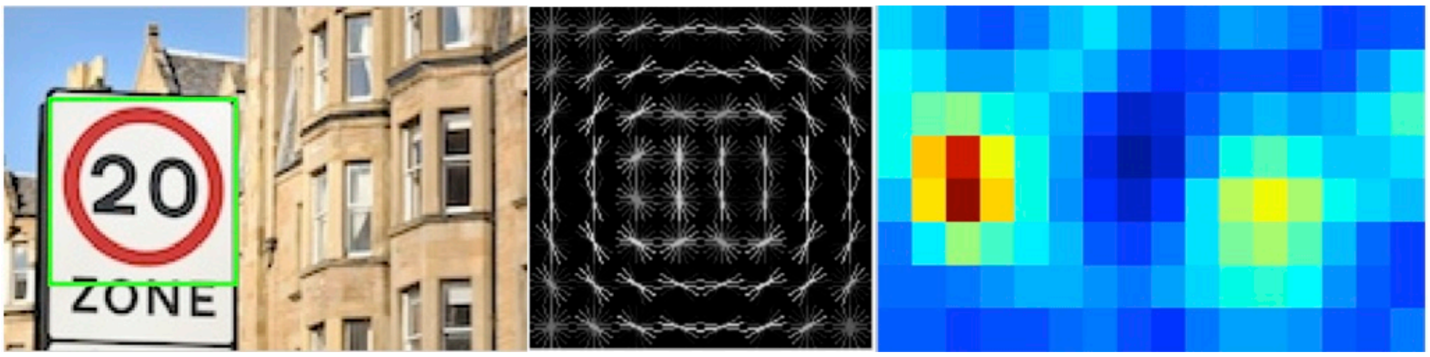
Object category detection



A computer vision practical of the Oxford Visual Geometry Group (http://www.robots.ox.ac.uk/~vgg), authored by Andrea Vedaldi (http://www.robots.ox.ac.uk/~vedaldi).

The goal of *object category detection* is to identify and localize objects of a given type in an image. Examples applications include detecting pedestrian, cars, or traffic signs in street signs, objects of interest such as tools or animals in web images, or particular features in medical image. Given a target class, such as *people*, a *detector* receives as input an image and produces as output zero, one, or more bounding boxes around each occurrence of the object class in the image. The key challenge is that the detector needs to find objects regardless of their location and scale in the image, as well as pose and other variation factors, such as clothing, illumination, occlusions, etc.

This practical explores basic techniques in visual object detection, focusing on *image based models*. The appearance of image patches containing objects is learned using statistical analysis. Then, in order to detect objects in an image, the statistical model is applied to image windows extracted at all possible scales and locations, in order to identify which ones, if any, contain the object.

In more detail, the practical explores the following topics: (i) using HOG features to describe image regions, (ii) building a HOG-based sliding-window detector to localize objects in images; (iii) working with multiple scales and multiple object occurrences; (iv) using a linear support vector machine to learn the appearance of objects; (v) evaluating an object detector in term of average precision; (vi) learning an object detector using hard negative mining.

Getting started

The practical can be downloaded here:

- Code and data: practical-detection-2014a.tar.gz ()
- Code only: practical-detection-2014a-code.tar.gz ()
- Data only: practical-detection-2014a-data.tar.gz ()
- Git repository (http://www.github.com/vedaldi/practical-detectiongit) (for lab setters and developers)

After the installation is complete, open and edit the script `exercise1.m` in the MATLAB editor. The script contains commented code and a description for all steps of this exercise, relative to Part I of this document. You can cut and paste this code into the MATLAB window to run it, and will need to modify it as you go through the session. Other files `exercise2.m`, `exercise3.m`, and `exercise4.m` are given for Part II, III, and IV (part4).

Each part contains several **Questions** and **Tasks** to be answered/completed before proceeding further in the practical.

Part 1: Detection fundamentals

In Part I–IV use as running example the problem of street sign detection, using the data from the German Traffic Sign Detection Benchmark (http://benchmark.ini.rub.de/?section=gtsdb&subsection=news). This data consists of a number of example traffic images, as well as a number of larger test images containing one or more traffic signs at different sizes and locations. It also comes with *ground truth* annotation, i.e. with specified bounding boxes and sign labels for each sign occurrence, which is required to evaluate the quality of the detector.

