The Histogram Feature – A Resource-Efficient Weak Classifier

Niklas Pettersson, Lars Petersson and Lars Andersson
NICTA
Locked Bag 8001, Canberra, Australia
{niklas.pettersson,lars.petersson,lars.andersson}@nicta.com.au

Abstract— **This paper presents a Weak Classifier that is extremely fast to compute, yet highly discriminant. This Weak Classifier may be used in, for example, a boosting framework and is the result of a novel way of organizing and evaluating Histograms of Oriented Gradients. The method requires only one access to main memory to evaluate each feature, in comparison with the more well-known Haar features which require somewhere between six and nine memory accesses to evaluate each feature. This low memory bandwidth makes the Weak Classifier especially ideal for use in small systems with little or no memory cache available.**

The presented Weak Classifier has been extensively tested in a boosted framework on data sets consisting of pedestrians and various road signs. The classifier yields detection results that are far superior than the results obtained from Haar features when tested on road signs and similar structures, whereas the detection results are comparable to those of Haar features when tested on pedestrians. In addition, the computational resources necessary for these results have been shown to be considerably smaller for the new weak classifier.

I. Introduction

Today, there are many successful pattern recognition approaches in the domain of visual detection [1] [2] [3] [4]. Many of these approaches give good detection performance, and with varying support for properties such as scale invariance, rotational invariance and perspective invariance.

Histograms of Oriented Gradients (HOGs) were used by Dalal and Triggs in [1] to detect pedestrians. However, this method required scanning a single image at 800 different positions which took about 0.5 seconds. In [2], this was significantly improved on by using integral images for each orientation and using a cascaded classifier approach.

In [4], the authors used HOGs as hand-crafted templates to detect road signs. However, in this paper, we detect road signs by using HOGs as low-level features. These low-level features are then combined by AdaBoost [5] into a strong classifier. In this paper, we also present a method for harnessing the discriminant nature of this type of feature in a general object detection framework.

A big step towards real-time object detection was taken with the work of Viola & Jones [3], which uses a boosted strong classifier consisting of Haar features that are easy to compute. However, even though the Haar features are cheap to compute, a substantial bandwidth to memory is necessary in order to support real-time computation of these features in large high-resolution images. This becomes a problem if the algorithm is to be implemented on very cheap embedded systems where there might be little or no memory cache available. Hence, there is a need for other methods that require less bandwidth to the memory, and this is where this paper makes a contribution.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

II. Method

The bottleneck limiting the speed of most visual detection algorithms is not the speed of the CPU but rather the bandwidth to the memory. Since image processing is memory intensive by nature, data does not normally fit in the fast cache memories. What makes matters worse is that detection algorithms often have a semi-random way of accessing memory. This leads to poor cache performance which causes the CPU to be idle until the lengthy memory access request to main memory returns.

Using the Haar feature with its integral image [3] as an illustration, one can see that the integral image computation is linear and requires only a minimum number of main memory accesses. However, when the Haar feature is calculated by accessing the integral image, this is no longer done in a memory efficient way. Typically, a Haar feature requires six to nine memory accesses spread out over the integral image in an unordered way. This contributes to a high rate of cache misses which causes the CPU to be idle. The main reason why the concept of a pre-computed integral image has been so successful is that it is re-used a large number of times to evaluate the Haar features for other patches in the same input image.

With the above reasoning in mind, it is clear that we somehow need to reduce the number of nonlinear memory accesses during feature computation. Hence, we propose to move more of the feature computation from the evaluation step to the preprocessing step. If we were to apply this idea to the case of Haar features, this would mean incorporating more information into the integral image so that every memory access can return a more useful value and the number of accesses can be reduced. This would, in turn, limit the memory bandwidth needed. More memory-friendly streamlined processing, such as MMX and SSE optimizations, can then be used at the preprocessing step.

Similar to work done by [1], we chose to work with gradients rather than with intensities, as edges often carry a significant amount of information about the underlying texture. From the above, we concluded that two crucial prerequisites for an efficient implementation are a) to construct a pre-computed entity that can be reused heavily for different parts of the input image; and b) to keep the number of necessary accesses to this entity to a minimium.

