ViCoS Eye - a webservice for visual object categorization

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,luka.cehovin,matej.kristan,marko.boben,ales.leonardis}@fri.uni-lj.si

Abstract. *In our paper we present an architecture for a system capable of providing back-end support for web-service by running a variety of computer vision algorithms distributed across a cluster of machines. We divide the architecture into learning, real-time processing and a request handling for web-service. We implement learning in MapReduce domain with Hadoop jobs, while we implement real-time processing as a Storm application. An additional website and Android application front-end are implemented as part of web-service to provide user interface. We evaluate the system on our own cluster and show that the system running on a cluster of our size can learn Caltech-101 dataset in 40 minutes while real-time processing can achieve response time of 2 seconds, which is adequate for multitude of online applications.*

1. Introduction

Increased processing power behind server based computers has in the recent years enabled many online services to offload their processing into a cloud-based computing. This has also become beneficial for computer vision problems where many computationally expensive algorithms are already enabling services such as TinEye[1], Macroglossa[2], Google Image Search[3] or Google Goggles[4]. These online services work particularly well for images that the system has previously seen on the internet but do not perform very well on previously unseen images. For instance, the TinEye service does not perform any object recognition[5], while querying Google Image Search with an camera snapshot of a simple coffee mug or a chair produces results where there is no mug or chair in any of the first 50 hits (see, Fig. 2). The only visual similarity between the query image and the results is a similar color distribution.

Adding more advanced computer vision algorithms such as object categorization would be highly beneficial as it could provide more information on objects that have not been previously seen but belong to a known category of objects. This would in turn open up the service for multitude of applications. For instance, the service could provide a cloud-based machine vision for robotic systems, or it could be used as a back-end for an application that is capable of poisonous plant or dangerous animal identification.

Most of the existing online computer vision services provide only simple image similarity searches. We show in this paper how advanced computer vision algorithms can be implemented as a cloud-based application to provide an online service. Simple offloading of computer

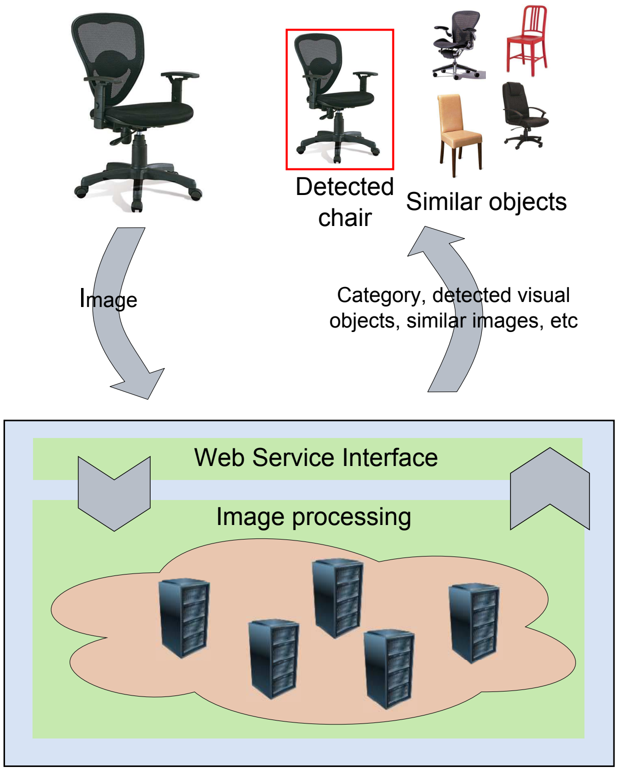

Figure 1. Conceptual overview of online computer vision service. Service takes a single image as a query, relays it for processing on cluster of machines and returns a result in a form of recognized category, detected objects, similar images, etc.

[1]http://www.tineye.com
[2]http://www.macroglossa.com
[3]http://images.google.com/
[4]http://www.google.com/mobile/goggles
[5]http://www.tineye.com/faq#similar

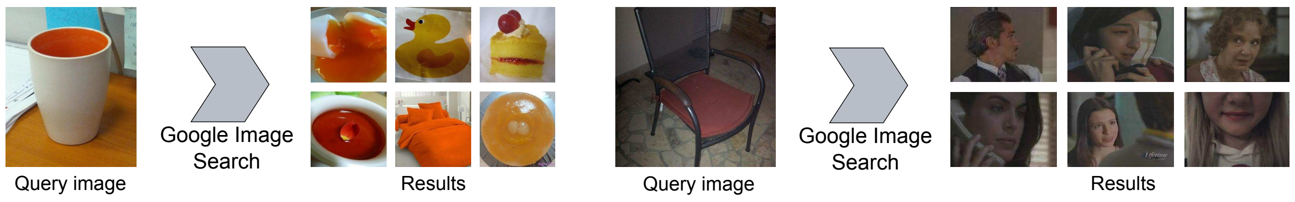

Figure 2. Two examples of Google Image Search on unknown images. Returned similar images match only based on color distribution but ignore any other information such as visual category.

