A web-service for object detection using hierarchical models

Domen Tabernik[1], Luka Čehovin[1], Matej Kristan[1], Marko Boben[1] and Aleš Leonardis[1,2]

[1] Faculty of Computer and Information Science, University of Ljubljana, Slovenia
[2]CN-CR Centre, School of Computer Science, University of Birmingham
domen.tabernik@fri.uni-lj.si

Abstract. This paper proposes an architecture for an object detection system suitable for a web-service running distributed on a cluster of machines. We build on top of a recently proposed architecture for distributed visual recognition system and extend it with the object detection algorithm. As sliding-window techniques are computationally unsuitable for web-services we rely on models based on state-of-the-art hierarchical compositions for the object detection algorithm. We provide implementation details for running hierarchical models on top of a distributed platform and propose an additional hypothesis verification step to reduce many false-positives that are common in hierarchical models. For a verification we rely on a state-of-the-art descriptor extracted from the hierarchical structure and use a support vector machine for object classification. We evaluate the system on a cluster of 80 workers and show a response time of around 10 seconds at throughput of around 60 requests per minute.

1 Introduction

Improvements in the field of computer vision and the available processing power from hardware improvements are enabling many new web-services to rely on computer vision algorithms. While many services work very well in most cases, they still have certain limitations. In particular, in the context of the content-based image search, addressed by services such as Google Image Search[1], many services work great for images found on the internet but fail completely for images not available on the internet. For instance, querying a Google Image Search with an image of a coffee mug captured by a cell-phone camera will not return any related images even though there are millions of images of coffee mugs found on the internet.

To improve content-based image retrieval the authors of [9] proposed to include more advanced computer vision algorithms. By finding semantic meaning of object(s) a system could utilize this information to improve content-based image retrieval. They provide an architecture for building a web-service that could achieve this but focus only on object categorization which requires object annotations. This service may be adequate for applications where user can easily input location of object, but when user cannot provide location or there might be more different objects in the image then simple

[1] http://images.google.com/

categorization cannot return proper image description. This problem can be addressed by adding localization capabilities to obtain additional regions of interest which can provide more accurate image description.

A first candidate for object detection would be the current popular state-of-the-art algorithm which utilizes HOG descriptor with a discriminatively trained deformable part models [2]. But running an algorithm as a web-service imposes fairly strict constraints on the processing speed. A web-service must quickly respond to each request since the user is not willing to wait for the results for too long. In [9], the authors achieve response time of two seconds for their service of object categorization, but response time for object detection can be higher due to localization and multiscale processing. With sliding-window being the main technique for the current state-of-the-art method a testing with hundreds of thousands of windows over different scales becomes computationally prohibitive. Certain optimization techniques [7,10] have been presented to mitigate this problem but they require dedicated hardware (FPGA,GPU) and do not address the problem of scaling to higher number of categories. Given those constraints a much more viable approach is using a hierarchical models such as [5,6,11], which enable sharing of parts among different categories [4]. This allows for more efficient object encoding and faster inference of multiple categories while avoiding the sliding windows. A significant drawback of hierarchical models are false-positive detections that occur in highly textured or cluttered images.

In this work we build on top of an architecture proposed by [9] and use their system to implement object detection web-service. As our first contribution, we demonstrate a detection using hierarchical models for a fast processing in a web-service. To achieve fast and distributed real-time processing we utilize the same Storm[2] platform as was used in [9]. As a second contribution, we propose a hypothesis verification step to address the problem of false-positives in hierarchical methods. The presented system can function not only as support for category-oriented content-based image retrieval, but as a standalone web-service it can also serve as a backend to multitude of different applications, such as providing a backend support for a robotic vision system.

The remainder of the paper is structured as follows. In Section 2 we present detection with hierarchical model and a hypothesis verification step. We provide a detailed implementation of the algorithm as a Storm topology in Section 3 and present performance analysis in Section 4. We conclude the paper in Section 5.

2 Object detection with hierarchical models

Hierarchical models follow the principle of modeling shapes with hierarchical compositions. They are arranged in layers where each layer represents shapes composed of simpler shapes from the previous layer. The lower layer usually represent small and simple shapes while higher layer capture shapes complex enough for object description. While many different hierarchical models exist, we demonstrate object detection on learnt-hierarchy-of-parts [5] (LHOP) model.

