CHAPTER 17

Detecting Objects in Images

Chapter 16 described methods to classify images. When we assumed that the image contained a single, dominating object, these methods were capable of identifying that object. In this chapter, we describe methods that can detect objects. These methods all follow a surprisingly simple recipe—essentially, apply a classifier to subwindows of the image—which we describe with examples in Section 17.1. We then describe a more complex version of this recipe that applies to objects that can deform, or that have complex appearance (Section 17.2). Finally, we sketch the state of the art of object detection, giving pointers to available software and data (Section 17.3).

17.1 THE SLIDING WINDOW METHOD

Assume we are dealing with objects that have a relatively well-behaved appearance, and do not deform much. Then we can detect them with a very simple recipe. We build a dataset of labeled image windows of fixed size (say, $n \times m$). The examples labeled positive should contain large, centered instances of the object, and those labeled negative should not. We then train a classifier to tell these windows apart. We now pass every $n \times m$ window in the image to the classifier. Windows that the classifier labels positive contain the object, and those labeled negative do not. This is a search over location, which we could represent with the top left-hand corner of the window.

There are two subtleties to be careful about when applying this recipe. First, not all instances of an object will be the same size in the image. This means we need to search over scale as well. The easy way to do this is to prepare a Gaussian pyramid of the image (Section 4.7), and then search $n \times m$ windows in each layer of the pyramid. Searching an image whose edge lengths have been scaled by s for $n \times m$ windows is rather like searching the original image for $(sn) \times (sm)$ windows (the differences are in resolution, in ease of training, and in computation time).

The second subtlety is that some image windows overlap quite strongly. Each of a set of overlapping windows could contain all (or a substantial fraction of) the object. This means that each might be labeled positive by the classifier, meaning we would count the same object multiple times. This effect cannot be cured by passing to a bigger training set and producing a classifier that is so tightly tuned that it responds only when the object is exactly centered in the window. This is because it is hard to produce tightly tuned classifiers, and because we will never be able to place a window exactly around an object, so that a tightly tuned classifier will tend to behave badly. The usual strategy for managing this problem is *non-maximum suppression.* In this strategy, windows with a local maximum of the classifier response suppress nearby windows. We summarize the whole approach in Algorithm 17.1.

```
Train a classifier on $n \times m$ image windows. Positive examples contain
the object and negative examples do not.
Choose a threshold $t$ and steps $\Delta x$ and $\Delta y$ in the $x$ and $y$ directions

Construct an image pyramid.

For each level of the pyramid
  Apply the classifier to each $n \times m$ window, stepping by
  $\Delta x$ and $\Delta y$, in this level to get a response strength $c$.
  If $c > t$
    Insert a pointer to the window into a ranked list $\mathcal{L}$, ranked by $c$.

For each window $\mathcal{W}$ in $\mathcal{L}$, starting with the strongest response
  Remove all windows $\mathcal{U} \neq \mathcal{W}$ that overlap $\mathcal{W}$ significantly,
    where the overlap is computed in the original image by expanding windows
    in coarser scales.

$\mathcal{L}$ is now the list of detected objects.
```

Algorithm 17.1: Sliding Window Detection.

The sliding window detection recipe is wholly generic and behaves very well in practice. Different applications require different choices of feature and sometimes benefit from different choices of feature. Notice that there is a subtle interaction between the size of the window, the steps Δx and Δy, and the classifier. For example, if we work with windows that tightly surround the object, then we might be able to use a classifier that is more tightly tuned, but we will have to use smaller steps and so look at more windows. If we use windows that are rather larger than the object, then we can look at fewer windows, but our ability to detect objects next to one another might be affected, as might our ability to localize the objects. Cross-validation is one way to make appropriate choices here. As a result, there is some variation in the appearance of the window caused by the fact our search is quantized in translation and scale; the training tricks in Section 15.3.1 are extremely useful for controlling this difficulty.

17.1.1 Face Detection

In frontal views at a fairly coarse scale, all faces look basically the same. There are bright regions on the forehead, the cheeks, and the nose, and dark regions around the eyes, the eyebrows, the base of the nose, and the mouth. This suggests approaching face finding as a search over all image windows of a fixed size for windows that look like a face. Larger or smaller faces can be found by searching coarser- or finer-scale images.

A face illuminated from the left looks different than a face illuminated from the right, which might create difficulties for the classifier. There are two options: we could use HOG features, as in Section 5.4; or we could correct the image window to reduce illumination effects. The pedestrian detector of Section 17.1.2 uses HOG features, so we will describe methods to correct image windows here.