vision processing to a single server has already been done [5, 7], but they implemented the algorithm on a single server which does not scale well and cannot be directly distributed across many servers. In our paper we therefore focus primarily on describing a system that can efficiently run computer vision algorithms on a cluster of machines. We analyze requirements for such a system for providing online service, describe our implementation and provide some performance analysis. The architecture of the system we are describing is general enough to allow for implementation of different kinds of computer vision algorithms, thus providing variety of services such as: detection of objects in images, recognition and identification of different visual categories or content based image search, not just based on color distribution but based on objects and visual categories identified in the query image. In our paper we demonstrate this service on a problem of object categorization using HoC descriptor [8].

This paper is structured as follows. In Section 2 we state the requirements that a computer vision algorithm must meet, the architecture is presented in Section 3 and the implemented computer vision algorithm is presented in Section 4. In Sections 5 and 6 we analyze in more detail how different back-end aspects of the service can be implemented and provide a general implementation details on user interface in Section 7. In Section 8 we provide some performance analysis of implemented service and conclude the paper in Section 9.

2. Requirements

We first present the following requirements for our system:

- providing an online service (a web-service) for computer vision algorithms,
- capability of distributed processing in a cluster of machines (a cloud),
- ability to handle hundreds of requests per second.

Running a system on a cluster of machines requires that all the implemented algorithms run distributed in order to utilize the resources as efficiently as possible. By distributing the processing across a cluster we also enable efficient scalability. Adding new machines to the cluster should be a straightforward process and should automatically provide higher throughput to handle more and more requests easily as the service expands.

Additionally, the system has to function as a service. This requires the system to handle requests that come either directly from the internet or from any underlying system using this service. Each request comes in a form of an image and the system has to be able to process it with computer vision algorithm and return a result in the form of a category or objects detected in the image or similar images. A general overview is depicted in Fig. 1.

As a service, the system also has to enable handling hundreds of requests simultaneously and process them sufficiently fast. This puts additional constraints on the algorithms that we can use. In general, the recommended tolerable waiting time (TWT) for web pages is approximately 2 seconds [6], but that number can vary for different applications. In our case, any algorithm capable of processing the image in up to five seconds or less should be acceptable for a multitude of online applications, while any further processing delay might deter the user from using this service.

3. Architecture

Many computer vision algorithms are divided into two stages: (i) learning and (ii) testing/classifying. We therefore implement two different subsystems. The first subsystem corresponds to the learning stage. As we also have to meet the required resource efficiency utilization by distributing the processing across different machines in the cluster setup, we term this subsystem as ***distributed learning***. The second subsystem provides a testing/classifying stage. This subsystem is also the main part of the system that has to be connected to the input of our web-service and in real-time provide results on each query requested by the user. As such we call this subsystem ***real-time stream processing***. We also implement a third subsystem called ***Web Service Interface*** that provides service API over the internet.

All subsystems have completely different assignments within the framework and are therefore implemented using different techniques. ***Distributed learning*** has to process relatively high number of training images (from thousands of images and upwards) therefore this process can be best distributed across a cluster by transforming it into a *MapReduce* [3] problem. In this domain the problem is represented by a set of input items which are processed with a *Map* and a *Reduce* function. By transforming the algorithm to a *MapReduce* domain a problem now becomes relatively straightforward to be distributed across