In this part we will build a basic sliding-window object detector based on HOG features. Follow the steps below:

Step 1.0: Loading the training data

The MATLAB m-file `loadData.m` loads the data relative to the practical into memory. The function `loadData(targetClass)` takes a `targetClass` argument specifying the object class of interest. Open the `example1.m` file, select the following part of the code, and execute it in MATLAB (right button > *Evaluate selection* or Shift+F7).

```
% Load the training and testing data (trainImages, trainBoxes, ...)
% The functio takes the ID of the type of traffic sign we want to recognize
% 1 is the 30 km/h speed limit
loadData(1) ;
```

This loads into the current workspace the following variables:

- `trainImages` : a list of train image names.
- `trainBoxes` : a $4 \times N$ array of object bounding boxes, in the form $[x_{\min}, y_{\min}, x_{\max}, y_{\max}]$.
- `trainBoxImages` : for each bounding box, the name of the image containing it.
- `trainBoxLabels` : for each bounding box, the object label. It is one of the index in `targetClass` .
- `trainBoxPatches` : a $64 \times 64 \times 3 \times N$ array of image patches, one for each training object. Patches are in RGB format.

An analogous set of variables `testImages` , `testBoxes` , and so on are provided for the test data. Familiraise yourself with the contents of these variables.

> **Question:** why is there a `trainImages` and a `trainBoxImages` variables?

Step 1.1: Visualize the training images

Select now the part of the code relative to section 1.1 and execute it. This will create an image visualizing both the complete list of object training examples and their average.

> **Question:** what can you deduce about the object variability from the average image?

Step 1.2: Extract HOG features from the training images

Object detectors usually work on top of a layer of low-level features. In this case, we use HOG (*histogram of oriented gradients*) features. In order to learn a model of the object, we start by extracting features from the image patches corresponding to the available training examples. This is done by the following `for` loop:

```
hogCellSize = 8 ;
trainHog = {} ;
for i = 1:size(trainBoxPatches,4)
  trainHog{i} = vl_hog(trainBoxPatches(:,:,:,i), hogCellSize) ;
end
trainHog = cat(4, trainHog{:}) ;
```

HOG is computed by the VLFeat (http::www.vlfeat.org) function `vl_hog` (doc (http://www.vlfeat.org/matlab/vl_hog.html)). This function takes as parameter the size in pixels of each HOG cell `hogCellSize`. It also takes a RGB image, represented in MATLAB as a $w \times h \times 3$ array (extracted as a slice of `trainBoxPatches`). The output is a $w/\mathtt{hogCellSize} \times h/\mathtt{hogCellSize} \times 31$ dimensional array. One such array is extracted for each example image end eventually these are concatenated in a 4D array along the fourth dimension.

Step 1.3: Learn a simple HOG template model

A very basic object model can be obtained by averaging the features of the example objects. This is done by:

```
w = mean(trainHog, 4) ;
```

The model can be visualized by *rendering* `w` as if it was a HOG feature array. This can be done using the `render` option of `vl_hog`:

```
figure(2) ; clf ;
imagesc(vl_hog('render', w)) ;
```

Spend some time to study this plot and make sure you understand what is visualized.

Question: Can you make sense of the resulting plot?

Step 1.4: Apply the model to a test image

The model is matched to a test image by: (i) extracting the HOG features of the image and (ii) convolving the model to the resulting feature map:

```
im = imread('data/signs-sample-image.jpg') ;
im = im2single(im) ;
hog = vl_hog(im, hogCellSize) ;
scores = vl_nnconv(hog, w, []) ;
```

The first two lines read a sample image and conver it to single format. The third line computes the HOG features of the image using the `vl_hog` seen above. The fourth line convolves the HOG map `hog` with the model `w`. It uses the function `vl_nnconv`[1] and returns a `scores` map.

Task: Work out the dimension of the `scores` arrays. Then, check your result with the dimension of the array computed by MATLAB.

Question: Visualize the image `im` and the `scores` array using the provided example code. Does the result match your expectations?