Generally, illumination effects look enough like a linear ramp (one side is bright, the other side is dark, and there is a smooth transition between them) that we can simply fit a linear ramp to the intensity values and subtract that from the image window. Another way to do this would be to log-transform the image and then subtract a linear ramp fitted to the logs. This has the advantage that (using a rather rough model) illumination effects are additive in the log transform. There doesn't appear to be any evidence in the literature that the log transform makes much difference in practice. Another approach is to histogram equalize the window to ensure that its histogram is the same as that of a set of reference images (histogram equalization is described in Figure 17.1).

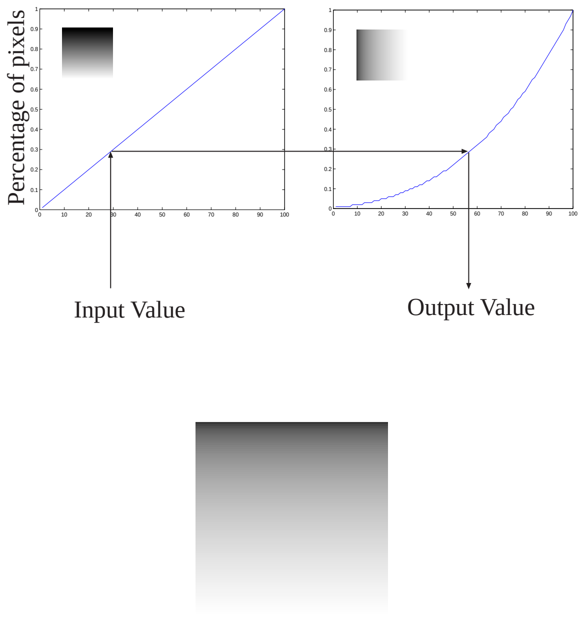

FIGURE 17.1: Histogram equalization uses cumulative histograms to map the gray levels of one image so that it has the same histogram as another image. The figure at the top shows two cumulative histograms with the relevant images inset in the graphs. To transform the left image so that it has the same histogram as the right image, we take a value from the left image, read off the percentage from the cumulative histogram of that image, and obtain a new value for that gray level from the inverse cumulative histogram of the right image. The image on the **left** is a linear ramp (it looks nonlinear because the relationship between brightness and lightness is not linear); the image on the **right** is a cube root ramp. The result—the linear ramp, with gray levels remapped so that it has the same histogram as the cube root ramp—is shown on the **bottom row**.

Once the windows have been corrected for illumination, we need to determine whether there is a face present. The orientation isn't known, and so we must either

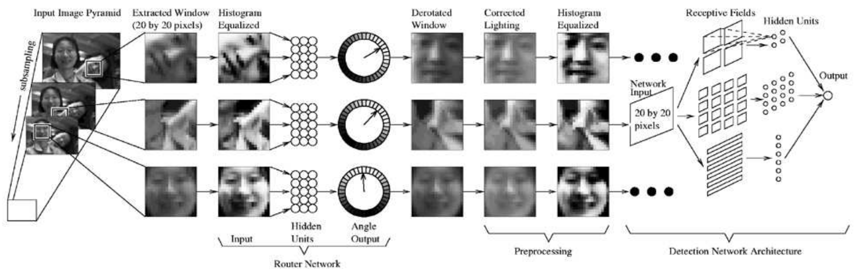

FIGURE 17.2: The architecture of Rowley, Baluja, and Kanade's system for finding faces. Image windows of a fixed size are corrected to a standard illumination using histogram equalization; they are then passed to a neural net that estimates the orientation of the window. The windows are reoriented and passed to a second net that determines whether a face is present. *This figure was originally published as Figure 2 from "Rotation invariant neural-network based face detection," H.A. Rowley, S. Baluja, and T. Kanade, Proc. IEEE CVPR, 1998, © IEEE, 1998.*

determine it or produce a classifier that is insensitive to orientation. A *neural net* is a procedure for parametric regression that produces an output that is a function of input and parameters. Neural nets are typically trained by gradient descent on an error function that compares computed output to labels for numerous labeled examples. Rowley *et al.* (1998*b*) produced a face finder that finds faces very successfully by first estimating the orientation of the window using one neural net then reorienting the window so that it is frontal, and then passing the frontal window onto another neural net (see Figure 17.2; the paper is a development of Rowley *et al.* (1996) and (1998*a*)). The orientation finder has 36 output units, each coding for a 10° range of orientations; the window is reoriented to the orientation given by the largest output. Examples of the output of this system are given in Figure 17.3.

