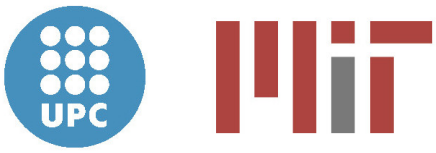


Final degree project

DetectMe: object detection on the iPhone

Author:
Josep Marc Mingot

Supervisor:
Antonio Torralba

December 9, 2013

Contents

1 Introduction

DetectMe is an iOS app that allows users to train and execute object detectors directly in their mobile device. To train a detector the users just need to take pictures of the target object label the image with the object position. Then they can execute the detector to detect objects in real time. The detectors can be shared between users and downloaded to analyze later on. Furthermore the mobile device can also broadcast the detections and the images in real time so the mobile can be used as the visual system for a great variety of applications. DetectMe is the first time a HOG object detector is totally implemented on an iOS device.

Many different organizations have already showed their interest in the project. The app can have great number of applications but the first organizations interested have focused on applications on robotics, healthcare patient tracking, visual impaired supporting systems and surveillance tracking.

This DetectMe project have two goals. On one hand facilitate all those interested organizations the implementation of the vision systems. Since the first days of the project we had in mind specifically people working in robotics. Many times they need to build from scratch the vision system of the robots. DetectMe will let them plug and play an iOS device that abstracts all the complexities of object recognition and let them directly deal with the detections. On the other hand, the project also pretends to explore how different users train a detector. As we will see, one of the most important parameters for the performance of an object detector is the training set used (number of images, size of the annotations, scene composition,...). It is not clear what is the optimal way to construct such data set. Our approach is to crowdsource the input of different training sets in order to analyze which ones perform better and why. Unfortunately this part is still under development and will be further discussed in the Future Work section.

Finally, the purpose of this document is also two fold. In first place to document the theory and implementation of the app for future developers of the project. Sections 2 and 3 deal with this part. The approach of this sections have in mind a target reader with a developer background and some notions of machine learning with no previous experience on computer vision. In second place, section 4 focuses on analyzing different ways to train a detector for some common tasks. These parts pretends to quantify the difference on performance of the different ways of training and give the user some intuition on the parameters that play a role on the training of an object detector. Sections 5 and 6 present what has been done for object detection on mobile devices and what are the next steps of the project. Finally, we sum up with the main conclusions of the project on section 7.

2 Theory

2.1 Some definitions

Object detection is a classification problem. Given a set of images containing annotations of where the object is in each of them, we want to use them to train a classifier (also known as *detector*). This process is known as *training* the detector and the initial set of images are called the *training set*. Then, given a non-annotated image, the detector has to decide if the image contains or not the object and, if it does, where is located. This process is known as *execution* or *testing*.

An image annotation is the pair of a *label* designing the object name or *class* and a *bounding box* localizing it inside the image. Bounding boxes of the training set are referred as *ground truth* bounding boxes. Images can have multiple annotations of different objects. When training for detecting a specific object, we just care about that object class ignoring the other ones.

To set up the classification task we need **examples** with **features** and **labels** to train the classifier or detector. We will now explore how those concepts apply in the particular task of object detection.

2.2 Examples: Image Patches

We have a training set of images with object annotations. We need turn them into positive and negative examples with features that our detector can learn from. The main idea is to use image *patches*. Each bounding box surrounding the object in an image (ground truth bounding box) defines a patch of a specific size inside an image. We use all the patches as the positive instances for the classifier. But what about the negatives ones? One option is to use the same image with the areas that do not contain the object. Another one is to use new images that do not contain the object and take patches randomly from them. So this whole set of patches constitute the examples we will give to the detector.

Those examples need features to learn from. The first (naive) approach for the features would be to simply take the value of the pixels inside the patch. So a patch of $n \times m$ pixels would generate an example of $n \cdot m$ features (assuming for simplicity grey scale images). This approach arises two different kind of problems. The first one is about the **quality** of features: are the pixels capturing meaningful information about their patch? The second one is about the **quantity**: what happens if we have patches of different sizes giving us different number of features? The next 2 sections will present possible answers to this questions.

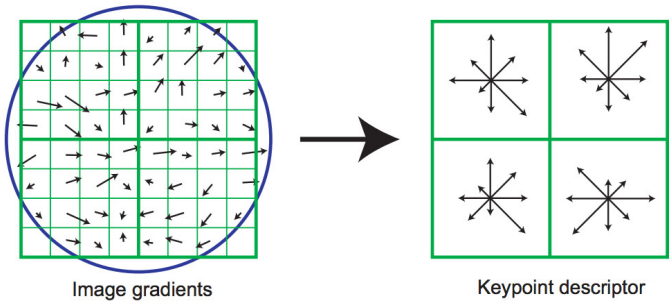


Figure 1: Image gradients combined into cells (4×4) to form histogram of gradients. The histogram is constructed taking into account the gradients on the cell weighted by their norm. Image credit to David Lowe [10].

2.3 Features I: Histogram of Oriented Gradients (HOG)

Image pixels has been our first approach for the example features but it has very important drawbacks: pixels are not robust to light changes and very sensitive to small variations of the image. We need more robust features. Some state of the art features for object detection are known as Histogram of Oriented Gradients (HOG, Dalal and Triggs [3]). We will just give a brief review on them to remark some of their important parameters that we will be using later on with our mobile detector.

HOG main idea is to gain robustness using gradients (instead of pixels) and group them into histograms of their angle as seen in the figure 1. So given an image, the first step is to compute the gradients on it by applying linear filters (exact implementation details in the section 3.3.1). Of the 3 channels of a RGB image, just the strongest gradient is taken into account. Then we group together some of the gradients into what is called *cells*. The size of the cells varies depending on the application but typical values are 6×6 or 8×8 pixels. Each of those gradients is put in a different histogram bin depending on its angle. Its contribution to the count in the bin is weighted by the gradient norm. Furthermore, there are two histograms computed simultaneously: ones that consider the gradient with orientation (so they angle is inside $[0, 360]$) and another one that considers the oriented gradient (now the angle is inside $[0, 180]$). As suggested in [3], the histograms are splitted in bins of $20°$ giving a total of 9 bins for the unoriented and 18 bins for the oriented histograms. So each of the cells (groups of pixels)

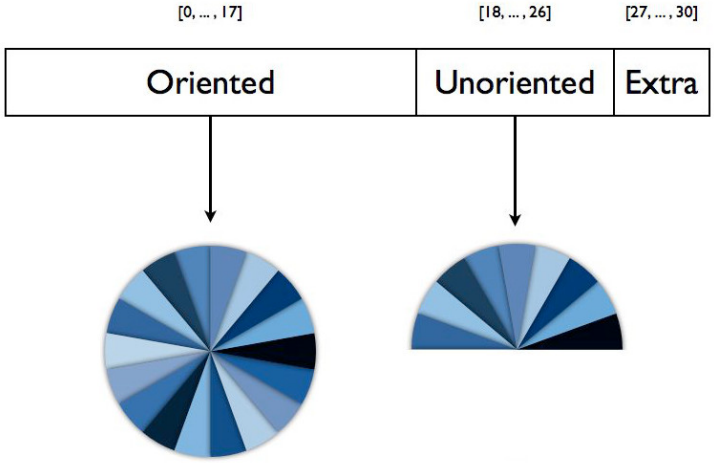


Figure 2: How the HOG features are stored in the features vector. First 18 correspond to the oriented gradients (contrast sensitive). Next 9 correspond to unoriented gradients (contrast insensitive) where the angle of the vector is just considered between 0° and 180°. Last 4 features capture gradient energy in the cell. Felzenszwalb implementation do not store the direct values but PCA projections.

will contain the set of the weighted counts[1] on the different histograms bins giving us 27 features. Felzenszwalb et al. [7] improves the HOG features in some additional ways. A part from differentiating between oriented and non oriented (Dalal initial work just take on of them), those features are reduced and compacted using PCA. Furthermore, the PCA space is decomposed in some simpler linear spaces to achieve fastest hog features. Additionally, the features contain a set of 4 new features giving us a total number of 31 features per cell. See their work [7] for more details.

To sum up, given a patch of $n \times m$ pixels, converting the pixels to the HOG space will give us a number of features of:

$$\frac{n}{cell\_size} \cdot \frac{m}{cell\_size} \cdot 31$$

corresponding to the histogram values on each cell.

2.4 Features II: Pyramid

Different sizes of ground truth bounding boxes can give us different number of HOG features per patch. So does having different image resolution.

[1]The actual implementation of the HOG does some additional tricks such as combine and average contingent histograms cells (into what is called *blocks*) to reduce the aliasing.

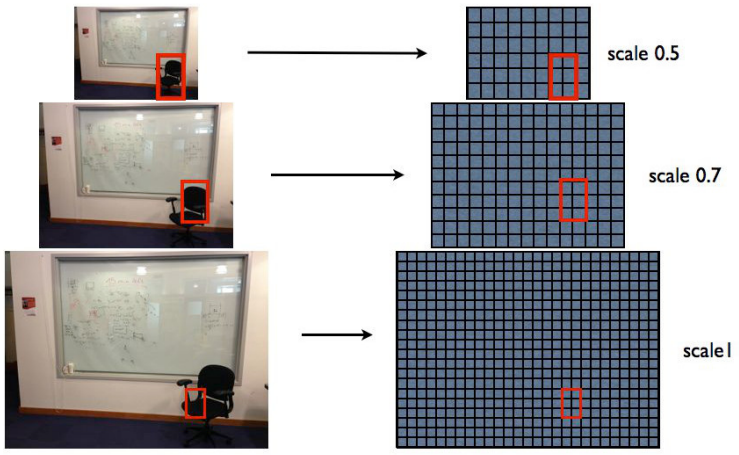


Figure 3: Pyramid construction. By scaling each image, the same template size (in this case 3×2) is able to detect different size of objects.

Furthermore how do we make our detector robust to detect different sizes of the same object? *Pyramids* give a possible solution to all those questions.

Given an image, the *pyramid* for that image is the set of images obtained by resizing (usually downsizing) the initial image a defined number of times. For example, given an image of 480×360 a possible pyramid associated to the scales $1, 1/2, 1/3$ would be formed by the original image, one scaled $1/2$ (240×180) and the other scaled $1/3$ (160×120).

Now let's define a fixed size of HOG cells (not thinking in pixels any more) for our patches. This specific size is known as the *template size*. By running the detector on the different images of the pyramid it will allow us explore different sizes of objects while maintaining the same template size. Now we just need to know how to put that together to deal with different sizes for ground truth bounding boxes.

2.5 Labels: the patch contains the image

We want to produce examples with the same number of features given a training set of images with heterogenous sizes of bounding boxes. We have seen that the pyramids lets us explore different object sizes by maintaining the same template size. To generate homogenous sizes of features, using the above fixed template size, we run the detector on all the levels of the pyramid a let it generate detections in the form of bounding boxes. All those bounding boxes have associated the same template size (and so the same number of features) for different levels of the pyramid. We then compute

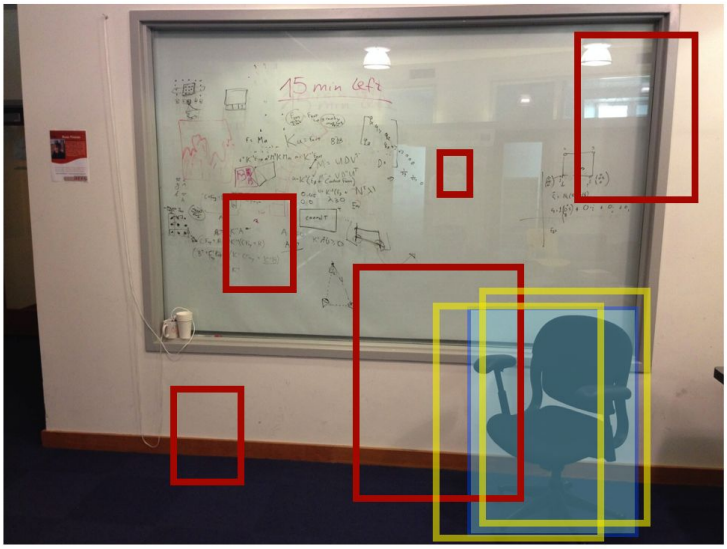


Figure 4: Annotated input image for training the detector. In blue, the ground truth bounding box (manually labeled). In red and green the detections produced by the detector. We consider the patch as a positive example (green) when it overlaps with the ground truth more than a certain percentage of the area. Otherwise the detection is considered a negative (red).

the overlapping area of those detections with the ground truth bounding boxes and, depending on that overlapping and some thresholds, we decide to consider that bounding box as positive or negative.