We achieved this by creating a *histogram image*. This is an "image" where each 32-bit word represents a histogram of orientations of a 4×4 pixel area in the input image. Just like the integral image is the basis for the fast computation of Haar features, this histogram image is the basis for the fast computation of our weak classifier (or feature), which we call *HistFeat*.

Calculation of these features only requires a single 32-bit word to be read from the histogram image. This is a reduction in memory accesses by a factor of six to nine when compared with Haar features. Moreover, we show later in this paper that, for particular classes of objects, a same-sized collection of HistFeat is more discriminant than the corresponding collection of Haar features.

Also, it is important to note that although the results presented here use the features in the context of a cascade of boosted strong classifiers, it may be used in other machine learning frameworks as well.

III. Implementation

A. Computing the Histogram Image

The histogram image requires more computations to calculate than the integral image. However, our experimental results indicate that the histogram image has a higher information density as well. Hence, we can query the histogram image fewer times in order to classify a patch with the same confidence as for the integral image. Thus, the goal is to move computational effort away from the evaluation step to the preprocessing step where it can be pipelined.

An overview of the computations needed to produce the histogram image is shown in Figure 1.

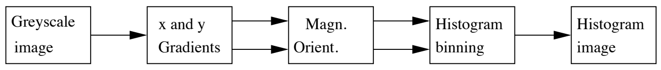


Fig. 1. Given a greyscale image, we first compute the x and y gradients at each pixel in the image. These gradients are then used to compute the magnitude and orientation at each pixel. We then use these magnitudes and orientations to create histograms at each possible 4×4 image patch. Each of these histograms then becomes a pixel in the histogram image.

1) Computing the gradients: The x and y gradients of a greyscale image can be calculated in a number of ways. The simplest is finite differences with the $[-1, 0, 1]$ kernel. This method was also shown to be the most discriminate in [1]. Since it is also easy to implement using SSE2 technology, this is the method we have adopted for computing the x and y gradients.

2) Orientation and magnitude: We then use these x and y gradients to compute the orientations and magnitudes at each pixel. A pixel's orientation is represented by a number between zero and seven. Hence, it can be stored as a 3 bit value.

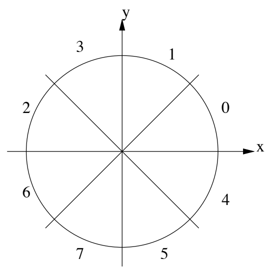


Fig. 2. We divide orientation space into 8 bins. Each pixel in the greyscale image is assigned one of these 8 orientations, depending on which sector of orientation space its orientation falls in.

Reducing the number of different orientations to such a low number as eight means that we can simply implement this via a sequence of comparisons. More specifically,

$$ori_{x,y} = (G_y < 0) \cdot 4 + (G_x < 0) \cdot 2 + (|G_y| > |G_x|) \cdot 1 \quad (1)$$

where $ori_{x,y}$ is the orientation at pixel (x, y), G_x is the x gradient at pixel (x, y) and G_y is the y gradient at pixel (x, y). This results in the unconventional encoding of the different orientations shown in Figure 2.

This choice of reducing the orientation information to as few as eight bins might seem crude. However, as we will show, we still get good discrimination performance and the features have a built-in invariance to small rotations.

While calculating the orientation associated with each pixel, we simultaneously calculate the pixel's approximate magnitude by taking the absolute sum of the two gradients G_x and G_y. This is again a tradeoff between the more mathematically correct Euclidean length of the vector and the less accurate but more computationally efficient sum of absolute values. This value is immediately thresholded. Thus, only one bit is used to represent the magnitude.

3) Histogram binning: Once the orientations and magnitudes of each pixel have been calculated, we compute a histogram for each possible 4×4 image patch in the greyscale image by summing the orientations in that small image patch. However, only those orientations whose corresponding magnitude is greater than some chosen threshold is included. Each of these histograms then becomes a pixel in the histogram image.

We chose to represent each histogram as a 32 bit word (i.e. four bits for each orientation) as shown in Figure 3. This allows a full histogram to be read by one single access to the memory. The overall storage needed for the histogram image is the same as for the integral image - 32 bits per pixel in the input image.

In our experiments, each histogram represents a small 4×4 image patch. If all orientations in this image patch line up in the same direction, then this sums to 16. Therefore, a check is needed to avoid overflow when storing this as 4 bits.