In the following we will denote the library of hierarchical parts trained for up to L layers as a set of N compositions $\mathcal{L} = \{P_i^l\}_{i=1:N}$, where P_i^l is an identifier of i-th

[2] http://storm-project.net

composition and belongs to the l-th layer of the library. At the last layer L, in our case LHOP library was learnt with up to 6 layers, each composition directly identifies one trained category, i.e. for each category we have only one corresponding composition on the L-th layer. Applying the library $\mathcal{L}$ on a given image $\mathcal{I}$, the algorithm of hierarchical model infers a set of K detected parts, $\mathcal{C}(\mathcal{I}, \mathcal{L})$,

$$\mathcal{C}(\mathcal{I}, \mathcal{L}) = \{\pi_k^l\}_{k=1:K}, \tag{1}$$

where the k-th detected part on the l-th layer $\pi_k^l = [P_k, \mathbf{c}_{\pi_k}, \lambda_k]$ is defined by its library identifier P_k^l, its location $\mathbf{c}_{\pi_k}$ in the image and its detection score λ_k. All the inferred parts from the last layer L directly correspond to detected objects in the image:

$$\mathcal{D}(\mathcal{I}, \mathcal{L}) = \{\pi_j^L\}_{j=1:J} \tag{2}$$

where $\mathcal{D}(\mathcal{I}, \mathcal{L})$ is a set of J detected objects in the image $\mathcal{I}$ processed with the library $\mathcal{L}$. While each detected object is defined the same as detected part at L-th layer $\pi_j^L = [P_j^L, \mathbf{c}_{\pi_j}, \lambda_j]$, we can also add a category information since a library identifier P_j^L from the L-th layer always directly matches to one learning category. We can also obtain a bounding box location of detected object simply by tracing down sub-parts of detected part π_j^L to the first layer. Minimal and maximal locations of all traced sub-parts define a bounding box of detected image. We can therefore define a set of detected objects from a given image $\mathcal{I}$ as:

$$\mathcal{D}(\mathcal{I}, \mathcal{L}) = \{(\pi_j^L, c_j, r_j)\}_{j=1:J}, \tag{3}$$

where c_j is detected category and $r_j = (x, y, w, h)$ is a detection bounding box.

2.1 Hypothesis verification

A common problem of many hierarchical models is occurrence of false-positive detections. In the LHOP false-positives occur due to allowed flexibility between detected sub-parts in the image and corresponding composition in the library and as such the constellation of sub-parts can slightly vary. This causes false objects to appear mostly in textured or cluttered images. To reduce the problem we introduce hypothesis verification by computing a descriptor from each detected hypothesis and use Support Vector Machine to verify the detected category. As descriptor we use Histogram of Compositions [8] (HoC), which relies on the same hierarchical structure as the LHOP model.

For each detected object from $\mathcal{D}(\mathcal{I}, \mathcal{L})$ we discard category information c_j and retain only detected bounding box r_j. Within a bounding box r_j we calculate a HoC descriptor $\mathcal{H}_j$ from detected parts of second and third layer. In [8] the authors use library pre-trained on a general set of images but since the HoC descriptor incorporates the same LHOP model as we use for the detection we can easily compute descriptor on the same library of compositions. This also eliminates the time required to reprocess the image with different library. All computed descriptors $\mathcal{H}_j$ are in the end sent to the Support Vector Machine for classification. We use LIBSVM [1] with an RBF-kernel and χ^2 distance function. As the final step we perform a non-maximum suppression using simple greedy approach.

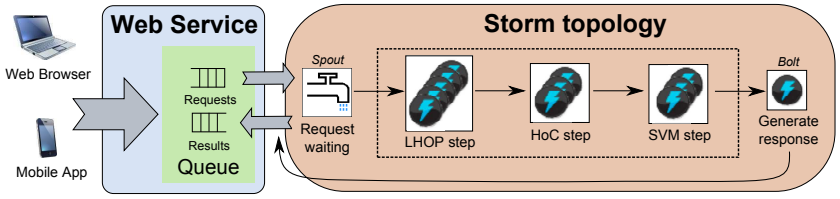

Fig. 1: Storm topology overview. Incoming requests from a web browser or a mobile application are received by the Web Service. Requests are pushed into a ***requests queue***, where main spout from storm topology listens for new requests. The final response is generated by the last bolt, which pushes the response into a ***results queue*** for web service to forward it back to the user.