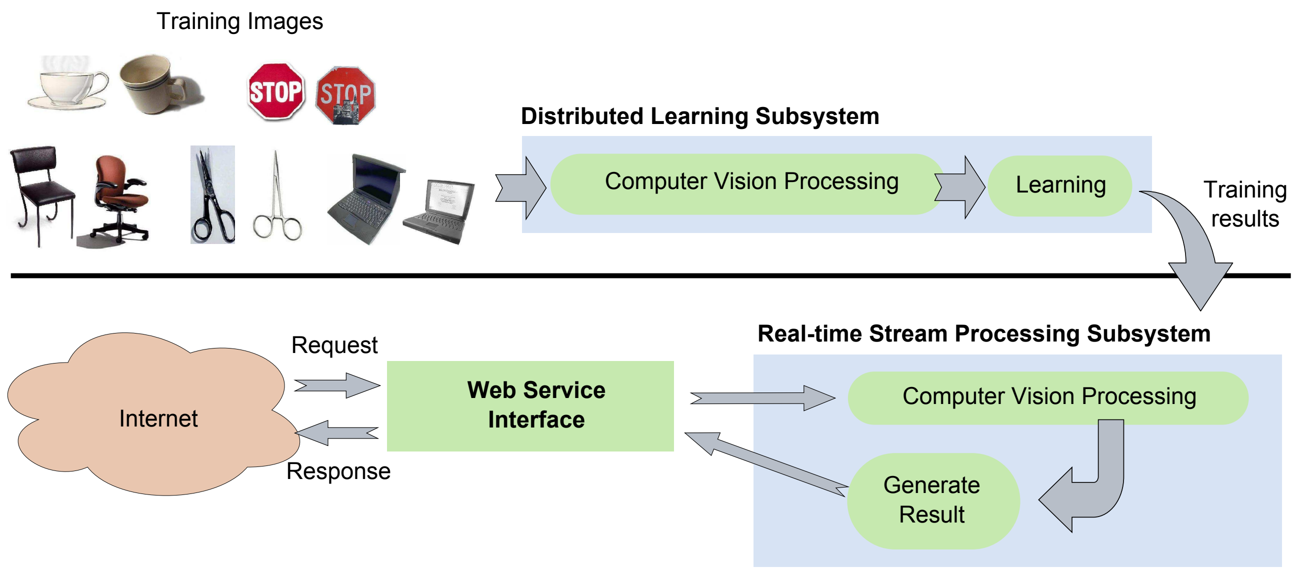
Training Images
Distributed Learning Subsystem
Computer Vision Processing
Learning
Training results
Real-time Stream Processing Subsystem
Request
Internet
Web Service Interface
Response
Computer Vision Processing
Generate Result

Figure 3. Architecture of the system providing a computer vision service. Service is divided into learning subsystem handling any learning and training of library, models, etc, and into real-time processing subsystem handling incoming request and generating results.

cluster machines, provided that each item can be processed with *Map* and *Reduce* function independently of other items. In our case we treat each training image as a single item. Any part of a computer vision algorithm that can process images independently can be trivially transformed into this domain. More complex parts would have to be re-factored in order to efficiently run on a cluster of machines.

Real-time stream processing must be implemented using different technique as its job and requirements are completely different. This subsystem is directly connected to the web-service and must serve hundreds of requests and return corresponding results in near real-time. For each query this system has to process a single image. Transforming this subsystem into MapReduce domain would not be possible since that system works as a batch processing of jobs, where all input data is known in advance. But in this subsystem we have to continuously handle requests that are coming into the system and therefore we do not know in advance how many request we will have. The best way to efficiently run computer vision algorithm on a cluster of machines would be to assign each request to a single machine for processing.

Web Service Interface subsystem provides API for access to the service and additional user interface in a form of a web site and an Android application. The subsystem also has to enable communication with *Real-time stream processing* subsystem. *Web Service Interface* has to work as a relay between the user and *Real-time stream processing* by forwarding the request from user for further processing and retrieving the results, which are then relayed back to the user. Graphical representation of all three subsystems can be seen in Fig. 3.

4. HoC-based object categorization

In general, the algorithm for image processing can be a multitude of different computer vision algorithms but in this paper we demonstrate the system on the problem of multiclass object categorization. For object categorization we use the LHOP [4] based HoC [8] descriptor with an SVM [2] classifier. The advantage of using this algorithm is LHOP's efficient inference. As the service has to enable quick response to user's query, this method with its efficient inference allows us to quickly process the image and generate the HoC descriptor. The descriptor based approach is used on the whole image. Therefore our system provides only object categorization service and not object localization or detection. In this section we briefly describe the process of categorization using HoC descriptor and refer the reader to [8] for further details.

Histogram of Compositions is a shape-based descriptor which uses learning of shapes to find only shapes and structures that are most relevant for object description. Relevant shape fragments identification is achieved by the learning process of learnt-hierarchy-of-parts [4] (LHOP) model, which produces a vocabulary of hierarchical compositions. This compositions are then further used in the process of constructing the HoC descriptor.

Constructing a HoC descriptor $\mathcal{H}$ from an image is a two step process. In the first step, image is processed with a previously learnt LHOP library $\mathcal{L}$ to produce a list of compositions $\{\pi_k\}_{k=1:K}$. Since LHOP represents shapes using hierarchical compositions we produce a list of compositions for each layer of hierarchy. Descriptor $\mathcal{H}$ is then constructed in second step from compositions of the desired layer(s) $\{\pi_k \mid \pi_k \in \text{selected layers}\}$. This process applies partitioning scheme of M-regions and creates small histograms of compositions $\mathcal{H}_m$ of each region that form a part of a final descriptor $\mathcal{H} = \alpha[\mathcal{H}_1, ...\mathcal{H}_M]$. This descriptor is then further used in an SVM to perform the final step in the object categorization.