B. Classifier Evaluation

To evaluate one feature in a selected window, one histogram is fetched from the histogram image. From this histogram, two bins (as defined by each individual feature) are

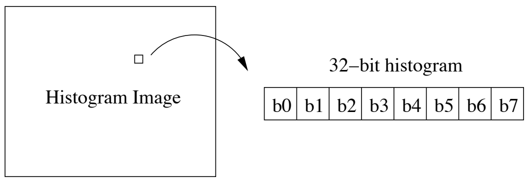


Fig. 3. Each pixel in the histogram image actually has a histogram of the orientations in a 4×4 neighbourhood. Each orientation is represented by 4 bits.

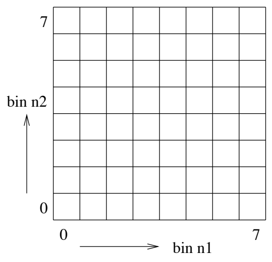


Fig. 4. The *a posteriori* map. Two bins selected from the histogram of orientations are used to index the map.

used as coordinates in a lookup table containing confidence values (the *a posteriori* map). This lookup table is shown in Figure 4.

The resulting confidence value is then used in the summation, as in the boosted classifier approach, to form the strong classifier. Hence, there is virtually no arithmetic processing needed in the evaluation of each feature.

The Haar feature, on the other hand, uses a number of arithmetic operations to evaluate each feature. Although this does give HistFeat an advantage with respect to the number of operations necessary, its biggest advantage lies in the low number of memory accesses and its discriminative power.

Now, each feature is defined by a number of parameters. These parameters include the position in the patch relative to the top left corner and the indices of the two selected histogram bins, $n1$ and $n2$, used for the table lookup. These parameters span the space of features available to the boosting algorithm to select from. The size of this feature space can be calculated as $(W-4+1)(H-4+1)(8)(7)$ where W and H represents the width and height of the classifier respectively. As an example, this results in 125,048 different classifiers for a typical 32×80 pedestrian classifier.

1) The a posteriori map: In the training phase, each feature creates its own *a posteriori* map in the same way as described for the Haar feature in [6]. That is, a confidence value is calculated for each position in the lookup table based on the observed statistics of the positive and negative training set. However, the lookup table used in the HistFeat feature is a 2D table of size 8×8 or 16×16 as shown in Figure 4.

C. Other architectures

In the description above, we have made references to efficient implementation on CPUs using the extended instruction sets SSE and MMX. However, the method is well-suited for implementation on any architecture that supports pipelined execution of the computations, most notably FPGAs. For example, the gradient calculation can be implemented in an FPGA as in [7].

IV. Experiments

In this section, we perform a set of experiments to evaluate different aspects of the presented weak classifier. These aspects include

- the time to compute the histogram image vs the integral image;
- the size of the strong classifier that needs to be constructed by AdaBoost in order to reach the same performance; and
- the number of features that can be evaluated per second in video data.

Again, we compare our results with those of the Haar feature as this is a well-known entity.

A. Time to Compute the Histogram Image

In order to evaluate the complexity of the histogram image, we performed both a) absolute measurements for different sized images and b) relative measurements against the integral image. To get accurate measurements, we used Oprofile [8].

Although the computation of both the histogram image and the integral image should not depend on the contents of the input images, we used real video sequences as the measuring stick. The sequences used have a resolution of 768×576 and were taken at 2 metre intervals along the road. These were then scaled to different sizes (while keeping the aspect ratio constant) to illustrate possible cache size issues. Furthermore, we ran the tests on different CPU architectures: Dual Core AMD Opteron 275 (using only one core) and Intel Core 2 Duo 1.6 GHz. Only the time spent computing the histogram image was considered.

Figure 5 shows the comparison between the histogram image and the integral image. At a standard PAL resolution of 768×576 pixels, it takes about 8.8 milliseconds to compute the histogram image compared with 2.6 milliseconds for the integral image on an AMD64 (2.2GHz). Note that the complexity of this preprocessing stage is linear with the number of pixels in the input image.

From this graph, it is evident from the nearly perfect linear behavior that there are no cache issues arising at different image sizes. It is also clear that the histogram image takes nearly 3.5 times longer to compute. Further analysis will show that the efficient evaluation and discrimination performance of HistFeat will more than make up for this penalty.