So we are able to transform a set of annotated images into a training set that a classifier can learn from. We will now explore how does the detector produce detections given a bounding box.

2.6 How does a detector detect?

Once the detector is trained, i.e. it has learned a set of weights w associated to each of the features it is able to detect the object in the image. To do so, given an image, we first compute its HOG features for different levels of the pyramid. Then a sliding window with the template size will be sliding over all possible position, containing different HOG features, x. For each of this positions, we will compute the score of that bounding box with the dot product $w \cdot x$. The higher the score the more confidence we have that this is actually the object we are looking for. Usually we consider a detection when it scores above a fixed *threshold*.

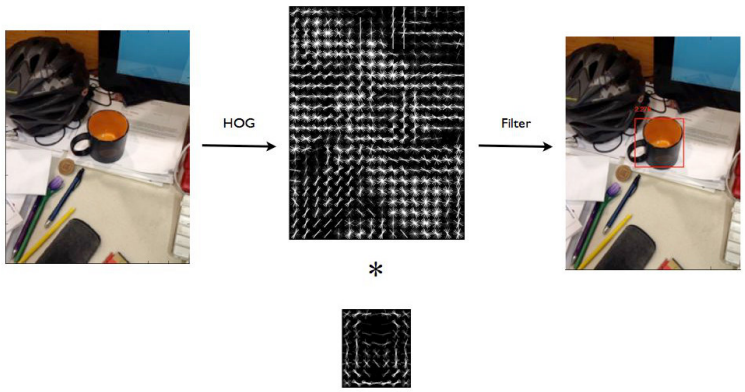


Figure 5: Overview of the detection: input image is converted to the HOG space where is convoluted with the detector weights. This give a score on each possible window. The highest score above a given threshold is selected and displayed on the screen.

We have presented here a simplified version of how and object detector works and the general definitions about it. All this information will be needed when explaining in detail the actual implementation of the detector on the iPhone[2].

To end this section, we explore some important metrics that allows us to measure the performance of a detector. We will make use of this metrics when evaluating the decision for different parameters of the detector implementation.

2.7 Performance metrics

The output of a detector is a set of bounding boxes over an image each one with an associated score. Some of them actually correspond with the object we wanted to detect, *true positives (TP)*. Other boxes are telling us that there is a detection where there actually there is not the object, *false positives (FP)*. If we miss scoring high a window that actually contain the object, we call it a *false negative (FN)* while if we score low not having the object it is called *true negative (TN)*. All those terms let us define two important concepts to evaluate a detector:

$$precision = \frac{TP}{TP + FP}$$

[2]Many times we will use iPhone as a specific device although it can be switched by iPad or iPod.

$$recall = \frac{TP}{TP + FN}$$

However, this both concepts are very tight related with the detection threshold: for different thresholds we will obtain different values for precision and recall in different ways from detector to detector. To capture the detector performance with independence of the threshold used we used what are known as *Precision Recall curves*. This curves contain the precision and recall in the Y and X axis respectively. The graph is obtained by sweeping over all possible threshold values(in practice, from -100 to 100 suffices). For each threshold value, we compute the precision and recall obtained over the test set and display the point in the curves. The more area under the curve, the better a detector is performing.

This curves let us compare detectors and decide the most appropriate threshold to use base on our needs (e.g. high precision or high recall). To compare detectors it is also used the *average precision AP* over the threshold sweeping. The AP captures in one number the performance of the detector.

In the figure 6 we present the P/R curves for the Deformable Part Model, DPM [7] over the Pascal Visual Object Classes (VOC) challenge [6]. For its different configurations, the curves are presented as well as the AP obtained. This detector is considered the state of the art for object detection. In section 3.2 we will use this detector in its simplest form (a single root filter) to compare it with the performance of the DetectMe. The DPM main idea is to use parts detectors to construct an object detector. For example, a car can be seen as a sum of the wheels part, the front part, the windows part,... In our performance analyses we will only compare it with the DPM of a single part.

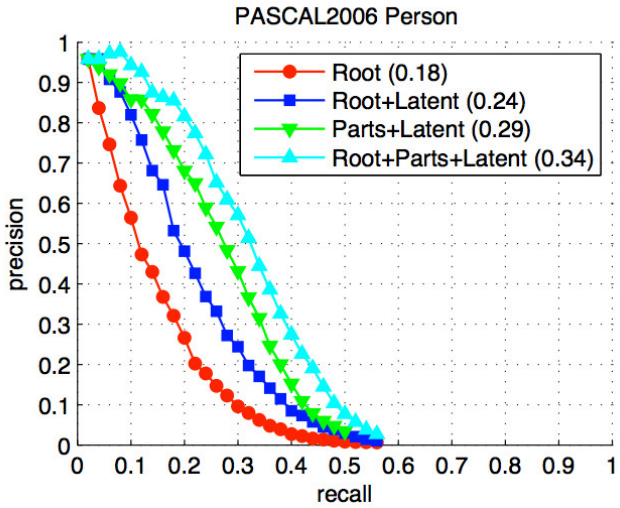


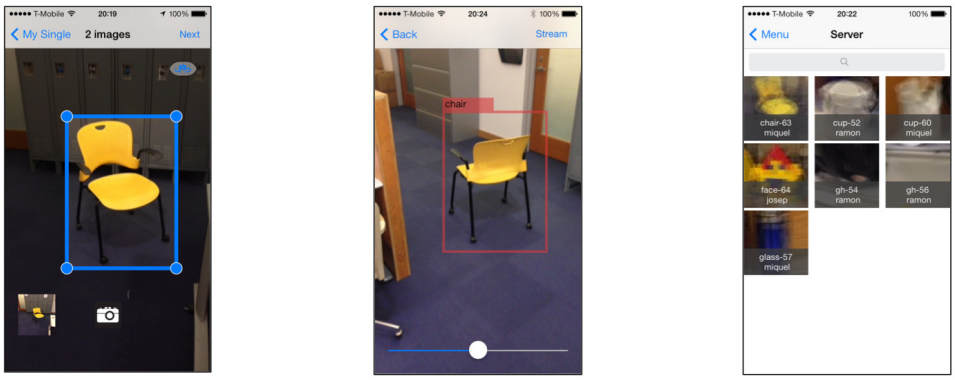
Figure 6: Example of P/R curves with different configurations of the DPM. In parenthesis the AP for each configuration.

3 Implementation

In the previous section we have covered the general concepts and ideas about how does object detectors generally work. In this section we analyze in deep the detector implemented for the DetectMe app for mobile devices.

The implementation has 3 different parts. On one hand, the **training process** by which we learn the classifier parameters from the training set of annotated images. On the other, the **execution process** responsible of making detections given a non-annotated image. As it was described in the previous section both processes are very related: for training, we execute the detector to produce candidate examples to train the classifier. There is another unrelated part in charge of **sharing** the information with the outside: the possibility to share detectors between users and to broadcast the detections in real time. Figure 7 show screenshots of the views implementing this 3 parts.

We begin exploring the training process treating the execution or detection process as a black box. In the next section, we analyze the execution part. In both section we sum up presenting the analysis of performance and show how we have balanced the trade-off speed-accuracy. We end up analyzing the technical implemention of the sharing part over a server and what technology has made possible transmitting real time detections over the network.



(a) Train (b) Execute (c) Share

Figure 7: Screenshots of the DetectMe app with 3 of its main views.

3.1 Training

The training process of the detector can be divided into 4 subparts: initialization of buffers and variables, hard-negative mining loop, feature extraction and SVM training. They all are summarized in the algorithm 1. The following subsections explore each of this subprocesses. At the end, we give some analysis on the performance of the detector in terms of accuracy and speed of training.

Algorithm 1 Training algorithm overview

Initialization
while $diff > STOP\_CRITERIA$ or $i < MAX\_ITER$ **do**
  $i \leftarrow i + 1$
  $[positives, negatives] \leftarrow$ Generate_Examples()
  $[w_i, SV] \leftarrow$ Train_SVM($positives$, $negatives$)
  $diff \leftarrow \|w_i - w_{i-1}\|$
end while

3.1.1 Initialization

The training process begins setting the template size given all the annotations in the training set. This determines the number of features of each training example. It then allocates the necessary memory to store the weights and the training set. Finally, it also allocates the necessary memory to allocate the HOG pyramid for the subsequent passes over the training images.

1. **Template size**. The template size refers to the size or number of

HOG cells that the training patch contains. We are using HOG cells of 6×6 pixels (as suggested in [3]). For example, a template size of 8×6 HOG cells has a corresponding 48×36 pixels image patch. To set the size of the template, we begin calculating the average height and width of all the ground truth bounding boxes to get the average aspect ratio. The sizes of those boxes are normalized to the image size so they belong to $[0, 1]$. We have fixed the maximum template size to 8 cells. This size was reasonable with the results in the literature([3], [7]) and gave us good processing times (training and execution). To fix the other size we use the average aspect ratio. For example, an average aspect ratio of $10 : 16$ would gives us a template size of 5×8. It is important to notice that this patch of 5×8 HOG cells will generate $5 \times 8 \times 31$ features for the learning algorithm (see 3.3.1 for more details). The template size is stored in the 3 dimensional array: $[t\_size_x, t\_size_y, t\_size_{HOG}]$. Notice that this configuration limits the number of features per example to $8 \cdot 8 \cdot 31 = 1984$.

2. **Weights**. Once the template size is fixed we allocate the necessary memory to store the weights of the SVM. This will be an array of $t\_size_x \cdot t\_size_y \cdot t\_size_{HOG}$ elements. Weights are stored as single precision float values (4 bytes). We initialized the detector weights to 1. This will produce very high scores on each bounding box the first time it runs (in the next section we will explain why this is important for us).

3. **Training features and labels**. All the training set will be preallocated in memory as well. This imposes high constraints and limitations on the number of examples and sizes. In the template size we already fixed the maximum size to 8 cells, giving a maximum number of 1984 features per example. We have also restricted the number of examples with the following 2 constraints:

   (a) Maximum number of training images per detector set to 50.

   (b) Maximum number of negative examples, or *quota*, (patches not containing ground truth) limited to 100 per image. This limitation lets us reduce the initial allocation of memory, speeds the process and helps keeping balance in the classes. As seen in the table 1, it does not harm the learning process because in each training iteration[3] the algorithm will take the highest ranked negatives, so if one negative was not able to enter in one iteration the learning algorithm will capture it in the following ones. After the first iterations, the number of boxes detected per image

[3]We are training the detector in an iterative way called *hard-negative mining*. The next section will explain the process in detail.

Quota	Time Learning (s)	AP	Buffer Size (MB)
50	51,32	0,15	99,2
100	79,02	0,23	119,04
200	105,81	0,22	158,72
400	136,46	0,24	238,08
800	135,51	0,24	396,8
1600	189,53	0,24	714,24

Table 1: Analysis of how does the maximum number of negative examples allowed per image (*quota*) affect to the training performance. The Average Precision (AP) remains constant while the learning time and buffer size increase with the quota.

was much less than the *quota* so all the negative examples were captured by the algorithm. This constraint let us highly reduce the amount of memory and time necessary to train the SVM.

Taking into account all the constraints, in the worst case scenario we are statically allocating 120 MB (12% of available RAM). Apple encourages to do a more dynamic allocation of resources, just reserving memory as we need it. However we find that this resulted in a simpler and faster approach. We are working in optimizing it for to consider a more dynamic allocation.