Step 1.5: Extract the top detection

Now that the model has been applied to the image, we have a response map `scores`. To extract a detection from this, we (i) find the maximum response and (ii) compute the bounding box of the image patch containing the corresponding HOG features. The maximum is found by:

```
[best, bestIndex] = max(scores(:)) ;
```

Note that `bestIndex` is a linear index in the range $[1, M]$ where M is the number of possible filter locations. We convert this into a subscript (h_x, h_y) using MATLAB `ind2sub` function:

```
[hy, hx] = ind2sub(size(scores), bestIndex) ;
```

(h_x, h_y) are in units of HOG cells. We convert this into pixel coordinates as follows:

```
x = (hx - 1) * hogCellSize + 1 ;
y = (hy - 1) * hogCellSize + 1 ;
```

Question: Why are we subtracting -1 and summing +1? Which pixel (x, y) of the HOG cell (h_x, h_y) is found?

The size of the model template in number of HOG cell can be computed in several way; one is simply:

```
modelWidth = size(trainHog, 2) ;
modelHeight = size(trainHog, 1) ;
```

Now we have enough information to compute the bounding box as follows:

```
detection = [
  x - 0.5 ;
  y - 0.5 ;
  x + hogCellSize * modelWidth - 0.5 ;
  y + hogCellSize * modelHeight - 0.5 ;] ;
```

Note: the bounding box encloses exactly all the pixel of the HOG template. In MATLAB, pixel centers have integer coordinates and pixel borders are at a distance $\pm 1/2$.

> **Question:** Use the example code to plot the image and overlay the bounding box of the detected object. Did it work as expected?

Part 2: Multiple scale and learning with an SVM

In this second part, we will: (i) extend the detector to search objects at multiple scales and (ii) learn a better model using a support vector machine. Let's start by loading the data as needed:

```
setup ;
targetClass = 'mandatory' ;
loadData(targetClass) ;
```

The `mandatory` target class is simply the union of all mandatory traffic signs.

Step 2.1: Multi-scale detection

Objects exist in images at sizes different from the one of the learned template. In order to find objects of all sizes, we scale the image up and down and search the object over and over again.

The set of searched scale is defined as follows:

```
% Scale space configuraiton
minScale = -1 ;
maxScale = 3 ;
numOctaveSubdivisions = 3 ;
scales = 2.^linspace(...
  minScale,...
  maxScale,...
  numOctaveSubdivisions*(maxScale-minScale+1)) ;
```

Given the model `w`, as determined in Part I, we use the function `detectAtMultipleScales` in order to search the object at multiple scales:

```
detection = detectAtMultipleScales(im, w, hogCellSize, scales) ;
```

Note that the function generates a figure as it runs, so prepare a new figure before running it using the `figure` command if you do not want your current figure to be deleted.

> **Question:** Open and study the `detectAtMultipleScales` function. Convince yourself that it is the same code as before, but operated after rescaling the image a number of times.
>
> **Question:** Visualized the resulting detection using the supplied example code. Did it work? If not, can you make sense of the errors?
>
> **Question:** Look at the array of `scores` maps generated by `detectAtMultipleScales` using the example code. Do they make sense? Is there anything wrong?

Step 2.2: Collect positive and negative training data

The model learned so far is too weak to work well. It is now time to use an SVM to learn a better one. In order to do so, we need to prepare suitable data. We already have positive examples (features extracted from object patches):

```
% Collect positive training data
pos = trainHog ;
```

Ino order to collect negative examples (features extracted from non-object patches), we loop through a number of training images and sample patches uniformly:

> **Task:** Identify the code that extract these patches in `example2.m` and make sure you understand it.
>
> **Question:** How many negative examples are we collecting?

Step 2.3: Learn a model with an SVM

Now thaw we have the data, we can learn an SVM model. To this end we will use the `vl_svmtrain` function. This function requires the data to be in a $D \times N$ matrix, where D are the feature dimensions and N the number of training points. This is done by:

```
% Pack the data into a matrix with one datum per column
x = cat(4, pos, neg) ;
x = reshape(x, [], numPos + numNeg) ;
```

We also need a vector of binary labels, +1 for positive points and -1 for negative ones:

```
% Create a vector of binary labels
y = [ones(1, size(pos,4)) -ones(1, size(neg,4))] ;
```

Finally, we need to set the parameter λ of the SVM solver. For reason that will become clearer later, we use instead the equivalent C parameter:

```
numPos = size(pos,4) ;
numNeg = size(neg,4) ;
C = 10 ;
lambda = 1 / (C * (numPos + numNeg)) ;
```

Learning the SVM is then a one-liner:

```
% Learn the SVM using an SVM solver
w = vl_svmtrain(x,y,lambda,'epsilon',0.01,'verbose') ;
```

> **Question:** Visualize the learned model `w` using the supplied code. Does it differ from the naive model learned before? How?