B. Size of Strong Classifier for Equal Performance

Strong classifiers can be trained in a variety of ways. Most often, training is done in a cascaded fashion with a number of stages one after the other, each stage rejecting a proportion of the input data. This is a good procedure when trying to achieve the best possible classifier for a particular application. Unfortunately, this procedure limits the possibilities of a fair comparison between methods. This

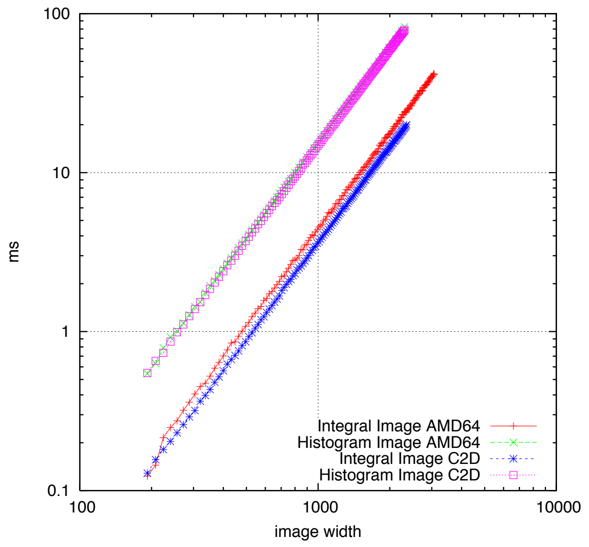


Fig. 5. Logarithmic plot of histogram image and integral image time consumption for input images of different resolutions.

is due to the different sizes of the individual stages, the thresholds used and the many other parameters that change when the underlying weak classifier changes.

Hence, to make a fair comparison of the histogram feature with the Haar feature, only a single stage was used. This results in a strong classifier that is not performing optimally on a particular problem but the comparison between methods becomes more fair.

Fig. 6. Example images of different types of road signs in our dataset.

1) Dataset: Positive input data to the AdaBoost training process for the different object types are listed in Table I. The numbers reflect the original raw hand-labelled images taken by a digital camera, and the number of images after distortions have been applied.

For training, each image has been multiplied using distortions such as occlusions, noise and rotations. Pedestrians have been mirrored only. The reasons for distorting the original images are to simulate a bigger training set and to better span the class of objects. This is particularly useful when only a few object instances are available.

Negative training and validation data was gathered from the same type of original images as the ones where positive data was found (i.e. typically road or city scenes). A total of 10,000 negative training data and 100,000 validation data were used in all experiments.

2) Training and evaluation: For each object type listed in Table I, AdaBoost was used to build single strong classifiers consisting of 1 to 250 features. The features used were HistFeat with a 8×8 and a 16×16 *a posteriori* map, Haar features as implemented in [6], as well as the improvement on Haar features as implemented in [9]. These features are referred to as *h8, h16, vpdf* and *spdf* in the graphs. For each such strong classifier, ROC curves were generated and the time to scan a video frame in typical input data was calculated.

TABLE I

Positive input data, Raw images from a digital camera and Distorted images generated from the original raw images.

Type	Raw Train	Raw Valid	Dist Train	Dist Valid
Speed Sign	1110	222	10000	2500
Stop Sign	129	32	10000	2500
Give Way Sign	154	38	10000	2500
Pedestrian	4926	1865	9852	1865

Based on the ROC curves and timing data, graphs were generated that show the resulting error versus time consumed. These graphs can then be used to give a fair answer regarding which method gives the best performance given a certain amount of computing time.

The behaviour was studied when a) the total error was minimized; b) the false negative rate was set to 0.01; and c) the false positive rate was set to 0.01. The results for all the object types considered are shown in Figure 7.

From these graphs, it is clear that HistFeat (h8 and h16) outperforms both the normal Haar feature (vpdf) and the improved Haar feature (spdf) when speed is critical. It is only when computing time is not essential and can be allowed to be very long that vpdf and spdf can reach lower error scores for some object classes.