4. **HOG Pyramid**. In each iteration of the algorithm, we will be running the detector over the training images to get bounding boxes. Once calculated the HOG pyramid for each training image in the first iteration, we store it to be used in the following iterations. This resulted in a considerable speed up of the learning process at the expense of an additional memory allocation of 40 MB at most.

3.1.2 Training Loop

The learning algorithm does multiple passes over the training set of annotated images. Doing this, it allows the detector to be retrained multiple times with examples that missed the target on the previous runs. This *hard* examples are known as *hard-negatives* and this iterative process of learning is known as *hard-negative mining*.

In each loop iteration, the algorithm: (1) generates training examples (running the detector over the training images), (2) trains the SVM with them. So, in each iteration i of the loop the detector will be running (with weights w_i) over all the training images. After the training, the detector will be updated with the new weights, w_{i+1}, for the next iteration. The loop will

stop when $\|w_{i+1} - w_i\| < threshold$ (with a *threshold* of 0.01) or we reach a maximum number of iterations (set to 10).

It is important to notice that in the first iteration of the algorithm, the HOG pyramids are computed and stored in the previously reserved memory. The subsequent passes over the images will make use of those pyramids skipping the HOG computation. It is also worth mentioning that the first iteration runs the detector with all weights set to 1. This produces all possible bounding boxes to be considered as detections.

3.1.3 Generate examples

The algorithm begins by extracting examples from the annotated training images. Those examples will be all grouped to construct the training set for the learning algorithm.

For each training image, the detector is executed giving us detections in form of bounding boxes that are sorted according to their score. As already said, when the detector runs in the first iteration over an image with all its weights initialized to 1, it returns all possible bounding boxes. We use those detections as the training examples. For each of those bounding boxes, we consider it a positive example if it overlaps more than 70% with the ground truth and a negative if it does less than 50% (and the *quota* is not full). With that we construct the training set that the SVM will train on.

The iOS 7 devices have at least 2 cores (see appendix A for more details). Parallel process in iOS are controlled throughout the Grand Central Dispatch (GCD) which let us define queues of processes to be executed in parallel. In particular, the detector over each training image is run in parallel as well as the detections over the different levels of the HOG pyramid. All the training examples are added to the training set buffer with the corresponding label. An important trick worth to mention is how to handle bounding boxes with very distinct aspect ratio in different training images. This produces a *fake* average aspect ratio. This issue is detected because after running the detector in the first iteration no positive bounding boxes are detected. When this happens, the solution has been to *unify* the sizes of the bounding boxes getting the maximum height and maximum width for all of them.

3.1.4 SVM Training

We have used the linear SVM from the OpenCV [1] toolset. OpenCV adapted all its libraries for the iOS platform on the 2011. The SVM that OpenCV uses is based on LibSVM [2]. The SVM is trained with a regular-

ization parameter C of 0.02 as suggested in [7]. After the SVM is trained, we extract and store the weights, the bias term and support vectors (SV). The SV are then added to the training set for the next round. This procedure was also suggested in [7] to boost the hard-negative mining.

3.1.5 Inheritance of detectors

As we have mentioned, DetectMe lets you share detectors with other users (technical details in the section 3.5). Once a user trains a detector, she can make it public and other users can download it and use it.

We have gone one step further and let the users retrain the downloaded detectors. To do so our first approach was to consider retraining with all the previous images (as they are all stored in the server). This option was quickly seen as not scalable: as the inheritance tree of a detector grows, more and more images need to be downloaded to train a son.

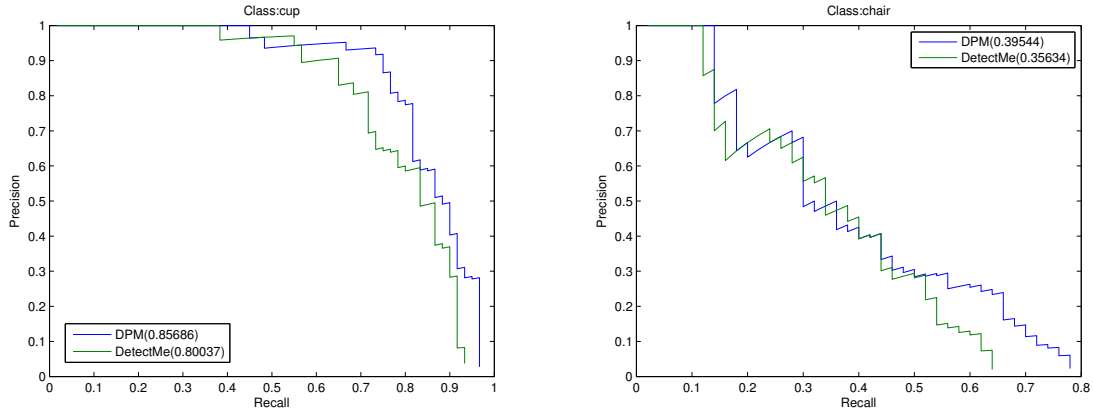
The solution was to just transmit the final support vectors (SV) of the detectors. The SV captures the essence of the learning done over a training set. They correspond to those examples more important and responsible for setting the boundaries of the decision detection - no detection. When a detector from another user is downloaded, so are their SV. To retrain it, the user has to provide new annotated images. This will produce the detector to take the essence of the parent and particularize with the new images.

To train the detectors with the old SV, we consider the SV as training examples in each iteration of the hard-negative mining. That way we force that the detector to consider the SV examples with the others hard-negatives.

3.2 Training Performance

We evaluate the performance of our detector in terms of accuracy and speed. For the accuracy, we compare it with the DPM. For the speed we give some and profile the process of training to show where the time is spend during the training. For all the test conducted we have tried to simulate the normal conditions under we expect DetectMe to work (in terms of number of training images, difference between them, clusters of positives and lighting conditions).

It is important to notice that we have limited the capabilities of the DPM in order to make the comparison. Our intention was to implement the simplest form of the DPM inside the mobile device. For this reason, the DPM we are using to compare with is only considering one component with a single root filter. We are using warped positives and random negatives with 4



(a) P/R for cup set

(b) P/R for chair set

Figure 8: Precision Recall curves for the sets cup and chair.

iterations. We also removed the flipped-positive examples from the DPM.

3.2.1 Accuracy comparison

We compare the accuracy of our detector with the DPM described above. To this end we have created test sets for 2 different classes: cup and chair. Each of this classes contains from 2 to 5 different clusters with chairs being a more difficult set because of the lowest symmetries and because it includes classes of chairs never seen before. The training and test set contain images corresponding to very similar scenes (same indoors with same lightening conditions). We have trained the detector with 20 images and evaluate the results in a test set of 60 images per each class. Figure 8 shows the results of those evaluations. DPM is still performing slightly better than our detector. The main reasons for this differences are:

- DPM is using more iterations (for the SVM training, not to be considered with the hard negative mining iterations) with a lower stop criteria.
- DPM is using all possible negative examples (not using *quota*). Although we have already showed the lower impact of this decision, it slightly contributes to its better score.
- DPM is using 42 scales over 4 octaves (2 up and 2 down). We are using just 20 over 2 octaves.

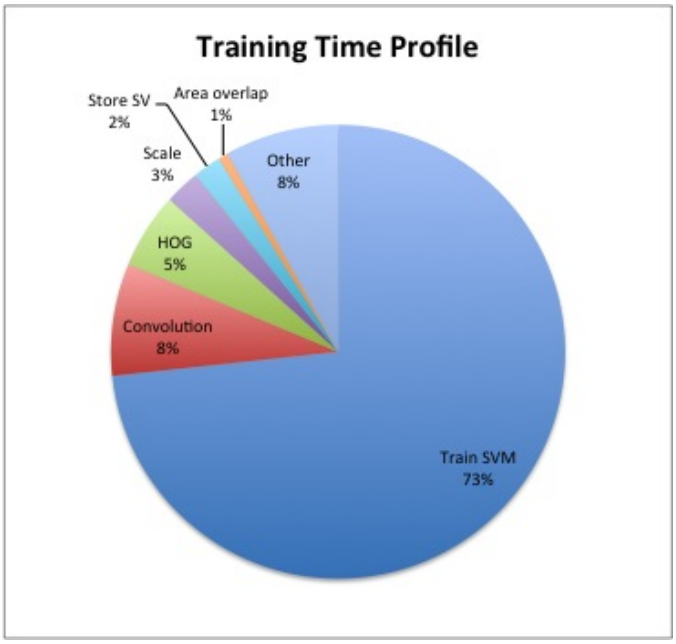


Figure 9: How the time spend training the detector is distributed. Training the SVM is the most resource consuming.

3.2.2 Speed

One of the most important of our detector is its speed. For a good mobile app experience we have to balance the trade-off speed-accuracy to the former. In terms of training, this corresponds to limiting the number of iterations, using not very low thresholds and, in general, keeping the images sizes pretty low. As shown in the figure 9 that profiles the training time, learning the SVM parameters is the most resource consuming. At the same time, the learning of the parameters is directly related with the number of features and number of examples.

We finally present the dependence of the detector training performance on the number of images used. Table 2 present the results on the time spend training and the AP. The trials were done on 2 different classes (chairs and cups) averaging the results over 5 different realizations. As we will examine in more detail in the section 4 the training set is one of the most important parameters on the performance of the detector. With DetectMe the user has the power to decide which and how many (with a limit of 50) images to use. It depends on her how much time and effort to spend on training each detector. Although each particular situation has a different number of images necessary to correctly train the detector, usually a good rule of thumb is to use between 20 and 30 images.

Number of Images	Time Learning (s)	AP
2	10,71	0,09
4	15,16	0,12
8	26,41	0,19
16	56,83	0,21
32	162,71	0,31

Table 2: Dependence of the detector training performance on the number of images.

3.3 Execution

The execution part of the algorithm is the responsible of, given an image, extracting detections from that image. Those detections take the form of bounding boxes. The main subprocesses of the execution part are to scale the images to construct the pyramid, extract the HOG features of each pyramid level and convolute the detector to score each possible window. Finally with all the scores it is important to apply some filters to just get the best windows that will be displayed as detections. Algorithm 2 summarizes the process. As with the training part, the following subsections explore all of those subprocesses.

Algorithm 2 Execution algorithm overview

function DETECT($image$)
  $image \leftarrow scale(image, initialScale)$
  $topDetections \leftarrow []$
  for all $scale$ in $[scales]$ **do**
    $imageLevel \leftarrow scale(image, scale)$
    $imageHOG \leftarrow hog(imageLevel)$
    $[boundingBoxes] \leftarrow convolve(imageHOG, detectorWeights)$
    $[topScaleDetections] \leftarrow filter([boundingBoxes])$
    $[topDetections].append([topScaleDetections])$
  end for
  return $[topDetections]$
end function

3.3.1 HOG

In order to run the detector on the image we first need to transform that image to the HOG Feature space. Each image is transformed taking the HOG extraction libraries from [7]. The gradients of the image are computed using the linear filter $(-1, 0, 1)$ for the x component and $(-1, 0, 1)^T$ for the y component. If an image has dimensions $w \times h$ pixels, then the transformed

Initial Scale	Resolution	FPS execution	Time learning (s)	AP
0,2	72x96	23	–	–
0,3	108x144	10	33,26	0,06
0,4	144x192	5	79,43	0,24
0,5	180x240	3,5	84,24	0,36
0,6	216x288	2,3	104,54	0,35

Table 3: Effects on execution (FPS) and training (Time learning, AP) of the initial scale chosen for the incoming images. Notice that for the scale of $0,2$ it was not possible to train due to the low resolution of the image and the disparity of the ground truth bounding boxes obtained at that resolution.

image in the HOG space will have $\frac{w}{cells} \times \frac{h}{cells} \times 31$ where $cells$ is the number of pixels per HOG cell (6 in our case).