There is now an extremely rich face detection literature based on the sliding window recipe. The most important variant is due to Viola and Jones (2001), who point out that a clever choice of classifier and of features results in an extremely fast system. The key is to use features that are easy to evaluate to reject most windows early. Viola and Jones (2001) use features that are composed of sums of the image within boxes; these sums are weighted by 1 or −1, then added together. This yields the form

$$\sum_k \delta_k B_k(\mathcal{I}),$$

where $\delta_i \in \{1, -1\}$ and

$$B_k(\mathcal{I}) = \sum_{i=u_1(k)}^{u_2(k)} \sum_{j=v_1(k)}^{v_2(k)} \mathcal{I}_{ij}.$$

Such features are extremely fast to evaluate with a device called an *integral image*. Write $\hat{\mathcal{I}}$ for the integral image formed from the image $\mathcal{I}$. Then

$$\hat{\mathcal{I}}_{ij} = \sum_{u=1}^{i} \sum_{v=1}^{j} \mathcal{I}_{uv}.$$

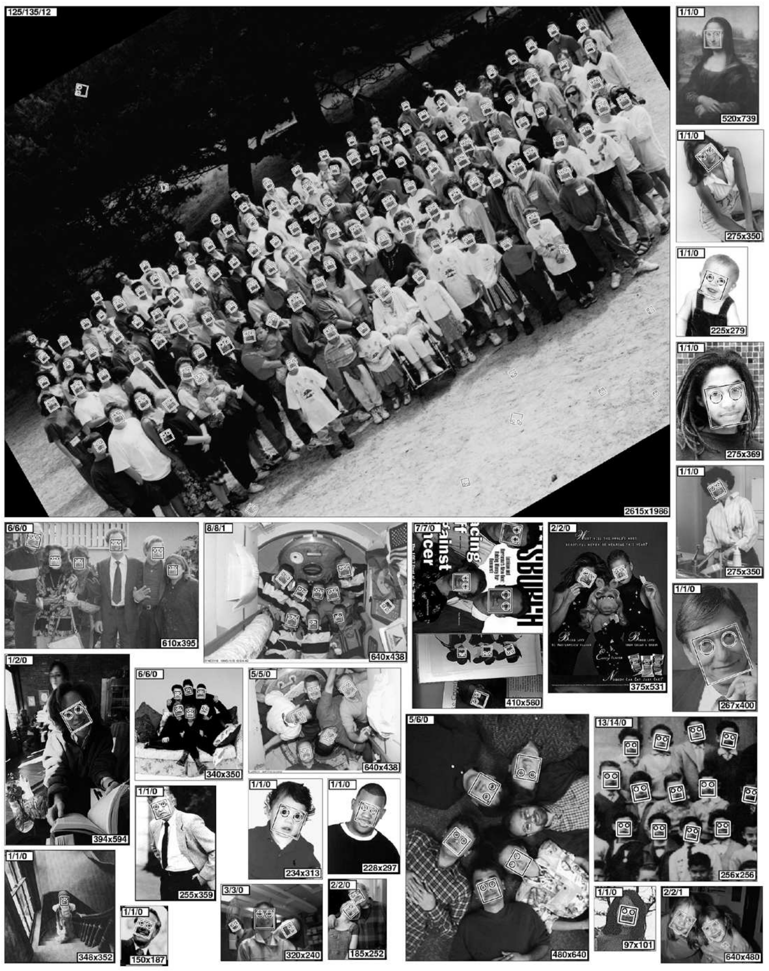

FIGURE 17.3: Typical responses for the Rowley, Baluja, and Kanade system for face finding; a mask icon is superimposed on each window that is determined to contain a face. The orientation of the face is indicated by the configuration of the eye holes in the mask. *This figure was originally published as Figure 7 from "Rotation invariant neural-network based face detection," H.A. Rowley, S. Baluja, and T. Kanade, Proc. IEEE CVPR, 1998, © IEEE, 1998.*

This means that any sum within a box can be evaluated with four queries to the integral image. It is easy to check that

$$\sum_{i=u_1}^{u_2} \sum_{j=v_1}^{v_2} \mathcal{I}_{ij} = \hat{\mathcal{I}}_{u_2 v_2} - \hat{\mathcal{I}}_{u_1 v_2} - \hat{\mathcal{I}}_{u_2 v_1} + \hat{\mathcal{I}}_{u_1 v_1},$$

which means that any of the features can be evaluated by a set of integral image queries. Now imagine we build a boosted classifier, using decision stumps based around these features. The resulting score will be a weighted sum of binary terms,

