Large-Scale Human Action Recognition with Spark

Hanli Wang #5, Xiaobin Zheng #2, Bo Xiao #3

# Department of Computer Science and Technology, Tongji University, Shanghai, China
§ Key Laboratory of Embedded System and Service Computing, Ministry of Education, Tongji University, Shanghai, China
1 hanliwang@tongji.edu.cn, 2 xiaobin@tongji.edu.cn, 3 lin@tongji.edu.cn

Abstract—In this paper, Apache Spark, the rising big data processing tool with in-memory computing ability, is explored to address the task of large-scale human action recognition. To achieve this, several advanced key techniques for human action recognition, such as trajectory based feature extraction, Gaussian Mixture Model, Fisher Vector, etc., are realized with parallel distributed computing power on Spark. The theory and implementation details for these distributed applications are presented in this work. The experimental results on the benchmark human action dataset Hollywood-2 show that the proposed Spark based framework which is deployed on a 9-node computer cluster can deal with large-scale video data and can dramatically accelerate the process of human action recognition.

I. INTRODUCTION

The rapid development of Internet and intelligent mobile devices has already led to a large amount of data generated over the past two decades, and multimedia data such as videos accounts for a big percentage of this deluge of data. In order to organize these videos and provide value-added services to users, it is worth understanding human activities (also known as human action recognition) automatically from videos, with a number of researches on this topic such as [1], [2], [3], to name a few. When the amount of videos to be analyzed becomes larger and larger, the challenges including efficiency and accuracy arise from the huge computational demands produced by large-scale human action recognition.

To meet the escalating computational demands, the development of distributed applications deployed on computer clusters has become much productivity since a wide range of programming paradigms, middle-wares and frameworks are developed. MapReduce [4] is such a popular programming paradigm proposed by Google, which applies the intuition that many large-scale workloads can be scaled out horizontally and expressed with map and reduce operations. Hadoop [5] is the most representative open source cloud computing implementation based on MapReduce, which offers locality-aware scheduling, fault-tolerance as well as load balancing. However, the acyclic data flow model employed by Hadoop shows poor performance on iterative tasks which are common when dealing with multimedia mining tasks.

Recently, Spark [6], an efficient cluster computing system which supports in-memory computing capability, becomes an alternative choice to improve the performance of iterative and interactive computations. Spark uses an abstraction called Resilient Distributed Dataset (RDD) [7] to represent a read-only collection of objects partitioned across a set of machines, which is able to rebuild lost partitions to make Spark fault-tolerant. Moreover, Spark not only supports map and reduce operations but also provides a variety of common operations like sample, union and join.

As a subproject of Spark, MLlib [8] offers a number of common machine learning algorithms and utilities. Besides MLlib, some researches have been presented to focus on the machine learning problems with Spark. Lin et al. [9] propose a distributed Newton method for solving logistic regression as well as linear Support Vector Machine learning with Spark. In [10], an optimized scalable Principle Component Analysis (PCA) algorithm is implemented with Spark. In [11], a framework, namely YAFIM, is proposed to parallelize frequent itemset mining with Spark and the experiments show that YAFIM achieves an 18× speedup in average for various benchmarks as compared with MapReduce.

In this work, a Spark based framework for human action recognition is proposed to speed up several key processes including trajectory based feature extraction, Gaussian Mixture Model (GMM) generation and Fisher Vector (FV) encoding [12]. To the best of our knowledge, it is the first work to employ Spark to design FV based human action recognition systems. The rest of this paper is organized as follows. Section II introduces the theoretical background about Spark, feature extraction, GMM and FV, which is useful to elicit the design concept and implementation of the proposed Spark based framework for human action recognition, which will be detailed in Section III. The experimental results are presented in Section IV to demonstrate the performance of the proposed framework. Finally, Section V concludes this paper.

II. THEORETICAL BACKGROUND

A. Apache Spark

Apache Spark [6] is a general and scalable processing system to process large-scale data. Unlike Hadoop [5] which is a disk based distributed computing system with the MapReduce [4] model, Spark provides memory based computation abilities making iterative computing tasks more efficient. Several key issues about Spark are given below.