One of the most important parts in terms of performance is the initial scale of the image. Images taken from the iPhone camera using the libraries `AVFoundation` for the medium quality preset extract images with size 480×360. For all the detection (in execution and when doing the hard-negative mining in training) we first scale the image. This initial scaling has very high impacts in the performance of the detector: in the speed on execution as well as the precision performance, as it can be seen in the table 3. This initial scaling is made to guarantee the real-time in detection. As shown in the image, it has to balance the trade-off speed-accuracy. For our detector, we decided to set a scaling of $0,4$ for execution (that gave us 5 FPS) and $0,5$ in training (i.e. in the detections performed during the training phase). By having a higher resolution while training, we are able to extract more meaningful examples to train with that later on benefit the robustness of our detector.

3.3.2 Pyramid

We want to be able to detect patches of different sizes. If we just take the above image features, just on fixed size of patch candidate to a detection. To allow multiple sizes, we scale the images several times for different values and compute the HOG features on each image to form the HOG pyramid. As shown in the picture 3 the trick of changing the image size lets us with the same template size detect an object at different scales.

We compute 10 pyramid levels over one lower octave, going from the initial size to half of it with 8 intermediate scales. So we are able to detect objects from the *natural* size they are labelled until double size. This applies for the 1st and 10th scale. The other 8 scales between are computed in a logarithmic fashion that models that we need more exploration on the high

resolution images:

$$\left\{1, \frac{1}{2^{1/10}}, \frac{1}{2^{2/10}}, \ldots, \frac{1}{2^{10/10}}\right\}$$

The number of pyramids to calculate also has a very important impact on the detection performance, as we will analyze in the following section.

An important improvement over the detection performance was achieved by dynamically limiting the number of scales to compute. Once an object is detected and for the subsequent images we just compute 1 scale lower and 2 scales higher from the detected scale. We choose 2 scales lower to counteract the effect of having more bounding boxes in high resolution scales. This resulted in a 24% increase of speed.

Finally and as explained above, all the images are also scaled to a factor 0.4. This initial scaling makes corresponding the template size on the base of the pyramid with a bounding box of coordinates $upperLeft = (0.3, 0.3)$ $lowerRight = (0.7, 0.7)$[4]. This box corresponds to the default (and suggested) size for annotating images. This initial scaling then is adapted so that the detections made correspond to reasonable sizes by just using a single octave for the pyramid level. It also speeds up the detections as the base of the pyramid has a lower resolution (144×192).

3.3.3 Convolution and filters

Once obtained the HOG Pyramid over an image, the detector scans all the levels and scores each window. Top scores are considered detections.

For each HOG level of the pyramid, we slide a window of with the template size and give the features inside, x, to the detector. The detector computes the dot product between its weights, w and the features present inside the sliding window x to give a score to that patch. Two different filters are then applied to all those patchest to select the final detections. If the passes the 2 filters, the box is considered a detection and send to be visualized.

The first filter is a detection score threshold. The detection threshold is controlled through a slider present in the execution screen (see figure (10)). This threshold is stored as a parameter of the detector and sets its sensibility to detections.

[4]Remember that the sizes are normalized by the image size. So for an image of 360×480 this initial patch is situated in the middle with dimensions of 144×192.

Figure 10: Screenshot of the real-time detection view with the threshold slider on the bottom.

The second filter is called *non maximum suppression*. After sorting them by decreasing score we make sure that we do not have overlapping boxes. If we do, we choose those with higher score.

3.4 Execution Performance

Most of the decisions taken with the DetectMe app went towards optimizing the performance of the detector execution. Fast real time detection is one of the key features of the app. A fast detection also helps speefing up the detector training as most of the training time is spend running the detector on the hard negative mining iterations. We present here some analysis of the times of execution as well as a time profiling. This later will gave us insights were the detector can be optimized. We finally will explain our attempts to make it faster and why they did not work.

The execution times are presented in figure 11 in function of the number of detectors running simultaneously. For 1 detector we obtain 5.5 FPS that decreases to 1.9 FPS for 16 detectors running simultaneously. This times have been obtained when no detection is done. As explained in previous section, for execution of 1 detector, we reduce the number of pyramid levels to explore when the object is detected. With this trick, when the object is detected we can achieve around 16 FPS giving the impression of real-time for the user. Although this trick is reproduced for multiple detectors running simultaneously, it does not improve the results for more than 2 detectors as it is less probable all of them belonging to the same pyramid level.

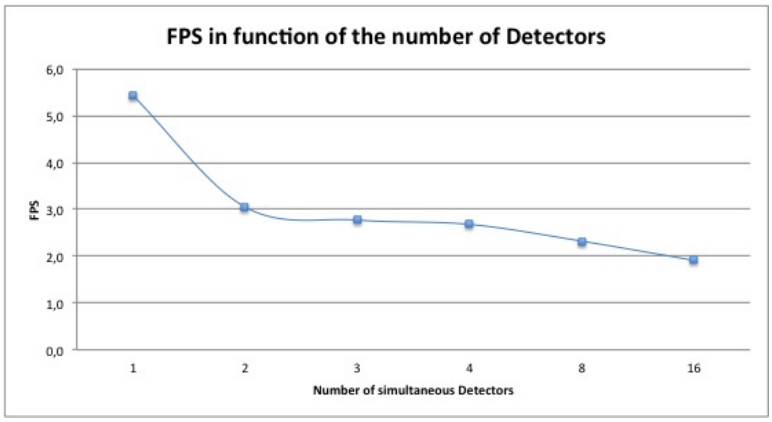


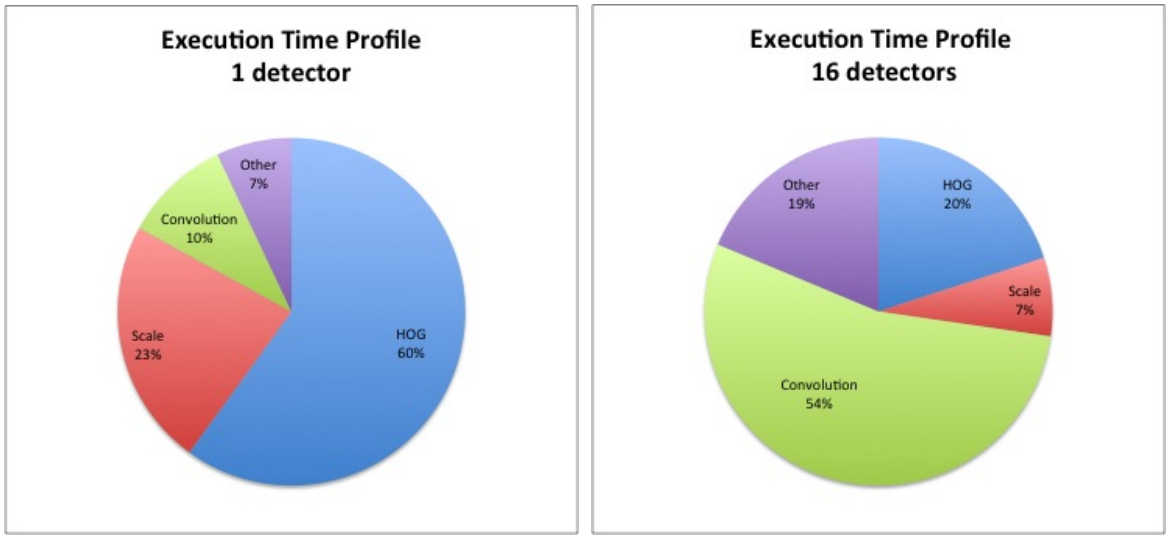
Figure 11: FPS in execution. Dependence on the number of detectors running simultaneously.

Figure 12 presents the profiling of the execution time for 1 detector and 16 detectors running simultaneously and tells us how is the detection time spent. For 1 detector, we see that most of the time is spend building the HOG pyramid: scaling and extracting HOG features. For 16 detectors though, convolution is getting most of the time.

This profiling guide our efforts to improve the current detector towards reducing the two biggest bottlenecks: HOG features and convolution time on multi detection. In first place, we already tried different strategies to improve the performance of the detector by reducing the HOG extraction time. As suggested in [5], we tried the nice idea to scale directly from the HOG features of the first scale. With this approach we could avoid all the image scales and their correspondent HOG extraction at the cost of a slightly reduction of accuracy. However this approach only presented significant improvement when the pyramid had many scales (e.g. 42 in the DPM). With our 10 scales, the improvements were almost imperceptible. Priscariu et al. [11] propose the implementation of the HOG extraction on the GPU. However they make use of the newly General Purpose GPU (GP/GPU, e.g. Nvidia Cuda) and the Apple devices have not yet implemented them. In second place, our future work will also focus on improving the convolution on multiple simultaneous detectors. Section 6 will go in more detail about our future improvement plans.

3.5 Communication

The last part of the DetectMe app is its sharing capabilities. The app lets you share the detectors (and the respective annotated images) among other users through a web server. This server also has its public API that lets you



(a) P/R for cup set

(b) P/R for chair set

Figure 12: Time profiling for the execution of the detectors.

download detectors from it or upload new ones. This lets you for example train your own detectors with Matlab to upload them afterwards. Finally the app has also the capability to broadcast the real time detections it is doing. This allows to visualize the images and detections directly in the browser or in Matlab.

3.5.1 Sharing detectors

One of the main goals of the project was to do a social network for detector sharing. Users would be able to train their own detectors with their images

Figure 13: Sharing capabilities of the DetectMe app. Sharing detectors let you get detectors form other users and download to retrain them on Matlan. Real-time broadcasting lets you receive the detections on real time on the browser or Matlab.

and share them. Then they would download detectors from others and rate them.

The purposes of this social network is two fold. On one hand, investigate how different ways of training affects the detector. The training set is one of the most determinant factors in the machine learning algorithms. For object detection in particular, it is not clear how different training sets (with images clustered in different ways) can affect the performance of the detector. Having a "crowd" trying unconsciously different ways to train one can highlight some good practices currently not being considered.

On the other, discover how does the rating of the user correlates with some established performance metrics of the detectors. All the metrics on detectors performance focuses on the precision and recall of the detectors, but none of them takes into account the execution time. For a real time detection engine, the speed of it can have a very high impact on the "perceived" performance. For instance, a detector not very precise but very fast would be perceived as "better" by the end users as one that is very precise but has to skip a lot of frames due to the processing times.

The share of detectors is possible thanks to a dedicated server (`http://detectme.csail.mit.edu/`) that allows the exchange of information between the devices and visualize detectors, training sets and real time detections on the browser. It also has a public API that allows any service to download detectors and the corresponding annotated images, update them or upload new ones. The detectors store many information about them that will allow us to analyze its performance. The most important information stored is: user ratings, support vectors, number of images, detector performance on the training set and a training log. The annotated images also store some useful information such as the GPS position or the 3D position of the device.

3.5.2 Broadcasting Real Time detections

The last important capability of the app is to begin a server inside the iPhone to broadcast the detections, as well as the images, in real time. This lets us capture what you are seeing with the device on the browser. It also open a stream communication on the device that lets any other service connect directly to it to get the feed of detections and images captured. In particular, we also developed a library for Matlab to capture this stream.

The purpose of this feature was to convert the device into a sensor for other systems such as robots. With this capabilities, people working with robots can forget about having to set up complex vision systems and just

plug the device (iPhone, iPad or iPod) and receive the detection through the browser, the API or directly into Matlab. Once there, it is easy to make the system take different decisions based on the detections. The transmission achieve rates of 15 FPS on local networks.

Apple does not make the transmission task easy as they do not want people using the usb cable to send this information. The Bluetooth interface was also not very interesting because of the short range. So our only option was to send the detections and images through wifi. Web sockets seem the clear option as they provide a full-duplex communication over TCP with very less overhead than HTTP providing lower latency. They allow the real time communication over TCP.

We use WebSockets as the underlying technology for the streaming on the device. More concretely we use the iOS library `BLWebSocketServer` to transform the device into a server. Once done, it begins listening on the port 9000 for incoming connections. When web socket client (e.g. the browser or Matlab) connects to it, the device begins broadcasting the images and the boxes captured. The web socket library being used requires the messages to be strings. For the detections, we encode them as JSON. With the images, we transform them into strings using the Base64 encoding to send.