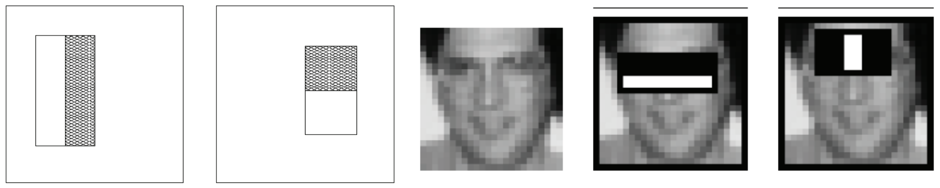

FIGURE 17.4: Face detection can be made extremely fast using features that are easy to evaluate, and that can reject most windows early. On the **left**, features can be built up out of sums of the image within boxes, weighted by 1 or −1. The drawings show two two-box features (some readers might spot a relationship to Haar wavelets). On the **right**, the features used for the first two tests (equivalently, the first two classifiers in the cascade) by Viola and Jones (2001). Notice how they check for the distinctive dark bar at the eyes with the lighter bar at the cheekbones, then the equally distinctive vertical specularity along the nose and forehead. *This figure was originally published as Figures 1 and 3 from "Rapid Object Detection using a Boosted Cascade of Simple Features," by P. Viola and M. Jones, Proc. IEEE CVPR 2001 © IEEE 2001.*

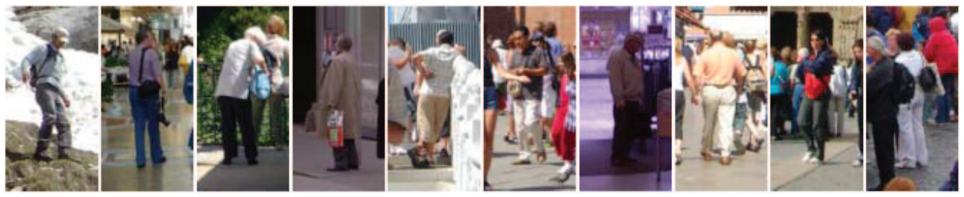

FIGURE 17.5: Examples of pedestrian windows from the INRIA pedestrian dataset, collected and published by Dalal and Triggs (2005). Notice the relatively strong and distinctive curve around the head and shoulders; the general "lollipop" shape, caused by the upper body being wider than the legs; the characteristic "scissors" appearance of separated legs; and the strong vertical boundaries around the sides. These seem to be the cues used by classifiers. *This figure was originally published as Figure 2 of "Histograms of Oriented Gradients for Human Detection," N. Dalal and W. Triggs, Proc. IEEE CVPR 2005, © IEEE, 2005.*

one for each feature. Now we can order the features by complexity of evaluation (for example, two box features will be much faster to evaluate than ten box features). For the simplest feature, we can then adjust the threshold of the weak learner such that there are few or no false negatives. Now any window that returns a feature value below that threshold can be rejected *without looking at the other features*; this means that many or most image windows can be rejected at the first test (Figure 17.4). If the window passes the first test, we can test the next feature with a threshold adjusted so it produces few or no false negatives *on the output of the first test.* Again, we expect to be able to reject many or most windows. We apply this strategy repeatedly to get an architecture of repeated classifiers referred to as a *cascade.* Classifiers in the cascade do not need to use only a single feature. Viola and Jones (2001) train the cascade by requiring that each stage meet or exceed targets for the reduction in false positives (which should be big) and the decrease