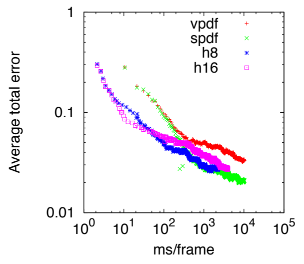
Furthermore, we can see that object classes with strong gradients at well-defined locations (e.g. road signs) are particularly well-suited for HistFeat. In some cases, given a particular scan time, the error rate can differ by more than a factor of 1000 between HistFeat and Haar. However, for the notoriously difficult object class of pedestrians, the difference is not as significant. The difference in error rates for the pedestrian object class is still notable though (by a factor of 2 to 3) for short scan times.

C. Features Per Second

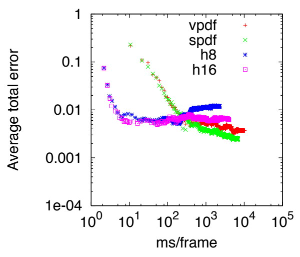
In order to get an estimate of how fast a system will be able to process images, you need to know a few things:

- Preprocessing: This depends on the size of the input image and how many different scales you would like to look at.
- Classification: This depends on how many windows in total you intend to inspect, the average number of features evaluated for each window and the time per feature.

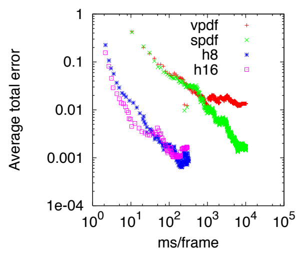
The reason the number of scales needs to be included is that preprocessing will have to be done for each scale. One could claim that the Haar feature does not need to recompute the integral image for different scales. However, in practice, this is the case if one wants to keep the desired performance charateristics of the classifier.