The only restriction is that the device has to be in the local wi-fi in order to be reached as a server (through its IP). This was very convenient for our purposes as it also ensures the security of the communication by restricting it to the local network.

With the browser connection the synchronization is a little bit more complex as it has to be ensured that just a given user can access to their specific device broadcasting over the browser. To implement this session based restriction we make use of a central *node.js* server monitoring the clients and servers to make sure the matches of session (i.e. a specific user on the browser has to be matched with the corresponding user device). When a new device begins to broadcast it informs the *node.js* server that it is transmitting under a certain user session. At the same time, when a browser asks for visualizing the detections, the *node.js* server looks if its device is already broadcasting. If it is, it allows the direct connection browser-device. If it is not, it puts the browser in a waiting queue.

4 Usage Examples

The DetectMe user have a great range of different ways to train a detector for a given task. The main one is to construct the training set for the detec-

tor: number of images, scene composition, boxes size, ... All this parameters greatly affect the performance of the detector. We present here some experiments of the detector training for different tasks with the intention to give the user a general knowledge of different ways to train a detector and their differences.

We distinguish between 2 main tasks when training a detector: *specific* and *general*. When training a specific detector, we are interested in detecting a particular and concrete object. For example, when trying to detect my chair (among all the possible chairs and other objects) or when trying to detect the 10 dollar note among different notes. On the other hand, when training a general detector, we are interested in detecting a specific class with as much generalization as possible. For example, when trying to detect persons, cars or chairs in scenes.

When correctly trained, DetectMe can be used for both cases. However, HOG features and the general setup of the detector benefits the general case. In the following sections we will be able to compare the detector in this two different tasks.

4.1 Training a specific detector

In this first section, we are interested in detecting a specific object among all the others of the same class. For example, a given can in a fridge, my pair of shoes inside a shop or Ulysses book in a library.

The example we have selected to work with is detecting a specific card from a card deck of 52 cards. Concretely, our example task is making a detector for the 6D card. Cards give us nice properties as it is relatively easy to distinguish them from other objects and there are gradually and very quantifiable differences between them that would give us some notion of until what point a detector can be discriminative.

We run two scenarios of experiments. In the first one we want to see how a detector trained to detect one specific card scores each of the other cards. In the second scenario, we analyze different training sets to detect a specific card among the others.

All the experiments have been performed with similar presets in the training and test sets. Same backgrounds, positions/rotations of the object (i.e. no flips have been considered). The intention of this is to isolate what is intrinsic of the object or class we are trying to learn. Making it robust to backgrounds or rotations can be later achieved by adding more images taking into account different scenes and positions.

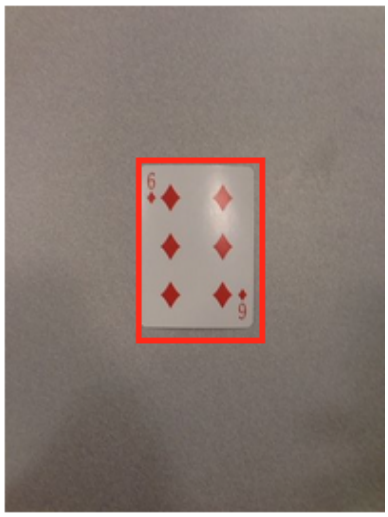


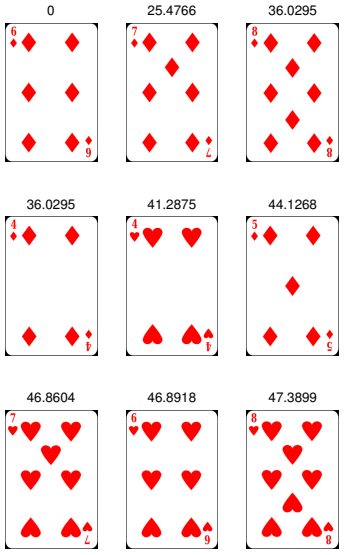
Figure 14: Image on the training set for the 6D detector.

4.1.1 Scenario 1: similarity search

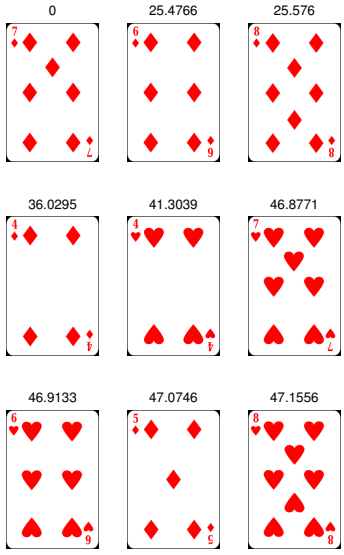
We want to see how a detector scores each of the 52 images when it is just trained with a single target card. The training set consists of 10 images of a single card image on a clean background (as the example of figure 14). Then each of the 52 cards are scored with the previously described detector. We tried this experiment with different card detectors. We show here the results of the top scores for 6D, 7D and 7H. We have selected this 3 examples because, regardless their apparently similarity, they show interesting properties that affect the detector performance.

In order to compare and justify the scores we also calculate the *real* distances between cards. We have taken synthetic images of the cards and calculated the distance (using the Frobenius norm) with all others. In figure 15 we show the top 8 closest cards obtained to 6D, 7D and 7H and their respective distances. The distances in the diamonds groups are lower than the hearts one, so in theory this will make it more complicated to detect. On the other hand we can see that virtually, the closest neighbors to 6D and 7H seem to be at the same distance.

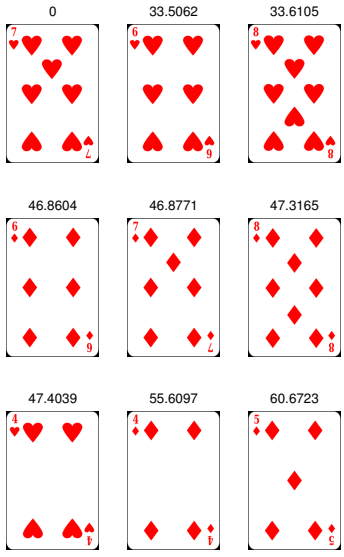
Figures 16, 17 and 18 show the top score results of the detectors scores for the 6D, 7D and 7H cards. While the 7H detector scores the 7H the highest one with a remarkably difference, 6D detector leaves the 6D card on the 7th position and 7D detector on the 2nd position. The mistakes done by the diamond detectors are in part due to the *distance* between the testing



(a) Distance to 6D.



(b) Distance to 7D.



(c) Distance to 7H.

Figure 15: Top 8 closest card to a target card using synthetic images. Distances computed as the Frobenius norm of the image matrices.

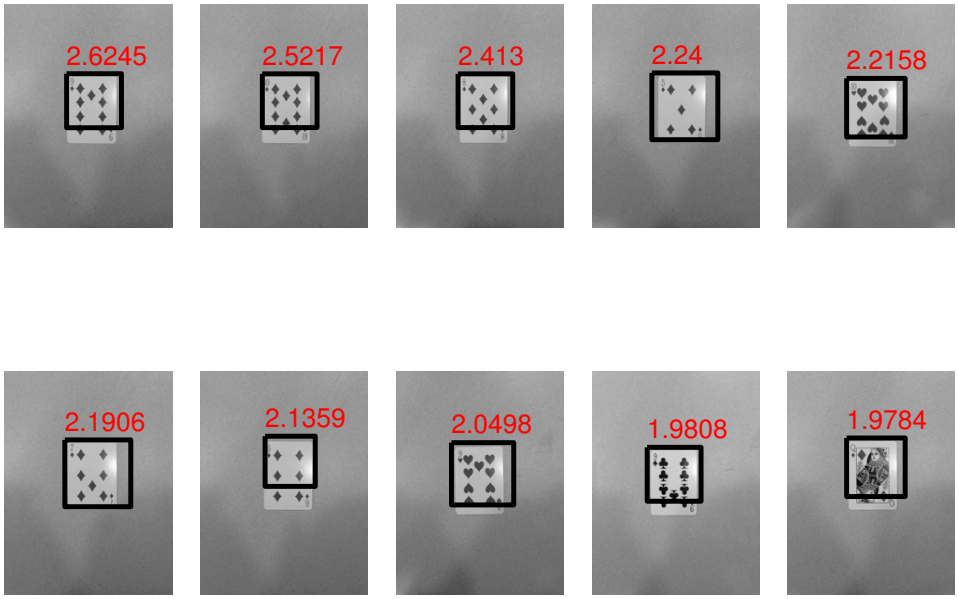


Figure 16: Top 10 scores for the 6D detector over the 52 cards. Due to 6D simetries it misses the top scores in benefit of similar cards.

objects. As we have seen, diamonds are intrinsically more difficult to differentiate which makes it more difficult for the detector as well. 7H has less symmetries on the object and on the number making it easier to distinguish from its peers. Furthermore, it is also important to notice how 6D can be seen as a portion of 9D (the highest score). In the middle of this 2 we have 7D that scores better than 6D but worse than 6H. This appreciation can not be seen in the synthetic example (figure 15) where 9D is not even in the top 8 closest neighbors. 7D misses 10D as a top score, but one more time, 10D actually contains 7D.

So, the detector takes into account other things a part from the actual distance of images. More precisely, if one image *contains* another one it will score much more higher although the actual distance is not as high. we can see this behavior clear with the high score that 9D gets.

In this initial example we have explored how the actual distances between images affect to the detector performance. For those objects with neighbors very similar to them, we will need more sophisticated training sets to train the detector to distinguish them. In the next subsection we will explore with more deep this later part.

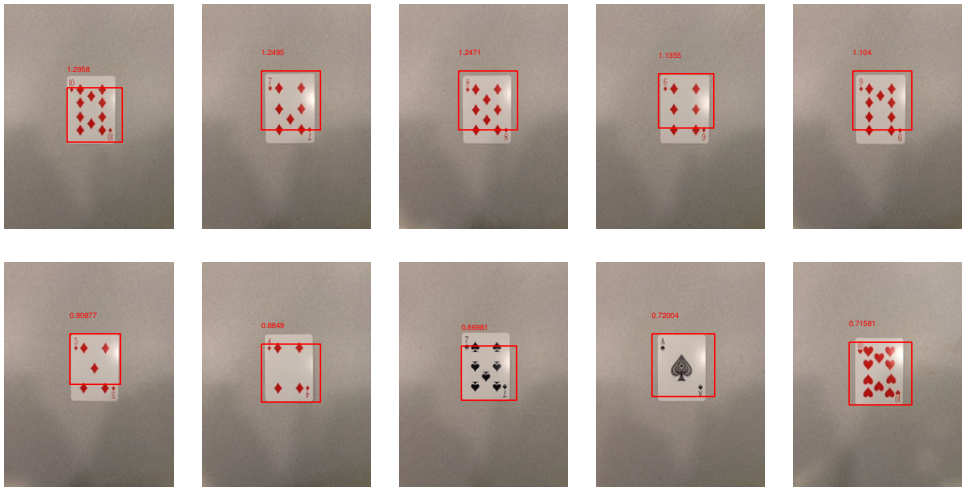


Figure 17: Top 10 scores for the 7D detector over the 52 cards. The highest score correspond to a false positive (7D). Notice though that 10D *contains* the image of 7D inside.