5. Distributed learning subsystem

In this section we detail implementation of distributed learning for object categorization using HoC descriptor. We define the process of learning as follows:

Input: $\mathcal{L}$ is a pre-learnt LHOP library and $\{(\mathcal{I}_i, c_i)\}_{i=1:I}$ is a set of training images $\mathcal{I}_i$ each annotated with a proper visual category name $c_i \in \mathcal{C}$.

Output: $\{m_j\}_{j=1:J}$ is a set of SVM models representing trained model m_j for each different category found in the set of category names $\mathcal{C}$.

Algorithm steps:

1. Process each image $\mathcal{I}_i$ from the set of training images $(\mathcal{I}_i, c_i)$ with an LHOP model to produce compositions π_k for the whole hierarchy:
$$\mathcal{P}(\mathcal{I}_i, \mathcal{L}) = \{\pi_k\}_{k=1:K_i}.$$
2. Generate HoC descriptor $\mathcal{H}_i$ for each image $\mathcal{I}_i$ processed with an LHOP model that produced the list of compositions $\{\pi_k\}_{k=1:K_i}$.
3. Group HoC descriptors $\mathcal{H}_i$ based on its category label c_i and train SVM model m_j for each grouped visual category c_j:
$$svm(\{\mathcal{H}_i\}_{i=1:I}, c_j) = m_j.$$

We use our implementation of LHOP [4] and HoC [8], and LIBSVM [2] for support vector machine.

Before translating our problem into MapReduce domain we first provide basic MapReduce notation and refer the reader to [3] for more detail. Input for a MapReduce problem is always an array of key-value pairs $\{(\kappa_l^{input}, \omega_l^{input})\}_{l=1:L}$ (shown as (a) in Fig. 4) that gets processed with two different functions. First with a map function $\mathcal{M}$ to produce (b) intermediate result of a key-value pair :

$$\mathcal{M} : (\kappa_l^{input}, \omega_l^{input}) \mapsto (\kappa_l^{inter}, \omega_l^{inter}).$$

All intermediate pairs are then grouped by their keys into (c) N different groups $\{(\kappa_n^{inter}, \{\omega_{o,n}^{inter}\}_{o=1:O_n})\}_{n=1:N}$. Each group with the same key is then further processed by a reduce function $\mathcal{R}$ that returns (d) final key-value pairs for each group with the same key:

$$\mathcal{R} : (\kappa_n^{inter}, \{\omega_{o,n}^{inter}\}_{o=1:O_n}) \mapsto (\kappa_n^{output}, \omega_n^{output}).$$

A simple graph of this process is depicted in Fig. 4. We can now transform each step of our object categorization procedure into a separate MapReduce problem and connect them together by chaining output of each step into the input of the next step.

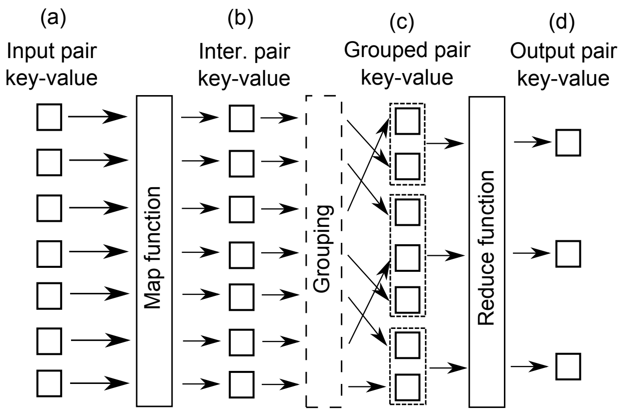

Figure 4. MapReduce processing pattern. Set of input pairs (a) gets processed in parallel by Map function then this intermediate result (b) gets grouped by its keys and each group (c) gets finally processed in parallel by Reduce function to produce final output (d).

5.1. Object categorization learning as MapReduce

In the first step we have input in a form of list of training images with category labels, which we now represent as MapReduce input pair with empty key and value as image with category:

$$\kappa_i^{input\_LHOP} = \emptyset,$$
$$\omega_i^{input\_LHOP} = (\mathcal{I}_i, c_i).$$

Processing with LHOP library can then be easily represented as a MapReduce map function $\mathcal{M}^{LHOP}((\kappa_i^{input}, \omega_i^{input})) = \mathcal{P}(\mathcal{I}_i, \mathcal{L})$, while reduce function $\mathcal{R}^{LHOP}$ is not needed in this case. Intermediate result of map function is directly result of MapReduce process for this step which in this case is represented as empty key and list of compositions with category name as value:

$$\kappa_i^{output\_LHOP} = \emptyset,$$
$$\omega_i^{output\_LHOP} = (\{\pi_k\}_{k=1:K_i}, c_i).$$