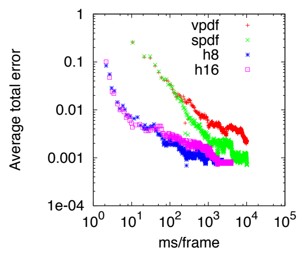
(a) Pedestrians



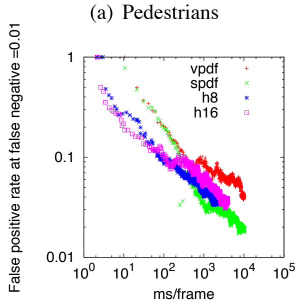
(b) Speed signs



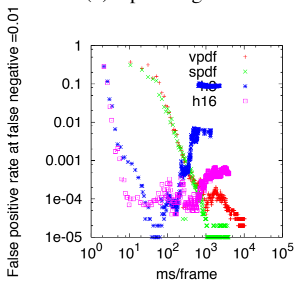
(c) Stop signs



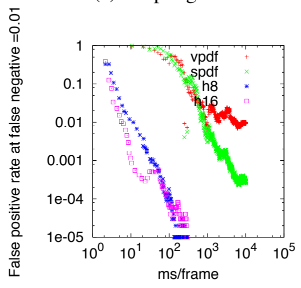
(d) Give way signs



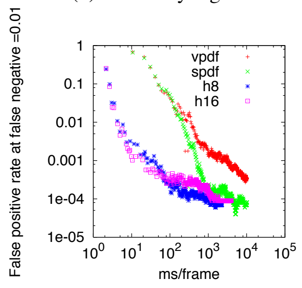
(e) Pedestrians



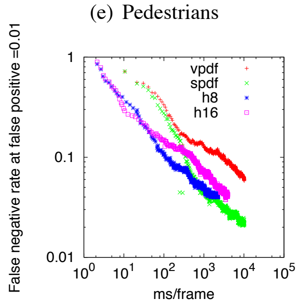
(f) Speed signs



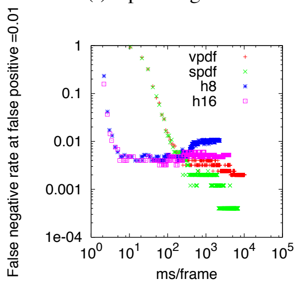
(g) Stop signs



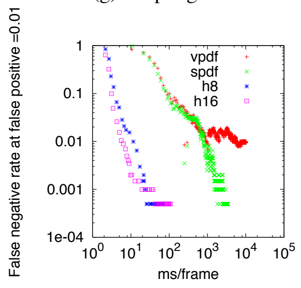
(h) Give way signs



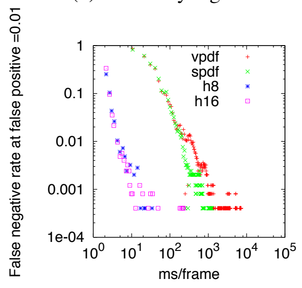
(i) Pedestrians



(j) Speed signs



(k) Stop signs



(l) Give way signs

Fig. 7. Performance versus computing time. The first column shows the results for the pedestrian object class, the second column shows the results for speed signs, the third column shows the results for stop signs and the fourth column shows the results for give way signs. The top row shows average total error versus computing time for the different object classes. The middle row shows the performance for a fixed negative rate of 0.01. The last row shows performance for a fixed false positive rate of 0.01. Note that the pedestrian column has a different scale on the y-axis.

To know the average number of features evaluated in the strong classifier, one would have to know the false positive rate of the different stages in the cascade. Normally, these are set targets when one trains the strong classifier. For instance, suppose we have a classifier with two stages. Suppose the first stage has 10 features and a false positive rate of 0.01, while the second stage has 20 features. (The false positive rate of the second stage does not affect the number of features evaluated.) Also, assume that the majority of the input data is negative. Hence, the average number of features evaluated for this classifier is $10 + 0.01(20) = 10.2$.

This gives the following simple equation for the total time spent per frame

$$T = T_p + f_{ave} \cdot N \cdot T_e \tag{2}$$

where T is the total time spent per frame, T_p is the preprocessing time for one frame, f_{ave} is the average number of features evaluated in an image window, N is the total number of inspected windows in a frame and T_e is the evaluation time per feature.

Now, we have two implementations of the features. The first implementation interprets the description of the classifier as it evaluates each window in the image. This implementation is the most common in the literature. However, the second implementation compiles each strong classifier into an optimized piece of code using gcc. Naturally, the second implementation is much faster than the first implementation, in this case about three times. Viola & Jones [3] reported that their implementation of the Haar feature required about 60 cpu instructions, whereas our compiled version of the Haar feature requires only 40 instructions. The compiled version of HistFeat requires 12 instructions per feature.

In our experiments, we used classifiers of different lengths ranging from one feature to about 200 features in a single stage. This simulates a strong classifier with the average number of features classified between 1 and 200. This was run on the same video sequence as used previously (768×576, 101 frames). The size of the classifier was 32×80 pixels resulting in about 400,000 windows if one uses a step of 1 pixel in x and y over a single scale.

Figures 8 and 9 show the speed at which one can evaluate features. To simplify the comparison, this does not include preprocessing. The two graphs show the result when scanning the input image at a high density (every pixel location) and at a low density (every 8th pixel location). We can then see that not as many features can be evaluated per second in the sparse case. This is caused by cache misses as the same portion of the histogram image and integral image respectively can not be re-used as many times. The peaks in the evaluation speed appearing when the strong classifiers contain less than approximately 50 features is, again, caused by cache behaviour. The entire classifier code and its associated data can then more easily fit in the cache and can quickly be called repeatadely.

It is also worth noting that in the case of sparse scanning, the impact of the longer time to compute the histogram image is more significant. These are important factors to consider when designing a real application with particular needs for certain scanning densities.

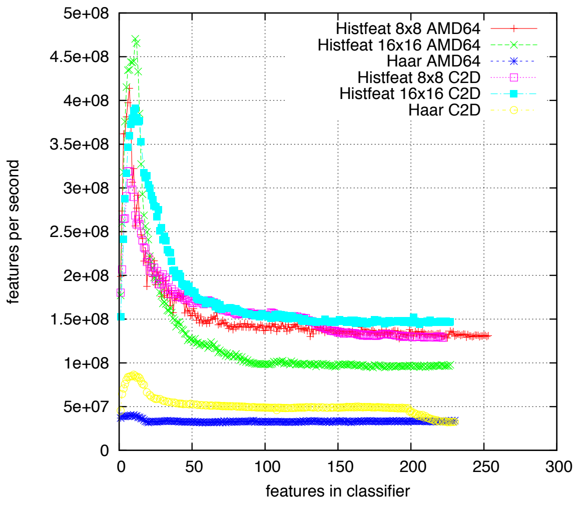


Fig. 8. The number of features that can be evaluated per second for HistFeat with 8×8 and 16×16 PDFs, and the Haar feature with a 128-bin PDF. Measurements were done on a video with 101 frames and a resolution of 768×576. The (x, y) step was $(1, 1)$, which results in about 365,000 patches per frame.