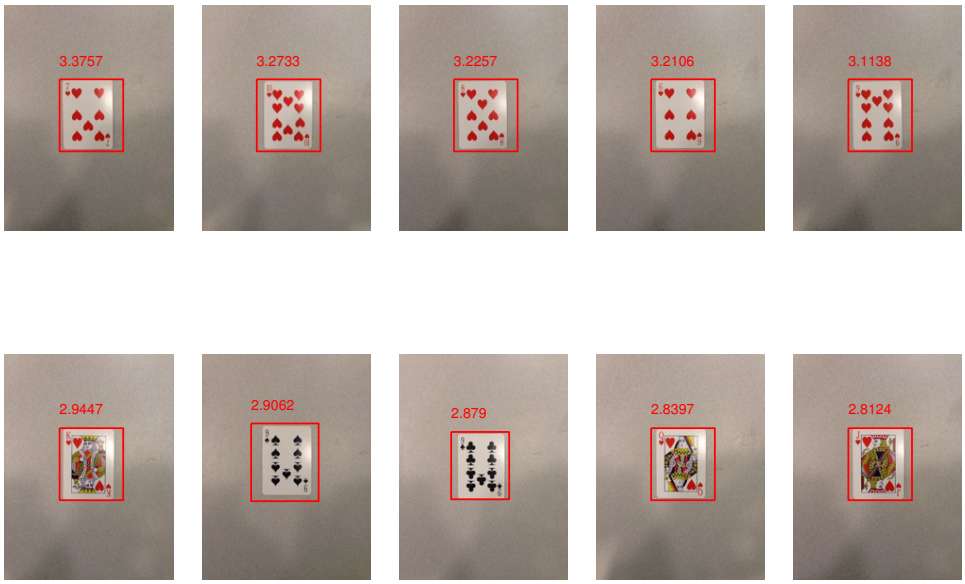


Figure 18: Top 10 scores for the 7H detector over the 52 cards.

4.1.2 Scenario 2: distinguish

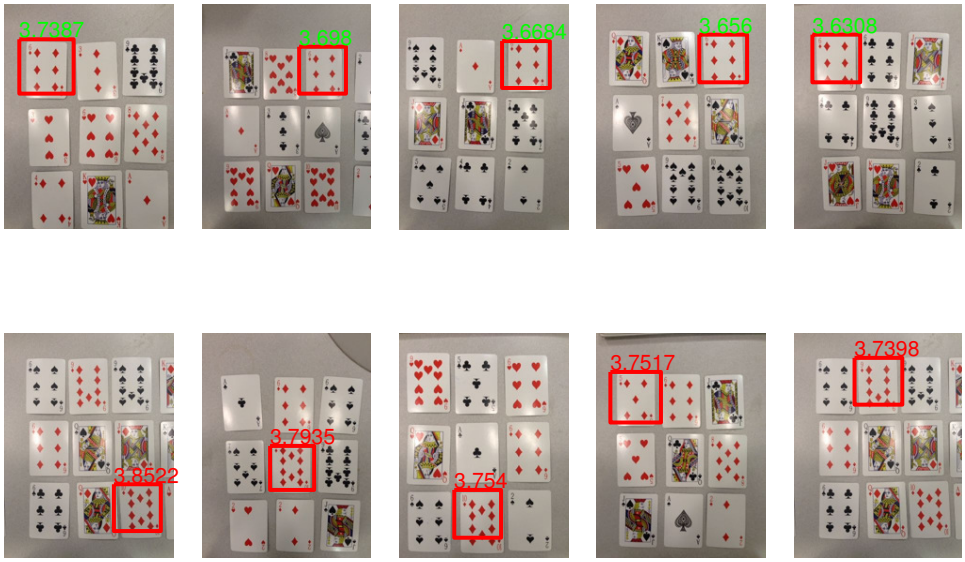
The intention now is to find a way to tell the detector how to distinguish between similar objects of the same class. So we will train it to take into account negative examples. For example, for the 6D detector and looking at the figure 16 we might be interested in explicitly telling to the detector that 9D and 10D are negative examples. We have selected the 6D detector for this task for being the most difficult one among the trials of the previous task.

The test set is constituted by 30 images of 3×3 randomly chosen cards. Our mission is to find the most simple training set that will give us a detector with higher performance. We will also explore how the different ways of training affects this performance.

Our first training set consists with just one single image 6D isolated as the one showed in the picture 14. Figure 19a presents the top 5 true positive detections and false positive detections. Figure 21 presents the precision recall curves. For this detector we have obtained an average precision of 0.51. Looking close to the figure 19a we observe that the top detections actually correspond to false positives (second row of images). Concretely the top false positives correspond to 10D, 5D and 9D. This results are coherent with the ones obtained in the previous section (figure 16). However 9D, the highest scored card in that experiment, appears now as the 5th top false negative. This is because we have trained our detector with just one image and the detector can be learning from specific image things such as light, reflections, position,... To get rid of those effect we trained again the detector but this time with 20 images of the 6D card alone. For this detector, we obtain an average precision of 0.49 more or less the same as the previous one. However the results nows in terms of false negative are more coherent with the ones obtained in the previous section. Figure 19b show the top false positive detections on where 9D gains positions.

The next training set shows to the detector those cards where it was scoring worse. In particular, we selected top 8 false positives from the previous trial and arrange them in a 3×3 grid with the 6D. We randomly arrange the grid and construct a training set consisting on 10 images permutations of that grid. This detector obtained an average precision of 0.65, a remarkable increase over the previous ones. As we could expect, inspecting the false positives on the figure 20a shows that now they correspond to other cards.

Previous results suggest our last experiment that is training another 3×3 grid now randomly sampled among all the cards constructing 10 different



(a) Detector with just a single image of the 6D.

(b) Detector trained with 20 images of the 6D card.

Figure 19: First row of each image contains the 5 top true positive detections and the second one the 5 top false positives. Detectors trained with just the 6D card on a clean background.

images for the training set. This last training set obtains the best results with an average precision of 0.92. Analyzing again the false positives on the figure 20b we can observe that the first one is actually a 6D but too big. The other ones correspond the the difficult cards 9D and 10D.

Figure 21 presents the PR curves of the 4 different training sets tried here. The single target image as a training set already obtains quite reasonable results. However, this single object detector produces an important number of false positives. This false positives are not decreased by training more equal images to get rid of illumination and position bias. By presenting on the training set the top false positives of the previous run, we gain a 30% boost on performance. This technique is highly encouraged for training a detector: increasing the training set with images where the detector fails. Finally the top performance is achieved when showing to the detector all the possible set of negative objects. Although this last approach obtains the highest AP, it is often no feasible in some real examples for not having access to all the negative examples (as detecting a specific can or a specific chair amont the others). With other applications, such as recognizing a specific note, it is possible to train the detector with the whole spectrum of negatives.

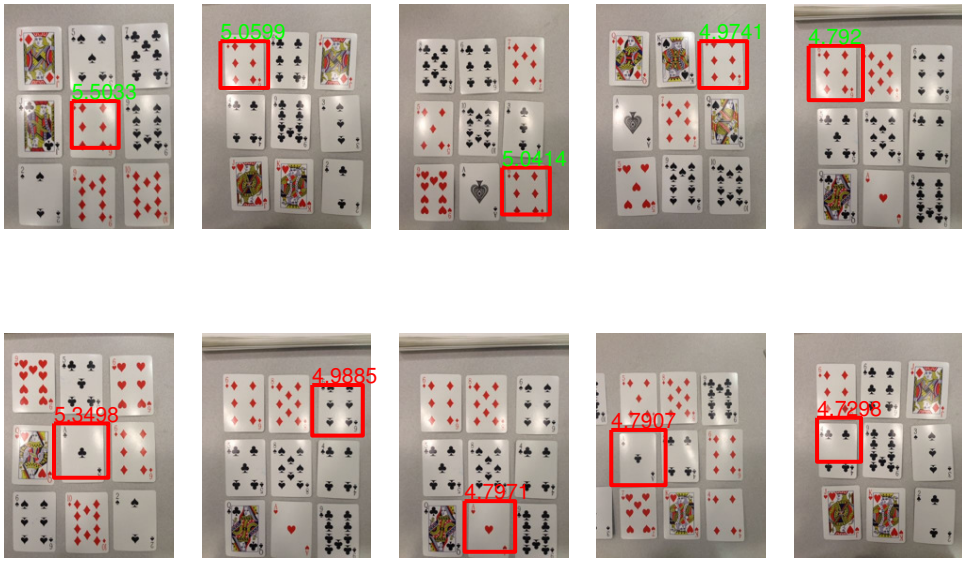
4.1.3 Conclusions

In this set of experiments we have analyzed different approaches available to the DetecteMe user for training a detector to detect a specific object within a class. The results have shown that training with a single image with the single object isolated produce remarkably good results. This false positives can be further reduced by specifically showing the detector what is doing wrong. When available, showing the whole spectrum of negatives outperforms showing just a few of the top scored ones.

4.2 Training general

In the previous set of examples we were trying to answer which is the simplest training set one can construct to train the detector to detect a specific object among other objects of the same class. We now focus in the problem counterpart: which is the simplest training set to train a detector to detect any object of a given class among a set of other objects from other classes. For example, detecting chairs, cups or person in scenes. We want to analyze how the detector generalizes its learning to other objects on the same class.

The following subsections analyze two different scenarios. On the first one, we do the experiment on the cards focusing on detecting the *cards class*. Due to the similarity between cards, we will see that this task is *easy* for



(a) Detector trained with a grid containing top 8 scored cards for the 6D detector.

(b) Detector trained with a grid containing random cards.

Figure 20: First row of each image contains the 5 top true positive detections and the second one the 5 top false positives. Detectors trained with 10 images of 3×3 grids of cards.

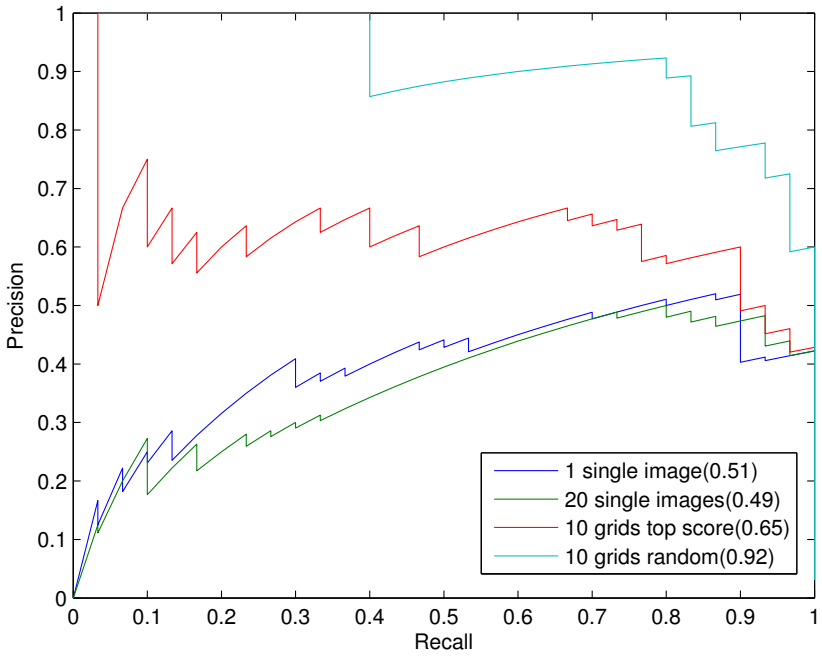


Figure 21

the detector and the highest performance is quickly achieved. On the second one, we analyze the same problem with a real application: detecting recipients of liquids in scenes.

4.2.1 Scenario 1: detect the cards class

Continuing with the cards, our first experiment will try to construct a general detector of cards. The experiment pretends to discover which is the simplest training set that let us detect any card among other objects and quantify how good the detection is. To simplify the problem we will just consider the spades suit as the *class* of objects. Second and third row of figure 22 present examples of the test set used in this example.

Our first approach to the problem is to see if just a single card with clean background was able to generalize to the others. For the first experiment we select the 4S card for being the closest to the centroid of spade cards. In the reality is very difficult to select an specific object by this procedure because very often we do not have access to all the possible objects of the class. Even when we do, computing image distances between non synthetic objects can be very tricky. However, the idea is to try to get the most representative and simpler object in the class. Figure 22 shows 5 random images

of the training set and detections on the test set. The training set consists on 20 images and the test set 30. The test set is done by selecting a random spades card and adding random objects. As shown in the figure 22 some of this objects such as other kind of cards are very similar in shape and size to the actual spade cards.