1) Resilient Distributed Dataset (RDD); RDD [7] is the key abstraction of distributed data used to achieve in-memory computation ability. Specifically, when a dataset is loaded into
a Spark cluster, partitions of this dataset will be cached into slave nodes according to the storage level\(^1\). The information of all the loaded data partitions is represented by an RDD. Operations provided by RDD’s high-level APIs can then be performed in parallel on partitions within each slave node.

2) **RDD Operation and Shared Variable**: Transformation and action are two types of operations that can be acted on RDD. Transformations create new RDD from an existing one, e.g., map and sample, and actions perform computations on RDD and return a value to the master, e.g., reduce and collect. Similar to Hadoop’s distributed cache [13], Spark also allows users to distribute a read-only variable cached on each slave node to avoid repeating shipping a copy for all tasks.

3) **Fault Tolerance**: Spark records transformation chain by a lineage graph. If any partition of an RDD is lost during job execution, the corresponding RDD can rebuild that lost partition from the parent RDD according to the lineage graph. Since RDD is read-only, any recomputation from the parent RDD can remain the same results.

**B. Trajectory Based Feature Extraction**

Owing to the ability of capturing local motion information of videos, trajectory based approaches are shown to be efficient in representing video features [2]. In general, there can be two basic questions about tracking motion trajectories: (1) how to select the tracking points and (2) how to track the sampling points by a certain strategy. To address these issues, a number of approaches have been designed, such as [2], [14] and [3]. Wang et al. [2] develop the dense trajectory model which is employed to track dense sampling points with optical flow. In [14], trajectory-based motion features which are robust to camera movement are applied to enhance the performance of human action recognition. Wang et al. [3] further refine the dense trajectory based method [2] by explicitly estimating camera motions. Due to the good feature representation performance, the improved dense trajectory based method [3] is employed in this work for video feature extraction, which is however time-consuming.

**C. Gaussian Mixture Model (GMM)**

GMM represents the probability distribution of observations by a mixture of K multivariate Gaussian distributions. In the field of multimedia, GMM can be used to cluster the low-level feature vectors into a visual vocabulary like K-Means and the generated vocabulary will be further used for feature encoding. In general, a GMM model with K components can be represented by the parameter set \(\theta = \{\pi_k, \mu_k, \Sigma_k; k = 1, \cdots, K\}\), where \(\mu_k\) and \(\Sigma_k\) are the mean and covariance of the \(k^{th}\) component, and \(\pi_k\) is the weight of the \(k^{th}\) component which can be regarded as the prior probability and should satisfy the conditions that \(0 \leq \pi_k \leq 1\) and \(\sum_{k=1}^{K} \pi_k = 1\). Assume there is a training dataset \(X = \{x_1, \cdots, x_n\}\), then the density \(p(x_i|\theta)\) induced on \(x_i\) can be expressed as

\[
p(x_i|\theta) = \sum_{k=1}^{K} \pi_k N_k(x_i|\mu_k, \Sigma_k),
\]

where \(N_k(x_i|\mu_k, \Sigma_k)\) stands for the \(k^{th}\) multivariate Gaussian distribution as

\[
N_k(x_i|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{d/2} \sqrt{\det \Sigma_k}} \exp \left\{ -\frac{1}{2} (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) \right\},
\]

where \(d\) is the dimension of the data vector \(x_i\).

The general way to estimate GMM parameters given the training dataset \(X\) is to maximize the log-likelihood \(l(\theta; X)\) by the Expectation Maximization (EM) [15] algorithm as

\[
l(\theta; X) = \sum_{i=1}^{n} \log \sum_{k=1}^{K} \pi_k N_k(x_i|\mu_k, \Sigma_k).
\]