Step 2.4: Evaluate the learned model

Use the `detectAtMultipleScales` seen above to evaluate the new SVM-based model.

> **Question:** Does the learned model perform better than the naive average?
>
> **Task:** Try different images. Does this detector work all the times? If not, what types of mistakes do you see? Are these mistakes reasonable?

Part 3: Multiple objects and evaluation

Step 3.1: Multiple detections

Detecting at multiple scales is insufficient: we must also allow for more than one object occurrence in the image. In order to to so, the package include a suitalbe `detect` functin. This function is similar to `detectAtMultipleScales`, but it returns the top 1000 detector responses rather than just the top one:

```
% Compute detections
[detections, scores] = detect(im, w, hogCellSize, scales) ;
```

Task: Open and study `detect.m`. Make sure that you understand how it works.

Question: Why do we want to return so many responses? In practice, it is unlikely that more than an handful of object occurrences may be contained in any given image...

A single object occurrence generates multiple detector responses at nearby image locations and scales. In order to eliminate these redundant detections, we use a *non-maxima suppression* algorithm. This is implemented by the `boxsuppress.m` MATLAB m-file. The algorithm is simple: start from the highest-scoring detection, then remove any other detection whose overlap[2] is greater than a threshold. The function returns a boolean vector `keep` of detections to preserve:

```
% Non−maxima suppression
keep = boxsuppress(detections, scores, 0.25) ;

detections = detections(:, keep) ;
scores = scores(keep) ;
```

For efficiency, after non-maxima suppression we keep just ten responses (as we do not expect more than a few objects in any image):

```
% Further keep only top detections
detections = detections(:, 1:10) ;
scores = scores(1:10) ;
```

Step 3.2: Detector evaluation

We are now going to look at properly evaluating our detector. We use the PASCAL VOC criterion (http://pascallin.ecs.soton.ac.uk/challenges/VOC/voc2012/devkit_doc.pdf), computing *Average Precision (AP)*. Consider a test image containing a number of ground truth object occurrences $(g_1, \dots, g_m)$ and a list (b_i, s_i) of candidate detections b_i with score s_i. The following algorithm converts this data into a list of labels and scores (s_i, y_i) that can be used to compute a precision-recall curve, for example using VLFeat *vl_pr* function. The algorithm, implemented by `evalDetections.m`, is as follows:

1. The candidate detections (b_i, s_i) are sorted by decreasing score s_i.
2. For each candidate detection in order:
   a. If there is a matching ground truth detection g_j $(\text{overlap}(b_i, g_j))$[3] larger than 50%), the candidate detection is considered positive ($y_i = +1$). Furthermore, the ground truth detection is *removed from the list* and not considered further.
   b. Otherwise ,the candidate detection is negative ($y_i = -1$).
3. Any ground truth detection that remains unassigned is considered a positive object $y_i = +1$ recalled with a score equal to $s_i = -\infty$.

Question: Answer the following: (i) Why are ground truth detections considred only once? (ii) What happens if an object is detected twice?

In order to apply this algorithm, we first need to find the ground truth bounding boxes in the test image:

```
% Find all the objects in the target image
s = find(strcmp(testImages{1}, testBoxImages)) ;
gtBoxes = testBoxes(:, s) ;
```

Then `evalDetections` can be used:

```
% No example is considered difficult
gtDifficult = false(1, numel(s)) ;

% PASCAL-like evaluation
matches = evalDetections(...
  gtBoxes, gtDifficult, ...
  detections, scores) ;
```

The `gtDifficult` flags can be used to mark some ground truth object occurrence as *difficult* and hence ignored in the evaluation. This is used in the PASCAL VOC challenge, but not here (i.e. no object occurrence is considered difficult).