For our surprise this simple single card detector is able to obtain an average precision of 0.93. This precision can be easily pushed forward showing the top false positives on the training set. The reason why it is doing so well is because the spade class is very close between them and quite distinguishable from the other kind of cards and objects specially for the white border.

To compare results, we do the same experiment with the QS, the furthest card to the centroid. Figure 23 shows 5 random cards from the training set and top true positive and false positive detections. Figure 24 shows the precision-recall curves comparing both detectors. The QS detector obtains an average precision of 0.65, quite below the 0.93 obtained by the 4S. Although still quite surprising the generalization capability of the QS detector, it generates more false positive specially on the other kind of cards.

Finally we also try a detector with all the cards. Because the training set has 20 images, some cards were repeated. Figure 24 also contains this results. In particular we see that it is performing slightly worse than the 4S detector. This behavior will be explored in much detail in the next section because shows an important characteristic of the detectors when generalizing.

This first task shows us the importance of correctly selecting the training set for the generalization task and how selecting a non representative object of the class impact the detector performance. However, the cards class is an easy problem as the simple one card detector was able to get all the detections correct and it does not allow us to test other strategies to increase performance. For this reason, we present in the next section another experiment that will allow us to explore more interesting training sets to generalize a detector.

4.2.2 Scenario 2: detect the cup class

We change of class to the *recipients* class. Suppose we want to train the vision system for a robot to detect objects that can be used to transport liquids. We refer to all this objects as *recipients*. Our test set is formed by 30 indoor images from different scenes each of it containing a recipient object (see second and third row of figure 25 to get an idea). Our task is to train the detector to generalize the recipient class without having access to

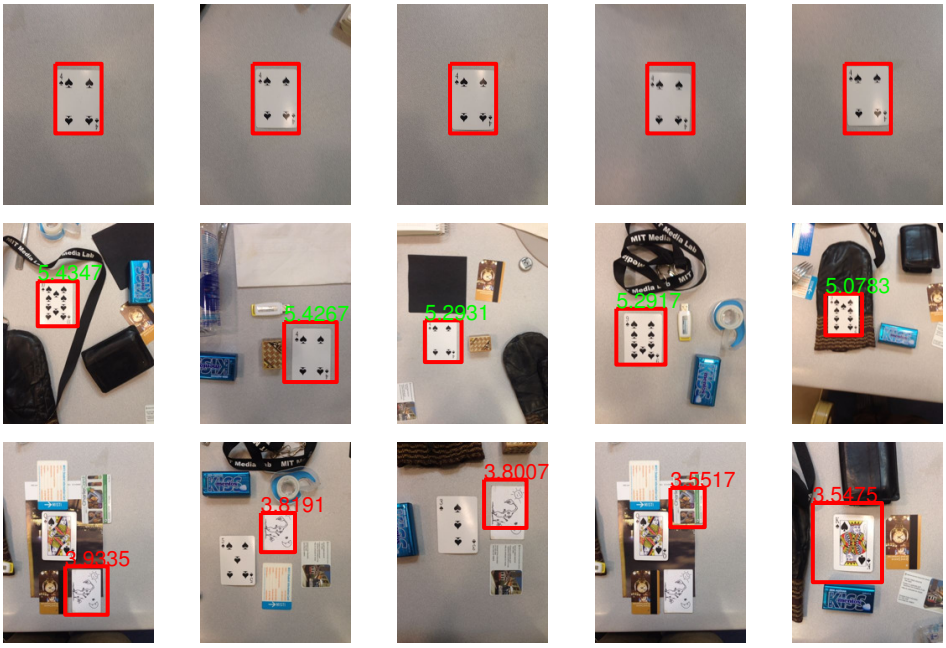


Figure 22: Insight on the 4S detector. First row show random 5 images from the training set. Second and third rows show top positive and negative detections with its scores.

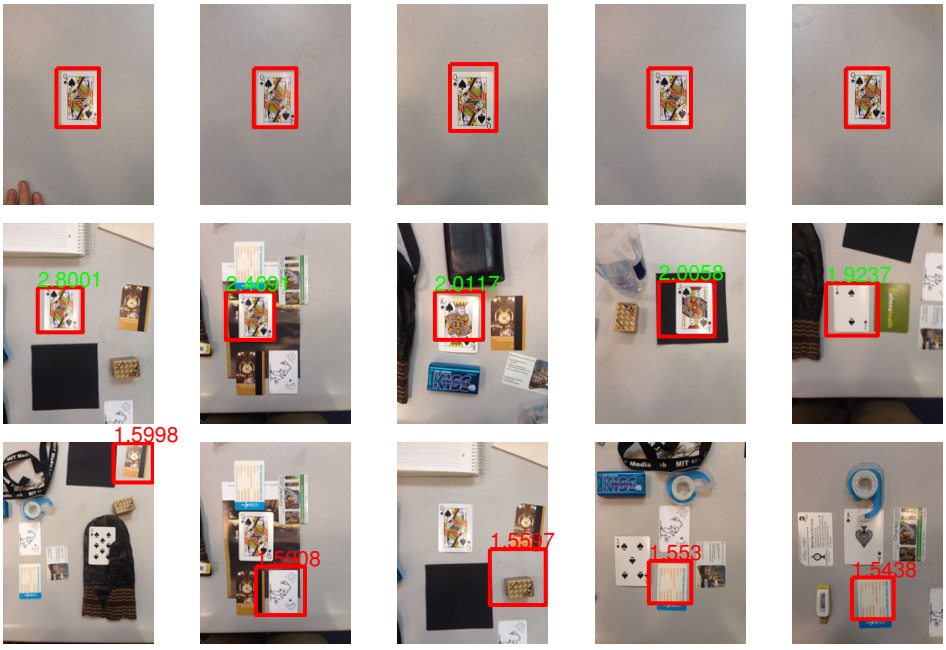


Figure 23: Insight on the QS detector. First row show random 5 images from the training set. Second and third rows show top positive and negative detections with its scores.

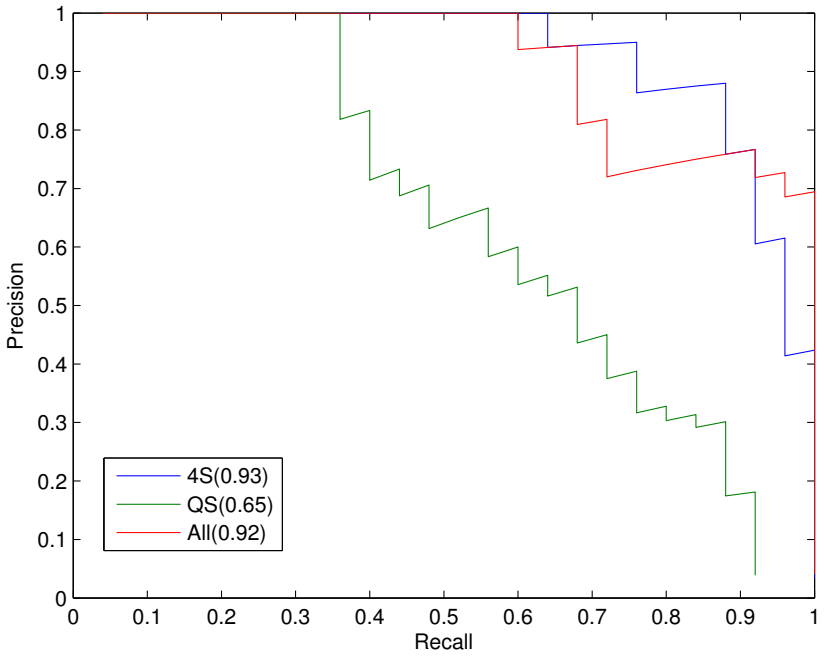


Figure 24: Precision-recall curves for the 4S, QS and all card detectors.

the specific scenes or the particular objects.

We realized 4 different trainings. All of them contain 20 images and share the same *clean* scenario: a table with a wood background and a single object in the image (see first row of figure 25 as example). For each detector we present the average of four different realizations with different images under the same concept.

The first detector was trained by a single object that tried abstract the recipient class: a disposable coffee cup. Although this object was just in 4 out of the 30 images of the training set this detector obtained the best results with an average precision of 0.37. Figure 26 presents 5 random images of the training set (first row) the top true positives (second row) and the top false positives (third row).

The second detector was trained with also with just a single object: a ceramic coffee cup with a handle. Figure 26 contains this second detector. The performance was worse than the previous one and the results presented much more variance when averaging different realizations. There are several reasons that explain this facts. The first thing to notice is that the handle

reduces the generalization capacity. It is like training with QS instead of the 4S in the previous examples. But because of the reduced number of images in the training set and the test set, the handle is also responsible of increasing the variance of the results. Depending on the matching between the handles in the training set and the test set we obtained better or worst results.

The last two detectors look what happens to the performance when more objects are used to train. Figures 27 and 28 show the detectors for 2 and 4 different objects. The first one just combine the two previous object with 10 images of each one. The second one add 2 more objects taking 5 images per object. The additional objects have been selected as representative of different subgroups on the test set. The results though also decrease the performance of the detections. This is also coherent with the results obtained with the cards. The reason behind this can be that with just 20 images, the classifier gets a better picture of the problem it is trained with easy examples. When adding complexity to the training set, we also make the classifier task more difficult and with so less examples a restrictions on the amount of iterations, the convergence can be quite hard. This is translated in a weaker detector that produces more false positives for not being sure enough of the positive detections. This is related with the result obtained by Kumar et al. [9] where they suggest training non-convex problems with easy examples first.

4.2.3 Conclusions

In this second set of experiments we have seen the performance of the detector when trained for generalizing. Comparing with the specific detector, we have been able to see how the DetectMe makes a better job generalizing than detecting specific objects. This is mainly due to the robustness of the HOG features. For the different strategies to train a general detector we have seen the power of just training with a very simple object representing the class. Choosing a bad example for the class damages the performance. We also have seen how if we wisely choose simple examples that abstract the class being detected, very good results can be obtained with very few class examples. Furthermore, adding complexity to the training set for a small fixed set of images it can translate to a worse performance.

5 Related work

Most of the object detectors for iOS and mobile devices focus on the detection of *specific* objects. They do a very good job extracting features from a particular object and recognizing it afterwards but they fail when trying to generalize to other objects on the same class. Many times the recognition

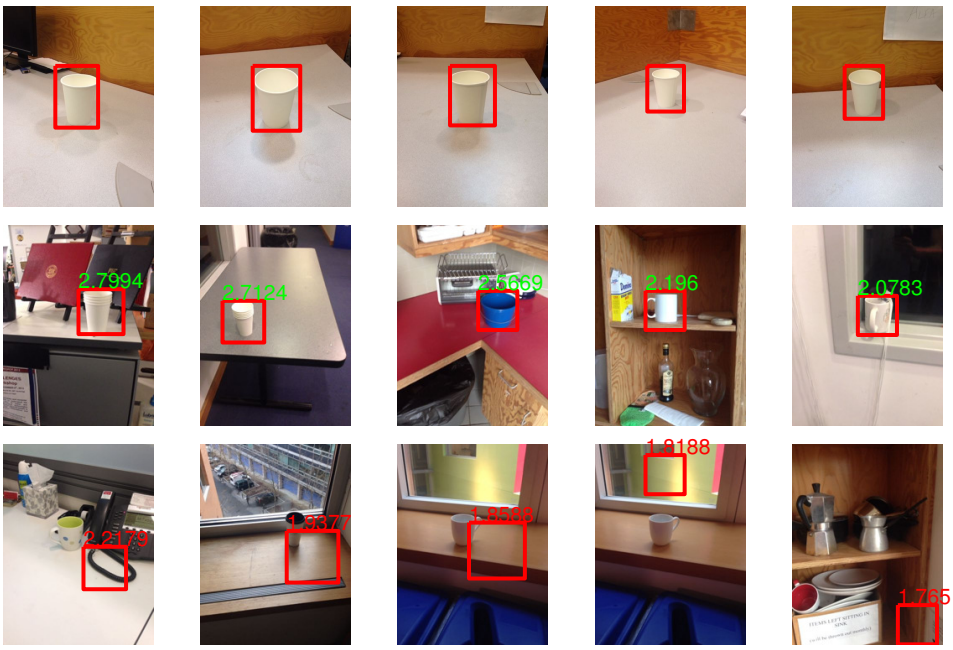


Figure 25: Training set, top true positive detections and top false positive for the detector trained with just the single plastic cup.

