an averaging filter of the same dimension with the energy matrix.

Now, we ensure that the parts we get are either non-overlapping and symmetrically placed or centered. This is done by appropriately restricting the search space of the convolved score matrix.

We find the best location in the convolution score matrix and set that to be the new part's anchor position. The weights of the part are the root filter weights that correspond to the coordinates of the anchor position. If the part is not centered, we also initialize its symmetric counterpart. The energy matrix values for all parts that are initialized are set to 0 before initializing more parts.

Finally if at some stage we are just left with one part to initialize, we enforce a constraint that the part to be added has to be centered. This ensures that we do not initialize more parts than the pre-determined number.

6.3. Inference: detect.m

This is the file where we loop through all root filter window positions and compute the overall score for each of them by finding the best scoring latent positions. We first pad the feature pyramid at each level to ensure that features are at least as large as the root filter. We then find the score of the root filter by convolving the root filter with each root level.

We then compute the corresponding anchor position of each part and convolve the part filter level with each part filter. The function that computes the distance transform for latent position and score calculation is called TwoDDistTransform.m. This file uses a mex file routine called OneDDT.cc to compute one dimensional distance transforms in each direction and combines the results to produce the final distance transform. This is used to update the score obtained by the root filter.

Finally we threshold the score for every root filter position and find all the scores that are greater than the threshold. The same file is used during training, except that we now pick only the best detection for training.

6.4. Get Hard Negatives: updateNegCache.m

We start with an empty cache. We add in all the examples of our old cache that are currently ambiguous (right at the margin). We then compute the score of the highest example window in each negative image. If the score is greater than the specified threshold, it is a hard negative, so we randomly add them in until the size limit is reached.

7. Results

All our experiments have been run on a training set that was randomly selected by selecting 1000 of the full training and test set. We ran four different variations: 1) our baseline result of just using the root filter + hard negative mining and latent detection, 2) multiple training iterations of our baseline (we chose 2 iterations), 3) baseline with 6 deformable parts, and 4) a multi-component (2 in our case) version of our baseline. Thus we can study the effects of the variations of the classifier.

For our baseline, we ran the initially trained root filter on our test set and got an average precision of 0.001. Our AP score for our multi-iteration model increased to 0.00138. Our multi-component model was our best AP of 0.003. Finally, we trained a one component model with parts and obtained an average precision of 0.0012.

This indicates that having multiple components is the most crucial aspect. This is clearly illustrated in the figure where the annotation bounding box has a very different aspect ratio from the single component root filter aspect ratio.

Adding parts gives a slight improvement in average precision. However, it seems from out results that splitting into components is more important than having more parts.

Although the average precision seems low, we believe it is due to the random selection of the 1000 training/test examples. When we ran our classifier on the full training set, we got an AP of 0.12, which is the expected baseline score. Furthermore, when we ran our classifier on the 400 small dataset, using just the root filter we have an AP of 0.07, and using 2 component root filter we have an AP of 0.13. Thus, we believe that due to random selection we have chosen difficult images that needed more than 1000 training data to be accurately discriminated, and we should not look at the absolute value of our AP scores,

but the relative value in how adding the various components to the simple root filter classifier helps our task.

Figure 1: visualization of a 2-component model. Left – component 1, with a lower aspect ratio. The model seems to capture people that are standing upright. Right – component 2 with a higher aspect ratio. This model seems to capture close up of faces. Together, they form a mixture model that outperforms parts.

Figure 2: visualization of our multi-iteration model. Left – one iteration. It seems to capture the general outline of a person, with their head, shoulder, and arms. Note, this is also the baseline of our comparison. Right – two iterations. The weights captured seems slightly sharper.

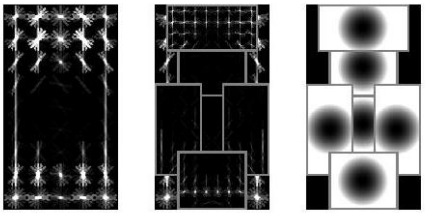

Figure 3: visualization of our parts model. Left – root filter, which seems to capture the general outline of a person's shoulders and arms hanging vertically on the side. Middle – part filters. Notice the head part filter has the most distinguished structure, indicating that the head is the most powerful indicator in our task. The other part weights are small in comparison. Right – deformation weights. They seem relatively circular; however, the head is slightly elliptical being more wide than tall. This indicates that the head can have a more relaxed variation along the x axis compared to the y axis, which makes sense since pedestrians can move left or right, but cannot walk up the air or down the ground.

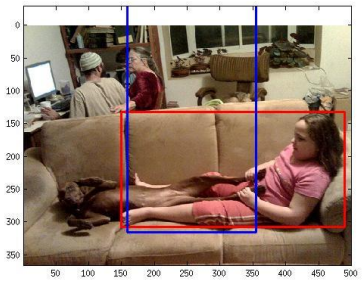

Figure 4: The importance of having multiple components. The annotated bounding box (red) can have any shape, and can not all be capture by one root filter (blue). This visualization is our best root filter overlap for the annotated bounding box found in the latent root filter update step. Although this was the best root filter position, it is a poor choice (and thus a low overlap, < 0.7) due to the difference in shape. However, if we have 2 components, both components together would have a better model of all the different variations of human poses, thus making a more generalized classifier.

8. References

[1] P. Felzenszwalb, D. McAllester, and D. Ramanan, A Discriminatively Trained, Multiscale, Deformable Part Model, Proc. IEEE, Conf. Computer Vision and Pattern Recognition, 2008.

[2] P. Felzenszwalb, R. Girshick, D. McAllester, and Deva Ramanan, Object Detection with Discriminatively Trained Part-Based Models, IEEE Transactions On Pattern Analysis And Machine Intelligence, 32:1627-1645, 2010.

[3] P. Felzenszwalb, and D. Huttenlocher, Distance Transforms of Sampled Functions, Technical Report 2004-1963, Cornell Univ. CIS, 2004N.

[4] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection, Proc. IEEE Conf. Computer Vision and Pattern Recognition, 2005.

[5] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman, The PASCAL Visual Object Classes Challenge 2007 (VOC 2007) Results, http://www.pascal-network.org/challenges/VOC/voc2007/2007.