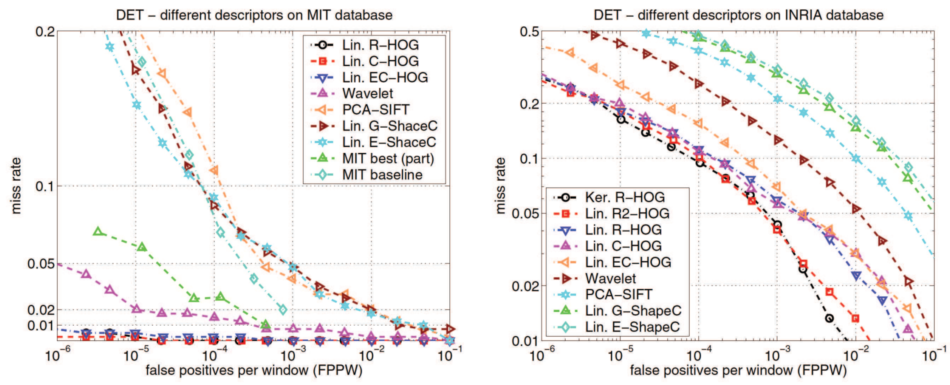

FIGURE 17.6: The performance of the pedestrian detector of Dalal and Triggs (2005), for various choices of features and two different datasets. On the **left**, results using the MIT pedestrian dataset, and on the **right**, results using the INRIA dataset. The results are reported as the miss rate (so smaller is better) against the false positive per window (FPPW) rate, and so evaluate the classifier rather than the system. Overall system performance will depend on how many windows are presented to the detector in an average image (details in the text; see Figure 17.8). Notice that different datasets result in quite different performance levels. The best performance on the INRIA dataset (which is quite obviously the harder dataset) is obtained with a kernel SVM (circles, Ker. R-HOG), but there is very little difference between this and a linear SVM (squares, Lin. R2-HOG). *This figure was originally published as Figure 3 of "Histograms of Oriented Gradients for Human Detection," N. Dalal and W. Triggs, Proc. IEEE CVPR 2005, © IEEE, 2005.*

in detection rate (which should be small); they add features to the stage until the targets are met.

Generally, frontal face detection is now a reliable part of vision systems (e.g., Section 21.4.4); usually other components of a system cause more problems than face detection does. It is much more difficult to detect faces in lateral views; there seem to be two major reasons. First, the profile of the face is quite important, and variable between subjects. This means that classifier windows must take an awkward shape, and some pixels in the window do not lie on the face and so contribute noise. Second, lateral views of faces seem to have a less constrained appearance than frontal views, so that classifiers must be more flexible to find them.

17.1.2 Detecting Humans

Being a pedestrian is dangerous, and even more so if one is intoxicated. Counting pedestrian deaths is hard, but reasonable estimates give nearly 900,000 pedestrians killed worldwide in 1990 (Jacobs and Aeron-Thomas 2000). If a car could tell whether it were heading for a pedestrian, it might be able to prevent an accident. As a result, there is much interest in building pedestrian detectors.

The sliding window recipe applies naturally to pedestrian detection because pedestrians tend to take characteristic configurations. Standing pedestrians look

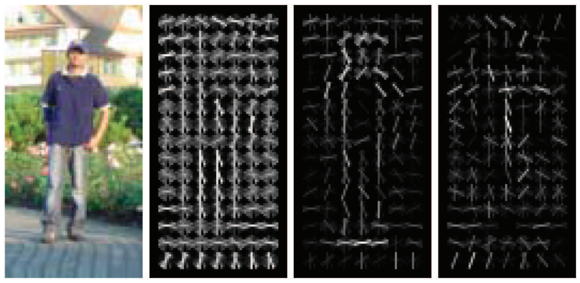

FIGURE 17.7: As Figure 17.6 indicates, a linear SVM works about as well as the best detector for a pedestrian detector. Linear SVMs can be used to visualize what aspects of the feature representation are distinctive. On the **left**, a typical pedestrian window, with the HOG features visualized on the **center left**, using the scheme of Figure 5.15. Each of the orientation buckets in each window is a feature, and so has a corresponding weight in the linear SVM. On the **center right**, the HOG features weighted by positive weights, then visualized (so that an important feature is light). Notice how the head and shoulders curve and the lollipop shape gets strong positive weights. On the **right**, the HOG features weighted by the absolute value of negative weights, which means a feature that strongly suggests a person is not present is light. Notice how a strong vertical line in the center of the window is deprecated (because it suggests the window is not centered on a person). *This figure was originally published as Figure 6 of "Histograms of Oriented Gradients for Human Detection," N. Dalal and W. Triggs, Proc. IEEE CVPR 2005, © IEEE, 2005.*

like lollipops (wider upper body and narrower legs), and walking pedestrians have a quite characteristic scissors appearance (Figure 17.5). Dalal and Triggs (2005) invented HOG features for this purpose, and used a linear SVM to classify windows, because it is as good as the best classifier, but simpler (Figure 17.6). Another advantage of a linear SVM is that one can get some insight into what features are distinctive (Figure 17.7).

Evaluating sliding window methods can be difficult. Dalal and Triggs (2005) advocate plotting the detection rate (percentage of true positives detected) against the false positives per window (FPPW). Figure 17.6 shows performance for various configurations of their system plotted on these axes. When evaluating these plots, it is important to keep in mind that they characterize the behavior of the *classifier*, rather than the whole system. This is attractive if you are interested in features and classifiers, but perhaps less so if you are interested in systems. A higher FPPW rate may be tolerable if you have to look at fewer windows, though looking at fewer windows might affect the detect rate. Dollar *et al.* (2009) have conducted a systematic evaluation of pedestrian detectors on a large dataset built for that purpose. As Figure 17.8 shows, the ranking of methods changes depending on whether one plots FPPW or false positive per image (FPPI); generally, we expect that FPPI is more predictive of performance in applications.