`evalDetections` returns a `matches` structure with several field. We focus here on `matches.detBoxFlags` : this contains a +1 for each detections that was found to be correct and -1 otherwise. We use this to visualize the detection errors:

```
% Visualization
figure(1) ; clf ;
imagesc(im) ; axis equal ; hold on ;
vl_plotbox(detections(:, matches.detBoxFlags==+1), 'g', 'linewidth', 2) ;
vl_plotbox(detections(:, matches.detBoxFlags==-1), 'r', 'linewidth', 2) ;
vl_plotbox(gtBoxes, 'b', 'linewidth', 1) ;
axis off ;
```

Task: Use the supplied example code to evaluate the detector on one image. Look carefully at the output and convince yourself that it makes sense.

Now Plot the PR curve:

```
figure(2) ; clf ;
vl_pr(matches.labels, matches.scores) ;
```

Question: There are a large number of errors in each image. Should you worry? Is what manner is the PR curve affected? How would you eliminate the vast majority of those in a practice?

Step 3.3: Evaluation on multiple images

Evaluation is typically done on multiple images rather than just one. This is implemented by the `evalModel.m` m-file.

> **Task:** Open `evalModel.m` and make sure you understand the main steps of the evaluation procedure.

Use the supplied example code to run the evaluation on the entiere test set:

```
matches = evaluateModel(testImages, testBoxes, testBoxImages, ...
  w, hogCellSize, scales) ;
```

Note: The function process an image per time, visualizing the results as it progresses. The PR curve is the result of the *accumulation* of the detections obtained thus far.

> **Task:** Open the `evaluateModel.m` file in MATLAB and add a breakpoint right at the end of the for loop. Now run the evaluation code again and look at each image individually (use `dbcont` to go to the next image). Check out the correct and incorrect matches in each image and their ranking and the effect of this in the cumulative precision-recall curve.

Part 4: Hard negative mining

This part explores more advanced learning methods. So far, the SVM has been learned using a small and randomly sampled number of negative examples. However, in principle, every single patch that does not contain the object can be considered as a negative sample. These are of course too many to be used in practice; unfortunately, random sampling is ineffective as the most interesting (confusing) negative samples are a very small and special subset of all the possible ones.

Hard negative mining is a simple technique that allows finding a small set of key negative examples. The idea is simple: we start by training a model without any negative at all, and then we alternate between evaluating the model on the training data to find erroneous responses and adding the corresponding examples to the training set.

Step 4.1: Train with hard negative mining

Use the supplied code in `example4.m` to run hard negative mining. The code repeats SVM training, as seen above, a number of times, progressively increasing the size of the `neg` array containing the negative samples. This is updated using the output of:

```
  [matches, moreNeg] = ...
     evaluateModel(...
     vl_colsubset(trainImages', schedule(t), 'beginning'), ...
     trainBoxes, trainBoxImages, ...
     w, hogCellSize, scales) ;
```

Here `moreNeg` contains the HOG features of the top (highest scoring and hence most confusing) image patches in the supplied training images.

Task: Examine `evaluateModel.m` again to understand how hard negatives are extracted.

Question: What is the purpose of the construct `vl_colsubset(trainImages', schedule(t), 'beginning')`? Why do you think we visit more negative images in later iterations?

The next step is to fuse the new negative set with the old one:

```
% Add negatives
neg = cat(4, neg, moreNeg) ;
```

Note that hard negative mining could select the same negatives at different iterations; the following code squashes these duplicates:

```
% Remove negative duplicates
z = reshape(neg, [], size(neg,4)) ;
[~,keep] = unique(z','stable','rows') ;
neg = neg(:,:,:,keep) ;
```

Step 4.2: Evaluate the model on the test data

Once hard negative mining and training are done, we are ready to evaluate the model on the *test* data (note that the model is evaluated on the *training* data for mining). As before:

```
evaluateModel(...
    testImages, testBoxes, testBoxImages, ...
    w, hogCellSize, scales) ;
```

---

1. This is part of the MatConvNet toolbox for convolutional neural networks. Nevertheless, there is no neural network discussed here. ↩
2. Measured as area of the intersection over are of the union: $|A \cap B|/|A \cup B|$. ↩
3. Measured as area of the intersection over are of the union: $|A \cap B|/|A \cup B|$. ↩