3 A distributed implementation in Storm

In this section we detail the detection as a Storm application. Storm is a distribute platform for real-time stream processing of BigData. Its computational model consists of a directed graph, a topology, with vertices representing parallel processing elements and edges represening direction of data stream. We build our topology for this computational model from three main steps: (i) an LHOP step where we process the image and produce hierarchical compositions π_k^l, (ii) a HoC step where we extract detected regions r_j and produce HoC descriptor $\mathcal{H}_j$ for each detected object π_j^L and (iii) an SVM step where we classify each descriptor into pre-trained categories (see, Fig. 1 for overview).

3.1 The LHOP step

We first define one main *Spout* termed *Request waiting spout* which communicates with a web-service through a queuing system. In our case we are using Beanstalk[3] queues. When a request is initiated by the user it is first received by the web-service which forwards it to the requests queue. The request comes in a form of a meta-data and a query image $(Metadata, Image)$. Meta-data defines additional information about request and among other contains a request identification and type of service requested by the user. Type of service is used to select the proper library for LHOP processing and for selecting proper group of SVM models for classification. The process of this spout is summarized in Algorithm 1. We assign only one worker for this spout as its workload is small and one worker can quickly handle multiple requests.

[3] http://kr.github.com/beanstalkd/

Algorithm 1 Request waiting spout running as a single worker

Input tuple: /
Output tuple: $(Metadata, Image)$
1: **loop**
2: $request =$ pop_queue$(requests_queue)$
3: $\mathcal{I} \leftarrow$ extract_image$(request)$
4: $Metadata \leftarrow$ extract_metadata$(request)$
5: **emit** $(Metadata, \mathcal{I})$
6: **end loop**

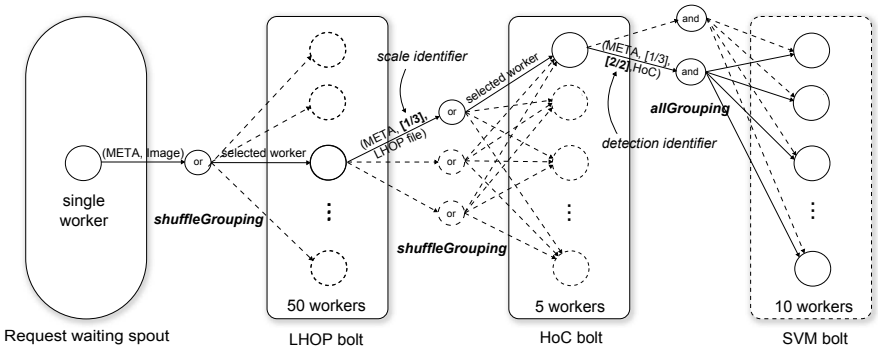

Fig. 2: Request from web-service is received by the *Request waiting spout* that emits it to the *LHOP bolt*. Due to ***shuffle grouping*** a single LHOP worker is selected for processing of the tuple. Selected LHOP worker emits scale compositions as separate tuples. Three scales are emitted in this example. Each is sent to the *HoC bolt* using ***shuffle grouping***. Following the first scale, this tuple is assigned to one of HoC workers. The HoC worker processes compositions and emits a tuple with the descriptor for each detection. Two detections are emitted in this example. Following the second detection, this tuple is sent to all workers of the *SVM bolt* by using ***allGrouping***.