To facilitate the EM based manipulation, a posterior possibility \(q_i(k)\) is introduced as

\[
q_i(k) = \frac{\pi_k N_k(x_i|\mu_k, \Sigma_k)}{\sum_{l=1}^{K} \pi_l N_l(x_i|\mu_l, \Sigma_l)},
\]

and \(q_i(k)\) indicates the soft assignment of \(x_i\) to the \(k^{th}\) component of the GMM model. This step is called the Expectation step (E-step). Then, the Maximization step (M-step) is carried out, which is achieved by re-estimating the GMM model based on the given training dataset \(X\) and \(q_i(k)\); specifically, the M-step can be expressed with the following three formulae.

\[
\mu_k = \frac{\sum_{i=1}^{n} q_i(k)x_i}{\sum_{i=1}^{n} q_i(k)},
\]

\[
\Sigma_k = \frac{\sum_{i=1}^{n} q_i(k)(x_i - \mu_k)(x_i - \mu_k)^T}{\sum_{i=1}^{n} q_i(k)},
\]

\[
\pi_k = \frac{\sum_{i=1}^{n} q_i(k)}{\sum_{i=1}^{n} \sum_{k=1}^{K} q_i(k)}.
\]

The E-step and M-step are repeated until \(l(\theta; X)\) changes slightly or the given iteration time is reached.

**D. Fisher Vector (FV)**

It is well known that the Bag-of-Words (BoW) [16] model is the most popular approach for image/video related tasks such as human action recognition. In BoW, the frequency information of visual words is used for visual feature description. In order to further improve the capacity of visual feature representation, the Fisher Vector (FV) model is proposed as an extension of BoW, in which not only frequency information but also extra distribution information of features are encoded. In [12], FV is applied to image categorization and excellent performances are obtained. Oneta et al. [17] employ FV to represent dense trajectory based features for human action recognition.

\(^1\)Spark supports several storage levels allowing users to store a dataset into different places, e.g., memory, memory as serialized or disk. It is usually a useful option to optimize.
In general, FV is performed with GMM. For a given GMM model with the parameter set \( \theta = \{ \pi_k, \mu_k, \Sigma_k; k = 1, \cdots, K \} \) and the dataset \( X = \{ x_1, \cdots, x_n \} \), FV can be calculated by

\[
u_k = \frac{1}{n\sqrt{\sigma_k}} \sum_{i=1}^{n} q_i(k) \left( \frac{x_i - \mu_k}{\sigma_k} \right),
\]

\[
u_k = \frac{1}{n\sqrt{2\pi k}} \sum_{i=1}^{n} q_i(k) \left( \frac{x_i - \mu_k}{\sigma_k} \right)^2 - 1
\]

where \( q_i(k) \) is the posterior possibility as given in Eq.(4), \( u_k \) and \( v_k \) stand for the mean derivation vector and covariance derivation vector related to the \( k^{th} \) Gaussian component, respectively, and \( \sigma_k \) is the diagonal vector of \( \Sigma_k \) with the division being element-wise. By concatenating \( u_k \) and \( v_k \), the final FV can be generated as

\[\psi(X) = [\cdots, u_k, \cdots, v_k, \cdots]^T.\]

Therefore, a \( 2Kd \)-dimension vector is generated by FV, where \( d \) is the dimension of \( x_i \). In order to further improve the FV performance, the power normalization operation can be applied on each dimension of FV as [18]

\[f(z) = \text{sign}(z)|z|^\alpha,
\]

where \( \alpha \) is a regulation parameter which is recommended to be 0.5 in [18] when the number of components \( K \) of the underlying GMM is equal to 256. Another improvement can be made by doing a L2-normalization on \( \psi(X) \) in Eq.(10). After these two operations, a new FV is produced which usually obtains a better classification performance than before.

III. PROPOSED SPARK BASED FRAMEWORK FOR HUMAN ACTION RECOGNITION

Based on the aforementioned introduction in Section II, the details about designing the proposed Spark based framework on trajectory based feature extraction, GMM generation and FV encoding are presented below.