Figure 26: Training set, top true positive detections and top false positive for the detector trained with just the single cup with handle.

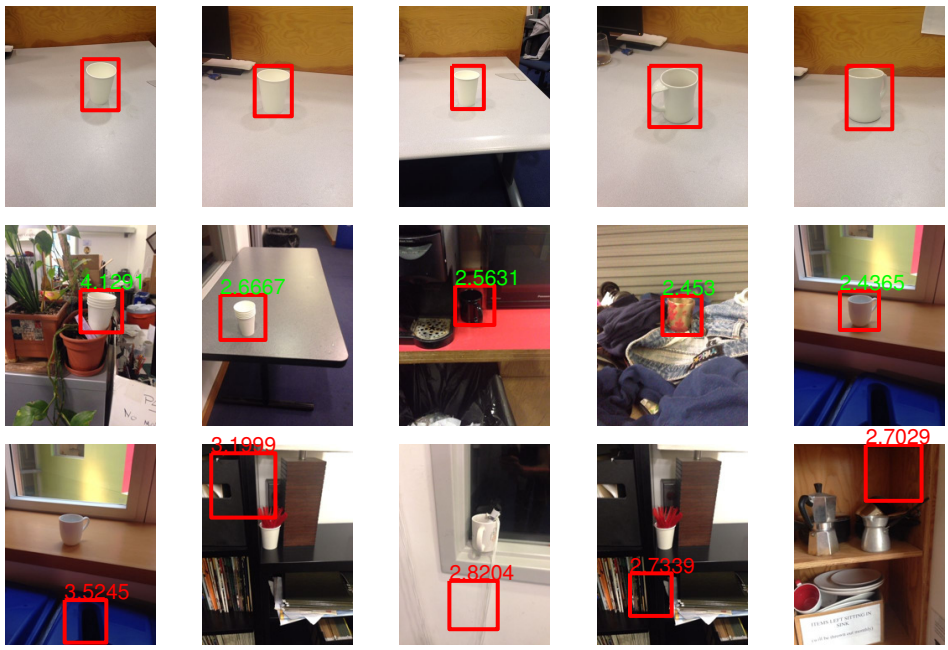


Figure 27: Training set, top true positive detections and top false positive for the detector trained with 2 objects: the plastic cup and the ceramic cup with handle.

Figure 28: Training set, top true positive detections and top false positive for the detector trained with 4 objects shows in the first row.

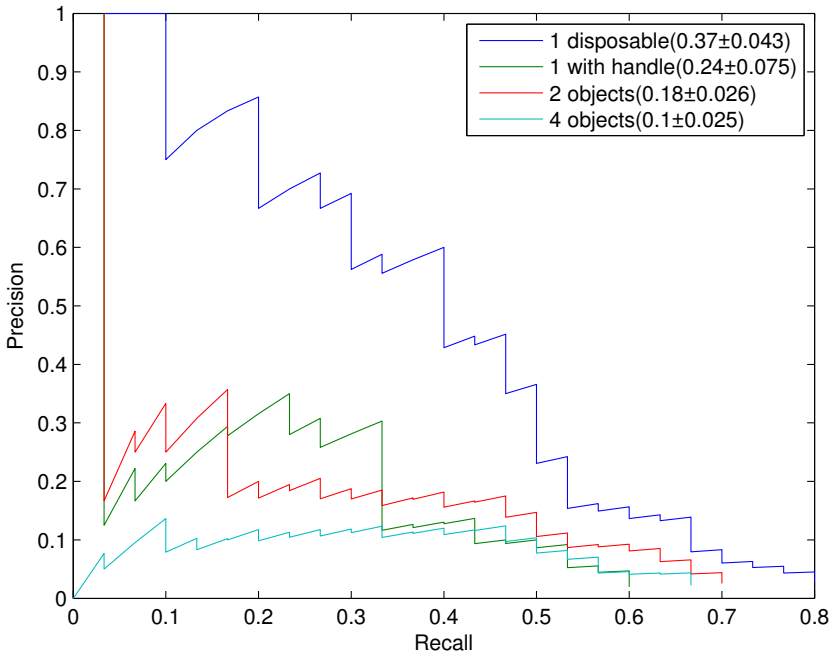


Figure 29: Precision recall curves for the 4 detectors discussed. Average precision in the legend.

process is also limited to locate the object on a specific region of the camera so they do not scan over the whole image to localize it. This kind of object detectors are meant to be integrated with SDKs or other apps with focus on augmented reality. Notorious SDKs for augmented reality and image recognition are Vuforia from Qualcomm and Moodstocks as a third party service. Aurasma is also a famous augmented reality app used to produce 3D animations over real objects throughout the mobile. LookTel is a well-known app to help the visual impaired to recognize objects. It comes with OCR capabilities and it allows to define names for specific objects. The main differences of all these apps and SDKs with DetectMe is that we allow the detection of the object scanning the whole image and the our capacity of generalization due to the use of HOG features with hard-negative mining.

The computer vision framework OpenCV [1] can be implemented in the iOS devices. The framework provides an already built in function for computing the HOG features from Dalal et al. [3]. In DetectMe we make use of the optimized HOG features from Felzenszwalb et al. [7]. We also have optimized the learning and detecting process to make use of the dual core of the iOS devices running as much processes in parallel as possible.

iOS itself also present some object detection capabilities. Core Image, the main Apple library for dealing with images, have a built in function for detecting faces. Right now however, this is the only detector available. A very notorious library for image processing on the iOS GPUs is Brad Larsson's GPUImage. This library allows the computation of common image processing filters and transformation to be run on the GPU with notorious speed ups over the default Core Image functions. HOG features however is still not implemented for it. As it will be seen in the Future Work section, implementing the HOG features on the GPU and the integration with the GPUImage libraries is one of our future goals.

6 Future work

DetectMe is going to be in the App Store at the beginning of the 2014. The further future work is divided between detector performance, app features and complementary projects.

The detector performance can be improved in accuracy and velocity. To improve it in accuracy we are exploring separating the general and specific task in two sets of distinct features to exploit, for example, the color when detecting a specific object. New features just present in the mobile devices such as the accelerometer 3DOF position can also be used to learn from the natural bias of most of the objects (e.g. cups usually are presented in

the same plane). The detector velocity can be highly improved computing the HOG features of the image on the iPhone GPU. Although some work has been done on implementing this features on the GPU (see for example [8]) usually they make use of the General Purpose GPU for example with NVIDIA CUDA making it easy to implement. We do not know of any work done to implement HOG features completely on the GPU without making use of the GP/GPU. The convolution can also be speed up when doing multiple object detection for example implementing the Dean and Segal 100000 object class detector [4].

Additional features can be added depending on the specific purpose of the detector. One of the most important ones is to automatize the way a detector can be evaluated. For this work, all the performance evaluations have been done through Matlab, but it would be very helpful that the user could get instant feedback on the detector performance. Another interesting feature is to give to the user more control over what to do when a detector is made. Right now, the phone is able to broadcast the detections and an experienced user can use this to program any kind of feature. But for the average user it is necessary to have a platform where actions (such as send mail, perform sounds,..) can be programmed from a high level perspective.

Finally this framework for training detectors on the iPhone can initiate a bunch of complimentary projects. The most interesting one is to analyze how different people train detectors and explore which training strategies produce better results. As we have shown with the training examples on this work, the training set plays a very important role in the detector training. Having a crowd sourced training of detectors can discover interesting new strategies. Also interesting is trying to correlate how the user perceives the performance of the detector with the traditional performance tests, using the PR curves and the AP over a test set. The traditional procedures for evaluating a detector do not take into account the detector speed and for the user this parameter can have a great influence on the detector performance. A fast detector can camouflage its false positives with much more detections while a slow accurate detector can destroy the real time user experience.

7 Conclusions

DetectMe is one of the first object detectors for the iPhone. Through out the development of this project we have faced many different particularities for implementing an object detector on a mobile device. We remark three main observations we have experienced.

First of all the trade off accuracy-speed. Almost any of the parameters

in the detector training and execution can be tuned to focus on accuracy or speed. It is though very important to define well what is our direction and which are our design constraints. Otherwise some parameters can contradict in the trade off and produce worse results. In our case, we have tried to always balance over speed to provide a good real time user experience. It will also interesting to see how the users rating over the detectors correlates with the traditional detector ratings. As already said, we believe that the speed of execution can play a very important role for humans in terms of perceived performance.

On second place, the importance of the device where the detector is running. The concrete capabilities of the iOS devices have guided many of our design choices. The mobile devices present some special features that need to be carefully considered and it is not direct to apply the design of detectors from desktop computers. In that sense, we have had many times important limitations on the amount of memory and also on the processing time. It is very important to keep in mind that all the processing time also consumes battery life so the app has to efficiently manage this resource as well.

Finally remark the importance of the training set. Object detection can be seen as an specific application of Machine Learning. Among all the particularities, one has captured our attention. Many times the goal of machine learning is to extract patterns of *given* data. In object detection though, one has to produce that data as the training set for the classifier. In this work we have analyzed many different examples where the performance of the detector was radically affected for the selection of the training set. We have seen how choosing the learning strategy can make the detector perform better in some task but worse on others. As already said in the future directions of the project, we believe that the DetectMe app can be very helpful in this task of selecting an adequate training set. By taking out the weight of the detection from the algorithm used to the training set selection, we enable users to rapidly test many different set of hypothesis and see the results. We can analyze on all this different training sets all constructed under the same features defined by the mobile device.

References

[1] G. Bradski. Opencv. *Dr. Dobb's Journal of Software Tools*, 2000.

[2] C. Chang and C. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[3] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *In CVPR*, 2005.

[4] T. Dean, M. Ruzon, M. Segal, J. Shlens, S. Vijayanarasimhan, and J. Yagnik. Fast, accurate detection of 100,000 object classes on a single machine. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[5] P. Dollár, S. Belongie, and P. Perona. The fastest pedestrian detector in the west. In *BMVC*, 2010.

[6] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[7] P. Felzenszwalb, R. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models, 2010.

[8] M .Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita. Gpu implementations of object detection using hog features and deformable models. 2013.

[9] P. Kumar, B. Packer, and D. Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems (NIPS 2010)*, 2010.

[10] D. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision.*

[11] V. Prisacariu and I. Reid. fasthog - a real-time gpu implementation of hog. Technical report, Department of Engineering Science, Oxford University, 2009.

A Technical Specifications

DetectMe targets iOS 7 operating system. A list of all the current devices supporting this OS:

iPhone: 4, 4s, **5**, 5c, 5s.

iPod: 5th generation.

Pad: 2, 3rd and 4th generation, Air.

The app has been tested on the above highlighted devices. However all the performance metrics and exhaustive tests have been executed on the iPhone 5. This particular device presents the following technical specifications:

Processor 1.3 GHz dual-core Apple-designed ARMv7s (*Apple A6*).

RAM 1GB LPDDR2 DRAM.

Graphics PowerVR SGX543MP3 (tri-core, 266 MHz) GPU.

Storage 16, 32 or 64 GB.

Back Camera 8 MP photos, f/2.4, 1080p HD video (30 fps), Infrared cut-off filter, Back-illuminated sensor, face detection, video stabilization, panorama and ability to take photos while shooting videos.

Front Camera 1.2 MP photos, 720p HD video (30 fps), Back-illuminated sensor.

For the preset we use, the camera uses images with a resolution of 460×380.