The second step is transformed in a similar manner. We first chain output of previous MapReduce step directly into input pair of current step:

$$\kappa_i^{input\_HoC} = \emptyset,$$
$$\omega_i^{input\_HoC} = \omega_i^{output\_LHOP} = (\{\pi_k\}_{k=1:K_i}, c_i).$$

Map function $\mathcal{M}^{HoC}$ in this case generates HoC descriptor, while reduce function $\mathcal{R}^{HoC}$ is not needed, so intermediate output of map directly becomes output of the current MapReduce step, in which we return an empty key and HoC descriptor with category label as value:

$$\kappa_i^{output\_HoC} = \emptyset,$$
$$\omega_i^{output\_HoC} = (\mathcal{H}_i, c_i).$$

For the last step, which is SVM training, we need to distribute processing not by images, but by all the possible categories as training of a single category will require

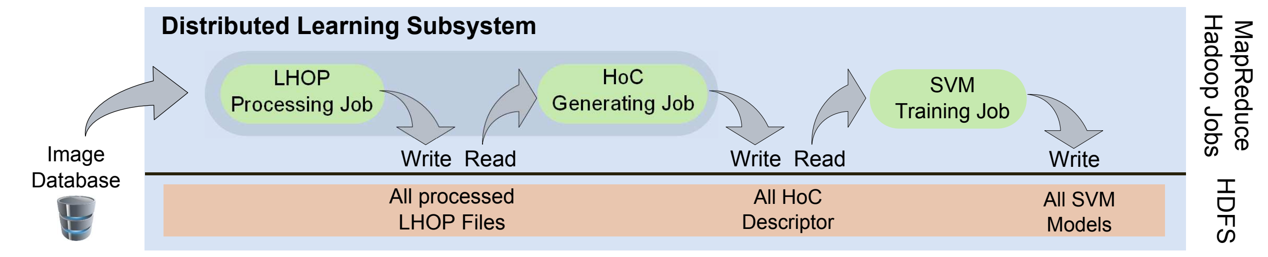

Figure 5. The process of learning visual categories using HoC descriptor and support vector machine. In general process is composed out of three different Hadoop jobs: LHOP processing, HoC generating and SVM training. We merge first two steps to avoid costly read/write to HDFS.

descriptors from all training images. However we can still train a single category independently of other categories. Additionally, we leverage the number of cluster machines to implement a simple grid optimization of SVM parameters. We achieve this by adding additional input pairs for each parameter configuration we check. We can now write the last step as MapReduce problem with input pair as category name for key, and HoC descriptor of all training images together with specific SVM parameter values for the input value

$$\begin{aligned}\kappa_j^{output\_SVM} &= c_j,\\ \omega_j^{output\_SVM} &= (\{\mathcal{H}_i\}_{i=1:I}, svm\_params).\end{aligned}$$

Map function is then implemented as an SVM training $\mathcal{M}^{SVM} = \text{svm}(\{\mathcal{H}_i\}_{i=1:I}, c_j, svm\_params)$ which returns category name for key, and a trained SVM model with some performance metric p for value:

$$\begin{aligned}\kappa_j^{inter\_SVM} &= c_j,\\ \omega_j^{inter\_SVM} &= (m_j, p).\end{aligned}$$

The reduce is then implemented as a *max* function across all the trained models for specific category:

$$\mathcal{R}^{SVM}(c_j, \{(m_o, p_o)\}_{o=1:O_j}) = (c_j, \underset{p_o}{argmax}(\{(m_o, p_o)\})).$$

The result of reduce function is then a set of pairs with category names and appropriate best-performing SVM model (c_j, m_j).

5.2. Hadoop as MapReduce implementation

There are many available implementations of MapReduce. We have chosen Apache Hadoop [9] which is an open source implementation written in Java. It is intended for processing of BigData (tens of Terabytes of data) as a batch of jobs and is easily scalable up to 1000 or more nodes (machines). Hadoop also comes with its own version of distributed file system called Hadoop Distributed File System (HDFS), which holds all the input and output data as well as any intermediate results.

Distributed learning subsystem can now be implemented as batch of three different Hadoop *job jars* that run in a sequence. In the first job we read images from the database, process them with LHOP and save results onto HDFS file-system. The second job waits for all the images to be processed, then reads them from HDFS, generates appropriate HoC descriptor and saves them back to HDFS. The final job waits until all the HoC descriptors are generated, reads them from HDFS and generates appropriate number of SVM problems for each category which are then trained on the cluster. The final job creates SVM models and sends them to *real-time stream processing subsystem*. This sequence of jobs is depicted in Fig. 5.