V. Conclusion

In this paper, a novel method for organizing and evaluating Histograms of Oriented Gradients was presented. It was applied as an extremely fast and discriminant weak classifier, HistFeat, for use in a machine learning framework such as AdaBoost.

A series of experiments were conducted on real video data to measure the system performance of HistFeat versus the more well-known Haar feature. Several object classes were used in the comparison such as road signs and pedestrians. It was concluded that although the histogram image takes 3.5 times longer to compute than the integral image, it is more than paid for by the discriminance of HistFeat and the speed with which HistFeat can be evaluated.

HistFeat works particularly well on objects with strong gradients at well-defined locations (e.g. road signs). Pedestrians represent a much more difficult object class and, although HistFeat has superior performance at short scan times, it cannot outperform the Haar feature if very long scan times

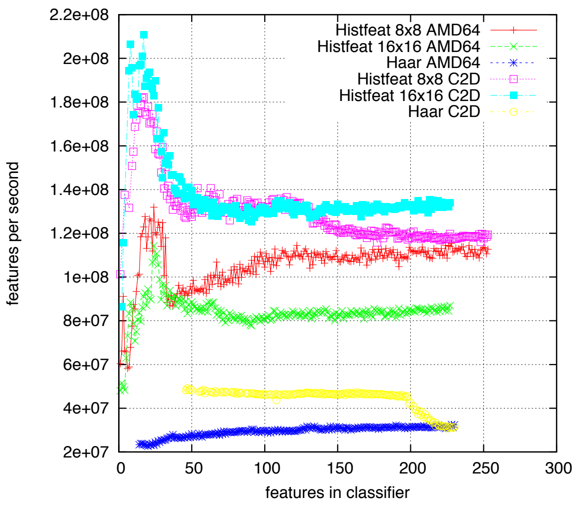


Fig. 9. The number of features that can be evaluated per second for HistFeat with 8×8 and 16×16 PDFs, and the Haar feature with a 128-bin PDF. Measurements were done on a video with 101 frames and a resolution of 768×576. The (x, y) step was $(8, 8)$, which results in about 5700 patches per frame.

are allowed. One reason for this is that the space of HistFeat features that can fit in a scan window is much smaller than the corresponding space of Haar features. Hence, there are just no features available to push the error down further. In a real pedestrian detection application, it makes sense to have a cascade of strong classifiers, with the first few strong classifiers built using HistFeat and the following strong classifiers built using Haar features.

All comparisons of HistFeat were made against both a basic version of the Haar feature [3] [6] as well as a much improved version [9]. These were also highly optimised implementations, governing that the results are indeed competitive.

References

[1] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Proc. of Computer Vision and Pattern Recognition CVPR*, June 2005.

[2] Q. Zhu, M.-C. Yeh, K.-T. Cheng, and S. Avidan, “Fast human detection using a cascade of histograms of oriented gradients,” in *CVPR ’06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 1491–1498.

[3] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” 2001. [Online]. Available: citeseer.ist.psu.edu/viola04rapid.html

[4] B. Alefs, G. Eschemann, H. Ramoser, and C. Beleznai, “Road sign detection from edge orientation histograms,” in *IEEE Intelligent Vehicles Symposium (IV2007)*, June 2007.

[5] R. E. Schapire and Y. Singer, “Improved boosting using confidence-rated predictions,” *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999. [Online]. Available: citeseer.ist.psu.edu/schapire99improved.html

[6] B. Rasolzadeh, L. Petersson, and N. Pettersson, “Response binning: Improved weak classifiers for boosting,” in *IEEE Intelligent Vehicles Symposium (IV2006)*, June 2006.

[7] N. Pettersson and L. Petersson, “Online stereo calibration using fpgas,” in *Proc. of IEEE Intelligent Vehicles Symposium, Las Vegas, USA*, 2005.

[8] “Oprofile,” 10th December 2007. [Online]. Available: http://oprofile.sourceforge.net

[9] G. Overett and L. Petersson, “Improved response modelling on weak classifiers for boosting,” in *IEEE International Conference on Robotics and Automation*, April 2007.