A. Distributed Trajectory Based Feature Extraction

In the proposed framework, we distribute the trajectory based feature extraction workload across the computer cluster utilizing the scalability as well as the fault tolerance power of Spark. Moreover, the Hadoop Distributed File System (HDFS) [19] is employed to store videos to achieve cluster-wide data visibility and high data throughput. We take the improved dense trajectory [3] to fulfill feature extraction which performs computing through consecutive video frames, thus a single video becomes indivisible and each Spark map task needs to take a whole video as input rather than some pre-extracted key frames. On the other hand, to take full advantage of the high-performance power of C/C++ programming, the proposed native feature extraction library is written in C/C++ and performed with the Java Native Interface (JNI). Moreover, to eliminate the redundant video data loading and transmission between Spark and native library, the Inputformat of ‘File Name’ is used to only pass the target video path to the native library. As a consequence, the native library is responsible for video data loading and feature output while the aforementioned redundant cost can be removed.

Due to the indivisible characteristic of a single video, the executing time is often different among extraction tasks. What’s worse, Spark has no information to judge which task will run faster than others since the executing time relies on both the video content and video length. As a result of the unbalanced tasks, the computer cluster will be dragged seriously by the late-completed ‘long task’. To address this issue, we first make an estimation of the execution time for a video, which can be achieved by counting the pixel number contained in this video, since more processing time will be required for a video with a larger number of pixels. Based on this kind of simple estimation, we then reschedule the task execution order by putting ‘long task’ ahead to achieve a better work-load balance.

B. Distributed Gaussian Mixture Model Generation

In this work, a distributed implementation of GMM generation is realized based on the EM algorithm [15] with Spark. As introduced in Section II-C, a GMM model can be generated by the EM algorithm, which is composed of the Expectation and Maximization steps.

The Expectation step aims to get the posterior possibility \( q_i(k) \) as given in Eq.(4) from the given dataset. It could be parallelized by partitioning the dataset and performing relevant computations across the cluster. Specifically, for a data partition \( p_t = \{ x_1^t, \cdots, x_m^t \} \), the following intermediate values are calculated.

\[ll^t = \sum_{i=1}^{m} \log \sum_{k=1}^{K} \pi_k p(x_i^t | \mu_k, \Sigma_k),
\]

\[q_k^t = \sum_{i=1}^{m} q_i(k),
\]

\[q_{-x_k}^t = \sum_{i=1}^{m} q_i(k) x_i^t,
\]

\[q_{-d_k}^t = \sum_{i=1}^{m} q_i(k)(x_i^t - \mu_k)(x_i^t - \mu_k)^T,
\]

where \( ll^t \) indicates the log-likelihood function in Eq.(3) related to the \( t^{th} \) data partition, \( q_k^t \), \( q_{-x_k}^t \) and \( q_{-d_k}^t \) stand for the sum of posterior probability, weighted mean and weighted covariance, which are computed in the Expectation step to support the next Maximization step. The Maximization step accumulates the aforementioned intermediate values and then updates the GMM parameter set \( \theta \). The detailed distributed GMM generation algorithm is described in Algorithm 1.

All the computations shown above are carried out by the native library written in C/C++ for efficiency. In order to make a further efficient transmission between Spark and the native library, the input dataset is serialized in the format of Protocol Buffer (PB) [20], which is regarded as one of most outstanding
libraries for serialization in terms of the size of serialized data and the speed for serializing and deserializing.

C. Distributed Fisher Vector Encoding

As far as FV encoding is concerned, VLFeat [21] is employed for realization which is an open source computer vision algorithm library written in C. For distributed FV encoding, the feature files stored in HDFS are respectively piped to slaves by the Spark pipe function to run the encoding task. Within each slave, a well-designed local native library is used to perform the actual encoding work. Then, the encoded FVs are piped from slaves to the master and finally stored in HDFS. The essential GMM model for FV encoding is broadcast to every slave before FV encoding starts.

Particularly, when the original features are unnecessary to be preserved, e.g., the GMM model is trained from a small proportion of feature data or it already exists, we could directly pipe the feature data stream from the feature extraction process to the FV encoding process. This kind of data flow design without feature data storage can save a lot of time as compared to the traditional process of feature extraction followed by data storage for subsequent FV encoding. The details about distributed FV encoding are illustrated in Fig. 1.