We noticed that, between each job, we write results in HDFS file-system and then read them again for the next job. This presents a possible bottleneck as access to HDFS can be expensive. Therefore we additionally optimize the sequence of jobs by merging the first and the second job together. In this way we generate HoC descriptor immediately from the image processed with an LHOP model that is still loaded in memory and can therefore avoid writing and reading results of the first job. Since HoC descriptor can now be generated immediately after the image is processed, we also eliminate any delay between the first and the second job and thus optimize the process even further.

6. Real-time stream processing subsystem

In this section we present a subsystem capable of handling requests from the web-service, process them with object categorization algorithm and provide appropriate response back to the requester. We define the algorithm required for this subsystem as follows:

Input: $\mathcal{L}$ is a pre-learnt LHOP library, $\{(c_j, m_j)\}_{j=1:J}$ is a set of trained SVM models and $(r_i, \mathcal{I}_i)$ is a request with an image $\mathcal{I}_i$ and request information r_i.

Output: (r_i, c_i, p_i) is a response to request r_i with an image classification category c_i and SVM score p_i.

Algorithm steps:

1. Process request image $\mathcal{I}_i$ with an LHOP model to produce compositions π_k for the whole hierarchy:
$$\mathcal{P}(\mathcal{I}_i, \mathcal{L}) = \{\pi_k\}_{k=1:K_i}.$$
2. Generate HoC descriptor $\mathcal{H}_i$ from image $\mathcal{I}_i$ processed with an LHOP model that produced list of compositions $\{\pi_k\}_{k=1:K_i}$.

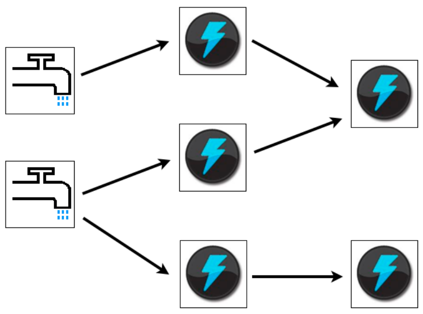

Figure 6. Example of Storm topology with two spouts connected to five bolts.

3. Classify HoC descriptor $\mathcal{H}_i$ into a number of pre-trained categories using SVM. We select category with the best score p_i:

$$c_i = \underset{(c_j,m_j)}{argmax}\{p_j = svm(\mathcal{H}_i, m_j);\ p_j > 0\},$$

   if any score is higher then 0 or return empty category otherwise. We return original request r_i and category classification c_i with its score p_i as result.

Implementing this kind of a system would not be possible with the Hadoop in MapReduce domain since in that system we have to know in advance the number of input items to process in order to split them in the most efficient way for processing across different cluster machines. While in our case input data would be unpredictably coming into the system based on user requests. Therefore the system does not know in advance how many requests we will have. The best way to implement this kind of request handling would be with a system of workers and queues distributed across different machines where each worker could then accept the workload as it arrives into the system. To implement this design we use Storm[6] which is an open source distributed real-time computation system. It provides all the necessary infrastructure of workers and queues distributed across cluster machines and provides convenient way for implementing applications on top of this system.

Writing application on top of Storm requires to define a topology. Topology is a digraph where nodes are processing elements and are implemented either as *spout* or *bolt*, and directed edges represent the direction of processed data. A general example of Storm topology is shown in Fig. 6. The work in this topology starts with a *spout*, which can be hooked to the outside world and would be generating (emitting) new data streams. A stream of data would then be sent to the appropriate neighboring *bolts*. Each *bolt* then processes the data and can emit its results as one or more new streams to its neighboring nodes. The process continues until no more new data is emitted. Processing in each node is handled by Storm and gets distributed across the cluster according to availability of workers therefore allowing for best utilization of resources.

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

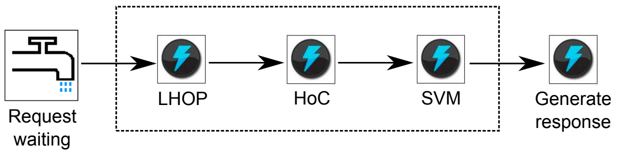

Figure 7. Storm topology for object categorization processing using HoC descriptor and support vector machine.

6.1. Real-time image categorization service as Storm topology

We define our system of image categorization service as Storm application by providing appropriate topology of spouts and bolts. In our case we define one bolt for each step of the algorithm and simply connect them in a sequence. Therefore a single bolt implementation represents LHOP processing, one bolt represents generating HoC descriptor and additional bolt represents classifying with the trained SVM models. We also need to define one spout which will wait for input requests and send appropriate stream of data to the first bolt. Also additional bolt has to be added at the end of the stream that will send response in a form of a category name back to the original caller. A graphical depiction of this topology can be seen in Fig. 7.