Our sliding window recipe has one important fault: it assumes that windows are independent. In pedestrian detection applications, windows aren't really independent, because pedestrians are all about the same size, have their feet on or close to the ground, and are usually seen outdoors, where the ground is a plane. If

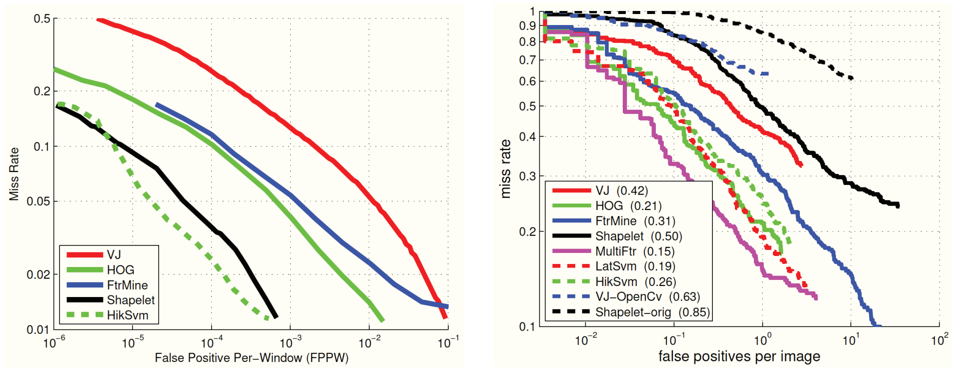

FIGURE 17.8: The FPPW statistic is useful for evaluating classifiers, but less so for evaluating systems. On the **left**, results on the INRIA pedestrian dataset for a variety of systems, plotted using miss rate against FPPW by Dollar *et al.* (2009). In this plot, curves that lie lower on the figure represent better performance (because they have a lower miss rate for a given FPPW rate). On the **right**, results plotted using miss rate against false positive per image (FPPI), a measure that takes into account the number of windows presented to the classifier. Again, curves that lie lower are better. Notice how different the ranking of the systems is. *This figure was originally published as Figure 8 of "Pedestrian Detection: A Benchmark" P. Dollár, C. Wojek, B. Schiele, and P. Perona, Proc. IEEE CVPR 2009 © IEEE 2009.*

we knew the horizon of the ground plane and the height of the camera above that ground plane, then many windows could not be legitimate pedestrians. Windows whose base is above the horizon would be suspect because they would imply pedestrians in the air; windows whose base is closer to the horizon should be smaller (otherwise, we would be dealing with gigantic pedestrians). The height of the camera above the ground plane matters because in this problem there is an absolute scale, given by the average height of a pedestrian. Assume the horizon is in the center of the image. Then, for cameras that are higher above the ground plane, legitimate pedestrian windows get smaller more quickly as their base approaches the horizon. There are two strong sources of information about the horizon and the camera height. First, the textures of the ground, buildings, and sky are all different, and these can be used to make a rough decomposition of the image that suggests the horizon. Second, observing some reliable detection responses should give us clues to where the horizon lies, and how high the focal point is above the ground plane. Hoiem *et al.* (2008) show that these global geometric cues can be used to improve the behavior of pedestrian and car detectors (Figure 17.9; see also Hoiem *et al.* (2006)).

17.1.3 Detecting Boundaries

Edges are not the same as occluding contours, as we said in Chapter 5, because many effects—changes in albedo, shadow boundaries, fast changes in surface normal—can create edges. Rather than relying on the output of an edge detector, we could explicitly build an occluding contour detector, using the sliding window recipe. At