In Fig. 1, the yellow flow refers to the aforementioned ‘direct pipe’ method which pipes feature data directly from the feature extraction process to the FV encoding process without saving operations, while the green flow indicates the traditional procedure to save the output feature data and then read it out for FV encoding.

IV. EXPERIMENTAL RESULTS

A. Experimental Configuration

In order to evaluate the proposed distributed Spark-based framework mentioned in Section III, we carried out a number of experiments on a 9-node computer cluster for human action recognition, focusing on assessing the performance of trajectory-based feature extraction, GMM generation and FV encoding. Besides, an additional PCA model is trained by utilizing the Spark’s PCA implementation to fulfill the preprocessing step which is common for FV encoding. As far as the video dataset is concerned, the benchmark Hollywood-2 [22] human action dataset is used, which provides 12 action classes collected from 69 different Hollywood movies and includes a total of 1,707 videos (823 for training and 884 for testing).

### TABLE I

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i5-3450</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>4</td>
</tr>
<tr>
<td>Memory</td>
<td>32 GB</td>
</tr>
<tr>
<td>DDR3</td>
<td>DDR3</td>
</tr>
<tr>
<td>Network</td>
<td>1 Gbps LAN</td>
</tr>
</tbody>
</table>

The hardware configuration of the eight slave nodes is shown in Table I. As noted in Table I, a computer node with middle-level configuration (as shown as ‘single*’ in Table I) is chosen to run a single CPU thread version of all the following evaluations to demonstrate the speedup achieved by employing the proposed spark-based framework. On the other hand, the hardware configuration of the master node is with CPU of AMD FX 6100 and 24 GB memory. Each node is installed
with Java 1.7, Spark 1.1.1, Hadoop 1.2.1 and GNU/Linux Ubuntu 12.04 operating system. The gigabit NICs are set among the cluster to improve the network transmission speed.

B. Trajectory based Feature Extraction Evaluation

The comparative results on trajectory based feature extraction between the cluster which runs the proposed Spark based framework and a single node are presented in Table II, where the comparative performances are shown for both training and testing. In Table II, two different task executing orders are employed, including the naive order and priority order. The naive order indicates that there is no preference to initiate the execution tasks and the traditional execution order used in Spark is carried out, while the late-completed ‘long task’ will be started earlier when performing the priority execution order as discussed in Section III-A. SR stands for the speedup ratio of the running time achieved by the cluster with the proposed framework against the single CPU thread version.

<table>
<thead>
<tr>
<th>Data</th>
<th>Platform</th>
<th>Task Executing Order</th>
<th>Time (s)</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>Cluster</td>
<td>Naive order</td>
<td>12,718</td>
<td>26.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priority order</td>
<td>11,005</td>
<td>30.1</td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td>N/A</td>
<td>331,167</td>
<td>1.0</td>
</tr>
<tr>
<td>Testing</td>
<td>Cluster</td>
<td>Naive order</td>
<td>13,301</td>
<td>28.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Priority order</td>
<td>12,291</td>
<td>30.3</td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td>N/A</td>
<td>372,117</td>
<td>1.0</td>
</tr>
</tbody>
</table>

From the results in Table II, the process of feature extraction is greatly accelerated when the cluster is used as demonstrated by the SR performances. When comparing the SR performances between the naive order and priority order, it is obvious to observe that the priority order is superior to the naive order in real-time performance. To further demonstrate the comparison between the naive and priority orders, the relative running time against all executed mappers are illustrated in Fig. 2, where there are a total of 32 mappers to run the feature extraction tasks, i.e., there are 32 CPU cores as shown in Table II with each CPU core running one map function. The relative running time of the $i^{th}$ mapper is defined as

$$\Delta T_i = \frac{T_i}{\max_i \{T_i\}},$$ (16)

where $T_i$ stands for the running time consumed by the $i^{th}$ mapper when the naive order or priority order is applied. As shown in Fig. 2, it is clear that the whole cluster is dragged by some ‘long task’ mappers with the naive order while the priority order offers a more balanced behavior, and thus it is no wonder that the priority order achieves a higher SR than the naive order.