The above topology allows for handling of each request by multiple workers across different machines as each bolt can be processed simultaneously. But due to sequential processing we still have response time that is the sum of each individual bolt processing time:

$$t_{response} = t_{LHOP} + t_{HoC} + J \cdot t_{SVM} + \bar{t},$$

where J is the number of categories and $\bar{t}$ is the additional time needed for communication between different bolts and also between user and web-service.

Note, that we have implemented SVM classification as a single bolt worker. This may not appear optimal at first since we need to test descriptor against multiple categories which can be done in parallel. Thus implementing SVM classification of each individual category as single bolt might provide better results. In practice however, the time needed for classification of all categories (100 categories in our case) in a single bolt was only 25% of the response time with the other 75% representing the time required for LHOP processing. Performance benefits of using multiple bolts might become more noticeable with higher number of categories but even in that case it would still be more reasonable to handle a subset of categories at once to avoid any additional delay of sending hundreds of streams across the cluster.

7. Web service interface

As our system is intended to work as a web-accessible service, we have implemented a simple web interface that

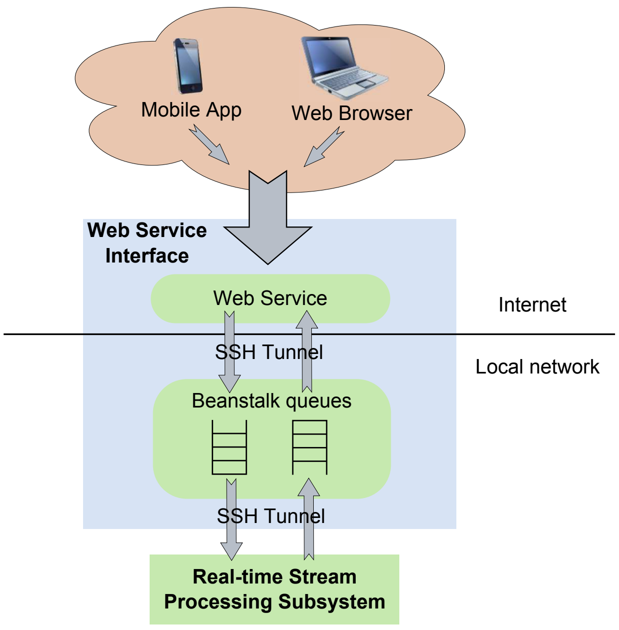

Figure 8. Web Service Interface architecture.

handles user-interaction, relays the requests to the back-end and displays the results to the user after the request has been processed by the service.

7.1. Web site

The main component of our interface is a web-site written in Python and JavaScript. The Python part accepts requests, validates input images (size and type) on the server, equips them with additional meta-data and pushes jobs to a Beanstalk[7] incoming queue. Once the job is processed, the results are inserted to a database by a daemon process where they are available to the service. The JavaScript part of the interface handles smooth transitions of the browser interface.

As security of the system is important the computational back-end is located on the local network and is connected to the public web server that runs the interface front-end using a persistent encrypted tunnel. This relation is illustrated in Fig. 8.

7.2. Android front-end

In our internal user-experience study we have found that a classical web-interface could be improved by allowing user to quickly capture and submit new images as queries. As nearly all mobile smart-phones and tablets now days contain a high-resolution camera, we have created a prototype Android application to streamline the process. The application enables users to capture images, uploads them to the web-service and displays the result.

8. Performance

Evaluation of our system was performed by implementing both subsystems on a cluster of three machines

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

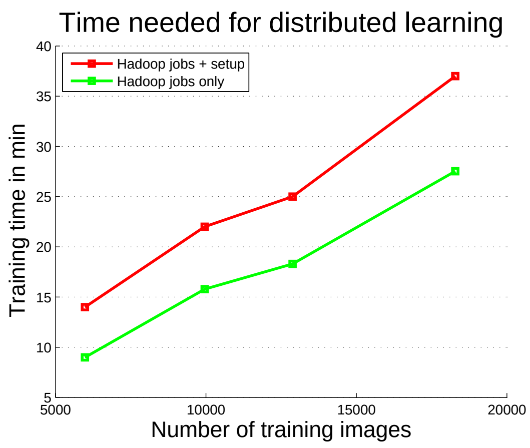

Figure 9. Time in minutes needed for distributed learning relative to the number of training images. The red line represents all time needed for training including Hadoop jobs with any additional time for setup preparation, while the green line represents only time needed for Hadoop jobs.

t_{LHOP}	t_{HoC}	$J \cdot t_{SVM}$	$\bar{t}$	$t_{response}$
1547 ms	85 ms	462 ms	48 ms	***2144 ms***

Table 1. Response time of object categorization service using Storm processing.