As a second part of this step we implement the *LHOP bolt* (see, Algorithm 2). We connect it to the Request waiting spout therefore this bolt receives $(Metadata, Image)$ tuple as input from the spout. The connection between this spout and bolt is a shuffle grouping which ensures optimal load balancing between processes assigned for the LHOP. This is achieved since with shuffle grouping for each request emitted by the spout the LHOP worker chosen to process it is selected using even distribution of workload (see, Fig. 2). The work for LHOP bolt consists of computing LHOP compositions $\{\pi_k^l\}_i$ from input image $\mathcal{I}$ for each scale i and emitting each scale to next bolt. We emit result as multiple new tuples, for each scale separately, to ensure detections of each scale can be processed in parallel. As an output tuple we pass along result together with original meta-data and scale identifier. We use meta-data as request identifier since there will always be multiple requests in the system and each bolt has to know associated request for every input tuple it receives. We also use a scale identifier, which is composed out of a scale index i and a max scale number, to help succeeding bolts associate input tuple with correct scale. The LHOP processing is the most consuming part of the system, therefore we assign 50 workers for this bolt.

Algorithm 2 LHOP-processing bolt running as 50 workers

Input tuple: $(Metadata, Image)$
Output tuple: $(Metadata, scale\_indexing, LHOP\_file)$
1: $\mathcal{L} \leftarrow$ select_library$(Metadata)$
2: $\mathcal{I} \leftarrow Image$
3: **for** $i = 1$ to max_scale **do**
4: $\quad \{\pi_k^l\}_i =$ construct_LHOP_hierarchy$(\mathcal{I}, \mathcal{L}, i)$
5: $\quad$ **emit** $(Metadata, [i/max\_scale], \{\pi_k^l\}_i)$
6: **end for**

Algorithm 3 HoC-processing bolt running as 5 workers

Input tuple: $(Metadata, scale\_indexing, LHOP\_file)$
Output tuple: $(Metadata, scale\_indexing, detection\_indexing, HoC\_descriptor)$
1: $\mathcal{L} \leftarrow$ select_library$(Metadata)$
2: $[i/max\_scale] \leftarrow scale\_indexing$
3: $\{\pi_k^l\}_i \leftarrow LHOP\_file$
4: $\{\pi_j^L\}_i =$ find_detections$(\{\pi_k^l\}_i)$
5: $num\_detections =$ size$(\{\pi_j^L\}_i)$
6: **for** $j = 1$ to $num\_detections$ **do**
7: $\quad r_{i,j} =$ get_bounding_box(π_j^L)
8: $\quad \mathcal{H}_{i,j} =$ compute_HoC$(\{\pi_k^l\}_i, r_{i,j})$
9: $\quad$ **emit** $(Metadata, [i/max\_scale], [j/num\_detections], (\mathcal{H}_{i,j}, r_{i,j}))$
10: **end for**

3.2 The HoC step

In the second main step we extract locations of all detected object of each scale, generate a HoC descriptor for each detection and pass it along to next step. The process of *HoC bolt* is described in Algorithm 3. We connect this bolt to the LHOP with shuffle grouping to allow for even distribution of workload (see, Fig. 2). Each HoC bolt worker will receive tuple with meta-data (i.e. request identifier), scale identifier and LHOP compositions for that scale. This bolt will output multiple new tuples, one for each detection, containing meta-data, scale identifier, detection identifier and a HoC descriptor with bounding box information. We assign only 5 workers for this bolt as this process is not computationally expensive.

3.3 The SVM step

The last main step in our topology consists of classifying each detection with a Support Vector Machine. We implement this in the *SVM bolt* as described in Algorithm 4 and connect it to the HoC bolt using *allGrouping* (see, Fig. 3). This type of grouping is used for sending tuple not only to one worker but to all workers at once, meaning that all workers of the SVM bolt will receive the same tuple with the descriptor from one detection. We can exploit this feature and instruct each worker to test descriptor

Algorithm 4 SVM-processing bolt running as 10 workers

Input tuple: $(Metadata, scale\_indexing, detection\_indexing, HoC\_descriptor)$
Output tuple: $(Metadata, scale\_indexing, detection\_indexing, svm\_score\_array)$
1: $task\_id \leftarrow$ running_process_id()
2: $\{svm\_model_m\} \leftarrow$ select_svm_category_subset$(Metadata, task_id)$
3: $[i/max\_scale] \leftarrow scale\_indexing$
4: $[j/num\_detections] \leftarrow detection\_indexing$
5: $(\mathcal{H}_{i,j}, r_{i,j}) \leftarrow HoC\_descriptor$
6: **for** $m = 1$ to size$(\{svm_model_m\})$ **do**
7: $\quad score_m =$ svm_classify$(\mathcal{H}_{i,j}, svm_model_m)$
8: **end for**
9: **emit** $(Metadata, [i/max\_scale], [j/num\_detections], (\{score_m\}_{i,j}, r_{i,j}))$