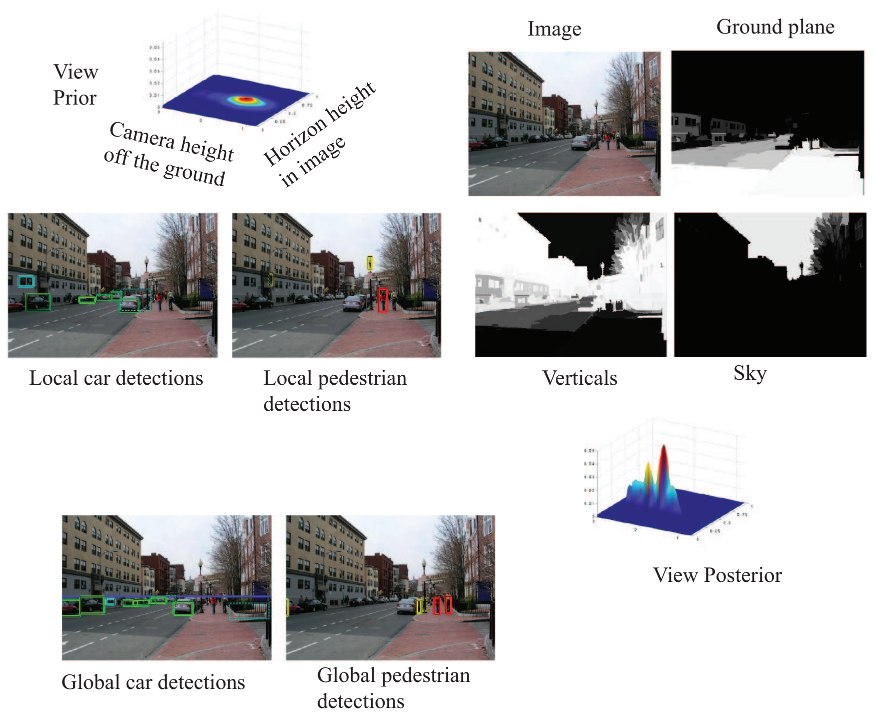

FIGURE 17.9: Hoiem *et al.* (2008) show geometric consistency can be used to improve detector performance. The main parameters are the height of the camera above the ground, and the positition of the image horizon. The texture of the ground plane, the sky, and vertical walls tend to be different, so that discriminative methods can classify pixels into these classes; with this information, combined with detector responses (**local detector results**), they obtain a significantly improved posterior estimate of the geometric parameters, and an improved detection rate for a given false positive rate (**global detector results**). *This figure was originally published as Figure 5 of "Putting Objects in Perspective," by D. Hoiem, A. Efros, and M. Hebert, Proc. IEEE CVPR 2006 © IEEE 2006.*

each window, we would look at a set of relevant features within the window, then use these to decide whether the pixel at the center of the window is an occluding contour or not. In practice, it is sometimes more useful to produce the posterior probability that each pixel lies on a boundary, at that pixel. Martin *et al.* (2004), who pioneered the method, call these maps the P_b, for **probability of boundary**.

For this problem, it makes sense to work with circular windows. Boundaries are oriented, so we will need to search over orientations. Each oriented window can be visualized as a circle cut in half by a line through the center. If this line is an object boundary, we expect substantial differences between the two sides, and so features will compare these sides. Martin *et al.* (2004) build features for a set of properties (raw image intensity, oriented energy, brightness gradient, color gradient, raw texture gradient, and localized texture gradient) by taking a histogram

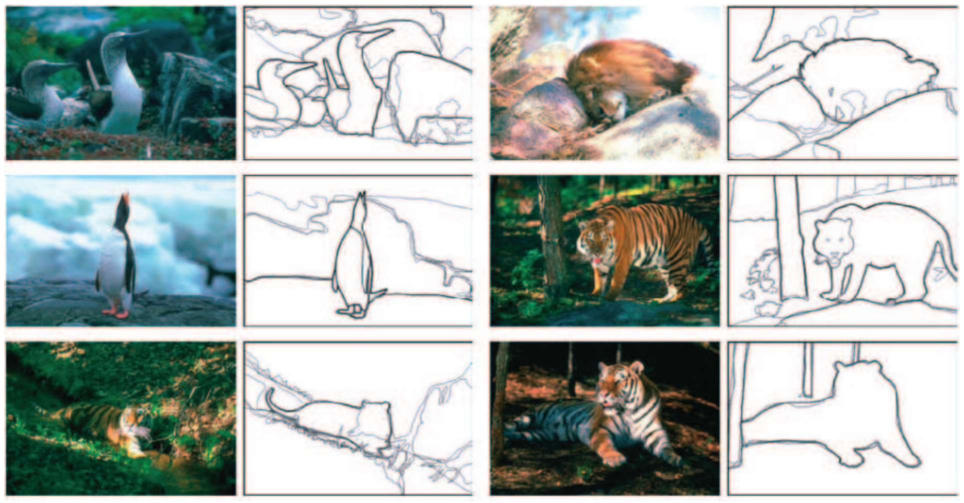

FIGURE 17.10: Object boundaries marked by human informants for some images from the Berkeley segmentation dataset, used by Martin *et al.* (2004) to train detectors that report the probability of boundary. Maps produced by many informants have been averaged, so that pixels are darker when many informants agree that they represent boundaries. *This figure was originally published as Figure 1 of "Learning to Detect Natural Image Boundaries Using Local Brightness, Color, and Texture Cues," by D.R. Martin, C.C. Fowlkes, and J. Malik, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004 © IEEE 2004.*

representing that property for each side, then computing the χ^2 distance between the histograms. This means that each feature encodes the tendency of a particular property to look different on the two sides of the circle. This set of features is then supplied to logistic regression.