C. Distributed GMM Generation Evaluation

In this part, we compare the performances of distributed GMM generation between our implementation and the latest MLlib [8] implementation as shown in Table III, where both the running time (single iteration) in units of seconds and the speedup ratio (SR) achieved by our implementation over MLlib are presented.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Trajectory</th>
<th>HOF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 cores</td>
<td>32 cores</td>
</tr>
<tr>
<td>MLlib</td>
<td>2,042</td>
<td>1,120</td>
</tr>
<tr>
<td>Ours</td>
<td>153</td>
<td>82</td>
</tr>
<tr>
<td>SR</td>
<td>13.3</td>
<td>13.6</td>
</tr>
</tbody>
</table>

As shown in Table III, the comparison is made under two experimental scenarios. Under the first scenario, we evaluate the comparative performance with two different sets of features: 1) the trajectory feature [3] of 30 dimensions and 2) the Histogram of Optical Flow (HOF) [23] of 108 dimensions. Under the second scenario, the comparison is performed with two sets of hardware configurations to analyze the scalability: 1) all slave nodes with 32 CPU cores are used and 2) only half of the slave nodes with 16 CPU cores are utilized. From the results, it is obvious that our GMM
implementation outperforms MLlib in running speed. This significant improvement mainly relies on our well-designed native library written in C. Although MLlib uses nativeBlas to realize the linear algebra computations, it is still slower than a full-C implementation. In addition, when the number of feature dimension increases (i.e., from trajectory to HOF), the speedup achieved by our implementation over MLlib is improved. As far as the scalability performance is concerned, the efficiency is not doubled as the number of mappers does. This is mainly because the cost of communication and scheduling rises as the number of mappers increases.

D. Distributed FV Encoding Evaluation

The speedup results about FV encoding achieved by the proposed Spark based framework are presented in Table IV, where two data storage strategies are tested, including ‘save-then-pipe’ and ‘direct pipe’, as discussed in Section III-C and shown in Fig. 1.

<table>
<thead>
<tr>
<th></th>
<th>direct pipe</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data</td>
<td>Platform</td>
<td>Time (s)</td>
<td>SR</td>
</tr>
<tr>
<td>Training</td>
<td>Cluster</td>
<td>1,417</td>
<td>30.7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td>43,500</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>Cluster</td>
<td>1,772</td>
<td>30.9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Single</td>
<td>54,875</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

From the comparative results presented in Table IV, it can be seen that the process of FV encoding can be dramatically speeded up by the proposed Spark based framework running on the 32-core cluster for both the storage strategies of ‘save-then-pipe’ and ‘direct pipe’. Moreover, when comparing the running time performances of ‘save-then-pipe’ and ‘direct pipe’, it can be observed that the amount of computations for FV encoding is significantly reduced by the ‘direct pipe’ strategy, i.e., the speedup of ‘direct pipe’ against ‘save-then-pipe’ is $2.7 \times 3.3 \times$. To validate our framework’s correctness, we adopt a similar experimental setting as [3] while the Human Detection (HD) technique is not applied for feature extraction. The mean Average Precision (mAP) result on Hollywood-2 is 63% which is consistent with that in [3].

V. CONCLUSION

In this work, we leverage the in-memory computing and fault-tolerance abilities provided by Spark to solve the large-scale human action recognition problem. The distributed solutions to several key components for human action recognition are designed, including trajectory based feature extraction, GMM generation and FV encoding. The experimental results demonstrate that the proposed Spark based framework can greatly improve the real-time performance for human action recognition, with satisfactory scalability performances achieved.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 61472281, the “Shu Guang” project of Shanghai Municipal Education Commission and Shanghai Education Development Foundation under Grant 12SG23, the Program for Professor of Special Appointment (Eastern Scholar) at Shanghai Institutions of Higher Learning (No. GZ2015005), and the Fundamental Research Funds for the Central Universities under Grant 0800219270.

REFERENCES