Algorithm 5 SVM-grouping bolt running as 5 workers

Input tuple: $(Metadata, scale\_indexing, detection\_indexing, svm\_score\_array)$
Output tuple: $(Metadata, scale\_indexing, detection\_indexing, best\_svm\_score)$
1: $[i/max\_scale] \leftarrow scale\_indexing$
2: $[j/num\_detections] \leftarrow detection\_indexing$
3: $(\{score_m\}_{i,j}, r_{i,j}) \leftarrow svm\_score\_array$
4: save_scores_to_memory($(Metadata, i, j), \{score_m\}$)
5: **if** saved all scores for $(Metadata, i, j)$ **then**
6: $saved\_scores_{i,j}$ = get_saved_scores_from_memory$(Metadata, i, j)$
7: $(c_{i,j}, score_{i,j})$ = select_best_category$(saved_scores_{i,j})$
8: **emit** $(Metadata, [i/max\_scale], [j/num\_detections], (c_{i,j}, score_{i,j}, r_{i,j}))$
9: **end if**

not on all categories but only on a subset of categories. This allows us to achieve distributed processing of a single detection and lower the response time for one request. We assign 10 workers for this bolt meaning each worker should handle every 10th category. When testing for more then 10 categories the processing time should reduce to 1/10th of the processing time with all the categories, while for less then 10 categories the improvement would not be noticeable.

To merge the classifications of a single detection from multiple workers we introduce additional bolt termed *SVM-grouping bolt*. We connect it to the output of the SVM bolt using field grouping. This type of grouping always sends tuple with the same field values to the same worker. We use this grouping to send tuples with the classifications belonging to the same detection to only one worker (see, Fig. 3). Worker can then save partial classifications to memory until all classifications for a single detection have been collected and after that worker outputs the collected classifications as a new single tuple. This bolt is described in Algorithm 5. We use only 5 workers for this bolt.

Algorithm 6 Detection grouping bolt running as 5 workers

Input tuple: $(Metadata, scale\_indexing, detection\_indexing, best\_svm\_score)$
Output tuple: $(Metadata, detections)$
1: $[i/max\_scale] \leftarrow scale\_indexing$
2: $[j/num\_detections] \leftarrow detection\_indexing$
3: $(c_{i,j}, score_{i,j}, r_{i,j}) \leftarrow best\_svm\_score$
4: save_detections_to_memory$(Metadata, (i, j, c_{i,j}, score_{i,j}, r_{i,j}))$
5: **if** saved all detections for $(Metadata)$ **then**
6: $saved\_detections$ = get_saved_detections_from_memory$(Metadata)$
7: $merged\_detections$ = non_maximus_suppression$(saved_detections)$
8: **emit** $(Metadata, merged\_detections)$
9: **end if**

3.4 The detection merging step

In the LHOP and HoC bolts a single detection has been split into multiple tuples for optimal computation and they need to be merged back into a single tuple before user re-