The boundary detector is trained using images whose boundaries have been marked by humans (Figure 17.10). Human annotators don't get boundaries perfectly right (or else they'd agree, which they certainly don't; see also Figure 17.12). This means that the training dataset might contain multiple copies of the same window with different annotations—some humans marked the point a boundary point, and others didn't. However, the set of windows is very large, so that such inconsistencies should be averaged out in training. The procedure we have described can be used to build two methods. One reports $P_b(x, y, \theta)$, that is, the probability the point is a boundary point as a function of position and orientation; the other reports $P_b(x, y) = \max_\theta P_b(x, y, \theta)$. The second is most widely used (Figure 17.11).

Testing requires some care, because reporting a boundary point close to, but not on, a boundary point marked by a human is not a failure. Martin *et al.* (2004) cope with this issue by building a weighted matching between the boundary points marked by a human and those predicted by the method. Weights depend on distance, with larger distances being more unfavorable. A predicted boundary point too far away from any human-marked point is a false positive. Similarly, if there are no boundary points predicted close enough to a human-marked point, then that

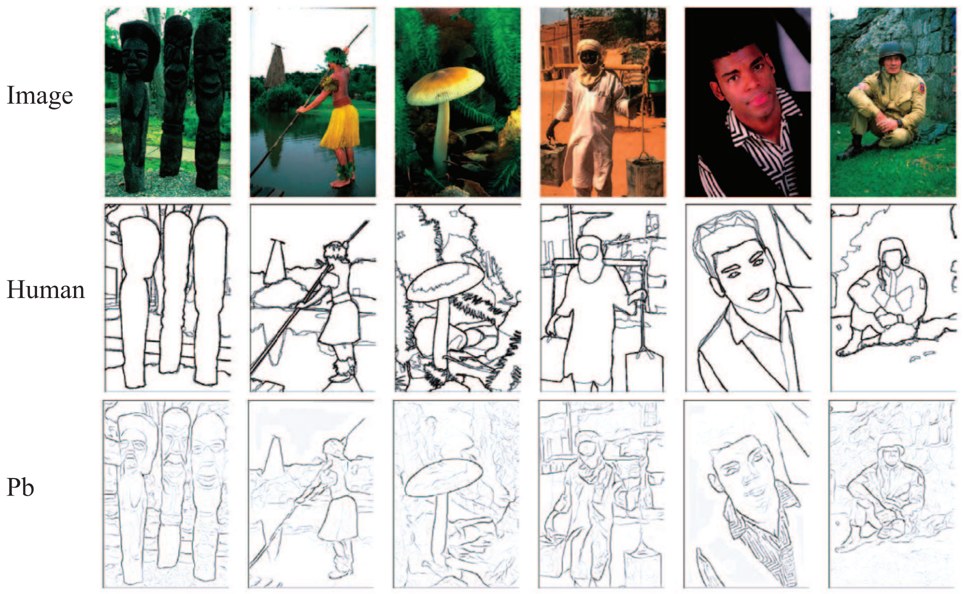

FIGURE 17.11: Some images from the dataset used by Martin *et al.* (2004). Boundaries predicted by humans (averaged over multiple informants; darker pixels represent boundary points on which more informants agree) compare well with boundaries predicted by the P_b method. Some P_b errors are unavoidable (see the detailed windows in Figure 17.13); the method has no-long scale information about the objects present in the image. *This figure was originally published as Figure 15 of "Learning to Detect Natural Image Boundaries Using Local Brightness, Color, and Texture Cues," by D.R. Martin, C.C. Fowlkes, and J. Malik, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004 © IEEE 2004.*

point counts as a false negative. We can then threshold the P_b map at some value, and compute recall and precision of the method; by varying the threshold, we get a recall-precision curve (Figure 17.12). Although this method doesn't perform as well as humans, who can use context and object identity cues (and so predict illusory contours, as in Figure 17.13), it significantly outperforms other methods for finding boundaries. P_b is now widely used as a feature, and implementations are available (Section 17.3.1). The most recent variant is globalP_b, which gets improved results by linking the P_b method to a segmenter, and so filling in pixels that are required to ensure that object boundaries are closed curves. You can see this as a method to force windows not to be independent. Precision-recall curves for this method appear in Figure 9.25, which should be compared with Figure 17.11.

17.2 DETECTING DEFORMABLE OBJECTS

The basic sliding window detection recipe is extremely powerful. It does assume (incorrectly) that windows are independent, but we have shown ways to manage the cost of that assumption. However, the recipe must fail when the classifier fails. There are two important effects that cause the classifier to fail: The object might