each with more than 30 CPU cores and more then 80 GB of memory. For Hadoop system we assigned 35 slots on each machine which allows simultaneous execution of 105 Map or Reduce functions. The distributed learning was preformed on all 9124 images of Caltech-101 dataset with 103 categories (with background images and separate faces and faces_easy category). We also added mirror image for each training example. This produced 18248 different training examples. We evaluated the performance of distributed learning subsystem by varying the number of training examples and timing each Hadoop job with any additional time for framework setup. The results are shown in Fig. 9. We were able to learn all 103 categories with 18248 images in less then 40 minutes but there is also some room for improvements as Hadoop jobs took less then 30 minutes. Based on the observed numbers we can see that we can easily learn Caltech-101 in less then an hour and since learning could be done only occasionally (depending on newly gathered image dataset from crawled web pages or user feedback once or twice a week could be enough) we could easily scale our system to hundred of thousands or even millions of training images as this could be processed in a day or two.

Evaluation of real-time stream processing subsystem was performed on a single machine with 8 CPU cores where we assigned 4 slots for Storm task processing. We evaluate only response time of a single query and provide theoretical extrapolation for max server load. Results of the evaluation can be seen in Table 1. We tested the server with 20 different images and provide an average time for each component. Assuming we can use the same cluster as for Hadoop (105 nodes) we can calculate the maximum number of incoming requests that we can handle. Using queueing theory [1] we can define our Storm application service as M/M/105 queue model, where arrival rate λ is according to a Poisson process and service times have an exponential distribution with parameter μ. Based on observed averaged processing time $t_{response} = 2.144\ sec$ we can set the service time $\mu = 0.4664\ sec^{-1}$ and calculate the maximum arrival rate before the system's queue grows to infinity (i.e. when $\rho > 1$):

$$\rho = \frac{\lambda}{m \cdot \mu,}$$

where $m = 105$ is number of processing servers. We calculate the max arrival rate $\lambda_{max} = m \cdot \mu = 48.9720$, which tells us that our service would be able to handle 48 requests per second before the queue would start to grow to infinity. The observed response time and a calculation of handling 48 requests per second in a cluster of 105 nodes could effectively allow us to use the system for multitude of online services. Scaling to any more traffic or requests would only require additional computing power which can be added trivially even without stopping the existing service thanks to flexibility of the Storm and Hadoop systems.

9. Conclusion

In this paper we presented implementation of a computer vision algorithm as an online web-service and analyzed the efficiency of implementation on a distributed processing platform. In particular we implemented a service for object categorization using HoC descriptor [8] and a support vector machine [2]. We divided the architecture of the system into two stages: distributed learning and real-time stream processing. The first part was responsible for the learning and was implemented in MapReduce [3] domain as a Hadoop job. While the second part, real-time stream processing, was responsible for handling online service requests and was implemented as a Storm application. An additional part of the system was also the web-service interface that acted as an relay between users and distributed processing platform. We implemented it as an web site and also as an Android application.

We analyzed the performance of each stage on a cluster with more then 100 CPU cores. The system was able to complete the learning phase in less than an hour for Caltech-101 dataset with around 18000 training images and more then 100 categories. Accounting for infrequent, runs and based on performance analysis, we have shown that such system should be able to scale to hundreds of thousands or even millions of training images. The analysis of the second part, the real-time stream processing, has shown that our service has response time of around 2 seconds which is completely adequate for multitude of online applications. We also showed that the system deployed on a cluster of 105 nodes should be able to handle 48 requests per second and we could easily scale to higher number by adding additional machines to the cluster.

Note, that the presented system can potentially be used not only as an online computer vision service but by adding user feedback to the web interface we could use the system as image annotation tool. For instance, a user might be able to provide a feedback for each missing object or improperly classified category. This data can then be collected by the system and used to generate new dataset of training images. This dataset could augment the existing training images and together with retraining enable better performance of the whole service. We would like to implement this extension in our future work together with some other small optimizations. Additionally, we would also like to extend the system from only object categorization to object detection and localization. In our future work we will use the system for implementing content-based image search that would take into account information about object categories found in the image.

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] A. O. Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[2] 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.

[3] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[4] S. Fidler and A. Leonardis. Towards scalable representations of object categories: Learning a hierarchy of parts. In *CVPR*. IEEE Computer Society, 2007.

[5] S. S. Kumar, M. Sun, and S. Savarese. Mobile object detection through client-server based vote transfer. In *CVPR*, 2012.

[6] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & IT*, 23(3):153–163, 2004.

[7] D. Omerčević and A. Leonardis. Hyperlinking reality via camera phones. *Machine Vision and Applications*, 22(3):521–534, 2011.

[8] D. Tabernik, M. Kristan, M. Boben, and A. Leonardis. Learning statistically relevant edge structure improves low-level visual descriptors. In *International Conference on Pattern Recognition*, 2012.

[9] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.