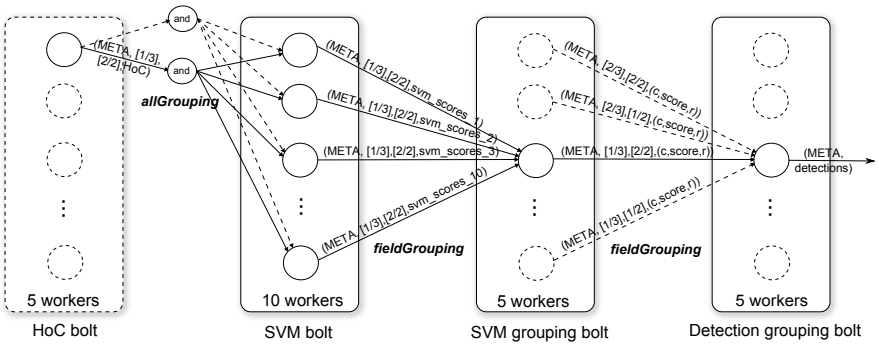

Fig. 3: The *SVM bolt* receives tuple with a HoC descriptor. Due to ***allGrouping*** connection between bolts a descriptor is sent to all workers where at each worker it is tested with the SVM models of a specific subset of categories. Next, using ***field grouping*** all classifications of a single detections are always sent to a single worker of the *SVM-grouping bolt*. There, classifications are merged together and sent to the *Detection grouping bolt*. Due to ***field grouping*** all detections from a single image are sent to a single worker where they are merged together into final result that will be returned to user.

sponse can be generated. We implement merging in the *Detection grouping bolt*, which is connected to the SVM-grouping bolt using a field grouping. We use field grouping to ensure all tuples belonging to the same request are always sent to the same worker (see, Fig. 3). Within this worker we save each detection $(c_{i,j}, score_{i,j}, r_{i,j})$ belonging to scale i and detected object j. To detect when all the tuples have been collected we rely on the numbers defining how many scales (max_scale) and how many detection $(num\_detections)$ were initially created. When all detections are collected we additionally run a non-maximum suppression to eliminate any multiple detections and send a tuple with meta-data and a list of detections to our final bolt. This process is described in Algorithm 6. The worker of the final bolt encapsulates the results and pushes it to the result queue where web-service is waiting. For the final two bolts we use 5 workers for each of them.

4 Performance

We evaluated our topology on a cluster of three high performance machines: one machine with 16 CPU cores and two machines with 32 CPU cores. Combined together we allocated 80 workers for the Storm topology. We generated one LHOP library with two visual categories selected from the *ETHZ Shape Classes* [3] dataset: apple logos and mugs.

The response time of our topology is composed of computation time of each individual bolt. Testing on a sample image of 750x560 pixels with 10 scales and 500 repeated requests we achieved average response time of around 45 seconds. To improve response time we also enabled parallelization of LHOP code using OpenMP [4]

[4] http://openmp.org

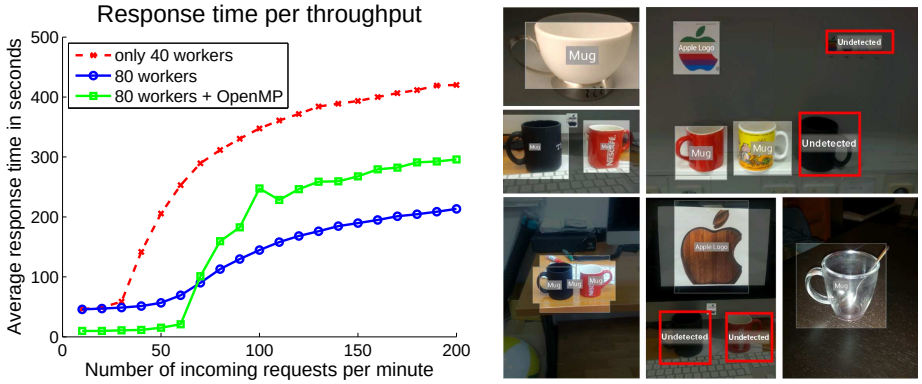

Fig. 4: Left: performance figures for three different configurations (i) 40 workers with disabled OpenMP, (ii) 80 workers with disabled OpenMP, (iii) 80 workers with enabled OpenMP. Right: examples of query images with correct (cropped gray) and missing (uncropped red) detection.

and utilized a multicore support of our processing units. This reduced processing time for LHOP by a factor of 4 and lowered the response time to around 10 seconds.

We computed average response time in relation to throughput of the topology using three different configurations: (i) using only 40 workers, (ii) using all 80 workers and (iii) using all 80 workers and with OpenMP enabled. The results of our test are shown in Fig. 4. The average response time of topology without OpenMP is slightly less than 50 seconds, while with OpenMP the response time lowers to around 10 seconds. This response time is maintained until requests cannot be processed at the faster rate than the rate at witch they arrive. With 40 workers this rate is around 30 requests per minute while with 80 workers the rate doubles to around 60 requests per minute until additional requests start to queue. Increasing the throughput by adding the hardware is an indication of efficient scalability of this system. We can also notice that OpenMP implementation is able to maintain response time of around 10 seconds for up to 60 requests per minute. After that, the additional requests require more running workers which compete with OpenMP parallel implementation for CPU time, therefore the response time increases even more than without OpenMP.

The max throughput of the system can be expressed with equation $\lambda_{max} = {}^{c \cdot m}/_{t_{res}}$, where t_{res} is response time for a single detection and m is a parallel factor defining number of parallel computation units. In our case we are using 80 workers for topology but workers are assigned for different tasks and are distributed unevenly. We added an adjustment factor c to the equation and can calculate $c = 0.5625$ based on values from graph: $m = 80$, $t_{res} = 45\,sec$ and $\lambda_{max} = {}^{60}/_{min}$. The relation between max throughput λ_{max} and number of workers m for this system can then be expressed as:

$$\lambda_{max} = \frac{0.5625 \cdot m}{45}, \tag{4}$$

from which we can estimate the parallelization to satisfy specific traffic requirements, e.g. 800 workers would be needed for throughput of 10 requests per second.

5 Conclusion

In this paper we presented an implementation of visual object detection on a distributed processing platform. We implemented an hierarchical model, specifically learnt-hierarchy-of-parts [5], for localization and used Support Vector Machine with HoC descriptor [8] as verification of each detection. We presented our implementation as a Storm application and provided an evaluation on a cluster of 80 workers. We have achieved response time of around 45 seconds using single core in the LHOP code and around 10 seconds with multicore processing enabled in the LHOP code. We have also shown ability to handle 60 requests per minute without considerable increase of response time and demonstrated a scalability of our system with the increase of processing power.

The achieved response time of 10 seconds can still be a limiting factor for many web-service applications and will therefore be the focus of further optimization in our future work. As the main bottleneck is LHOP processing we will focus on optimizing this bolt even further. As our future work we also plan to implement a content-based image search which will be connected with this service to provide category information and improve retrieval of relevant images.

Acknowledgments. This work was supported in part by ARRS research program P2-0214 and ARRS research projects J2-4284, J2-3607 and J2-2221.

References

1. C. C. Chang and C. J. Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.
2. P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on PAMI*, 32:1627–1645, 2010.
3. V. Ferrari, T. Tuytelaars, and L. V. Gool. Object detection by contour segment networks. In *Proceeding of the ECCV*, volume 3953 of *LNCS*, pages 14–28. Elsevier, June 2006.
4. S. Fidler, M. Boben, and A. Leonardis. Evaluating multi-class learning strategies in a generative hierarchical framework for object detection. In *NIPS*, 2009.
5. S. Fidler and A. Leonardis. Towards scalable representations of object categories: Learning a hierarchy of parts. In *CVPR*. IEEE Computer Society, 2007.
6. I. Kokkinos and A. Yuille. Inference and learning with hierarchical shape models. *Int. J. Comput. Vision*, 93(2):201–225, June 2011.
7. C. H. Lampert, M. B. Blaschko, and T. Hofmann. Beyond sliding windows: Object localization by efficient subwindow search. In *CVPR*, pages 1–8, 2008.
8. D. Tabernik, M. Kristan, M. Boben, and A. Leonardis. Learning statistically relevant edge structure improves low-level visual descriptors. In *ICPR*, 2012.
9. D. Tabernik, L. Čehovin, M. Kristan, M. Boben, and A. Leonardis. Vicos eye - a webservice for visual object categorization. In *Proc. of the 18th Computer Vision Winter Workshop*, 2013.
10. C. Wojek, G. Dorkó, A. Schulz, and B. Schiele. Sliding-windows for rapid object class localization: A parallel technique. In *Proceedings of the 30th DAGM symposium on Pattern Recognition*, pages 71–81, Berlin, Heidelberg, 2008. Springer-Verlag.
11. L. Zhu, Y. Chen, A. Torralba, W. Freeman, and A. Yuille. Part and appearance sharing: Recursive compositional models for multi-view. In *CVPR*, pages 1919 –1926, june 2010.