Vishakh.S (B16CS038)

Sabitra Panigrahi (IDRP19CG201)

CSL7093 Project Report

BigDL

May 25, 2020

Abstract

Continued advancements in the field of deep learning have caused an increasing demand from the organizations to apply deep learning to the production big data platforms. BigDL is a distributed deep learning framework for Apache Spark, which has been adopted by a number of industry users to take DL models to the production pipeline. BigDL has been written as a library on top of the standard Apache Spark ecosystem. It allows the users to write their deep learning applications as a Spark program and run it on the existing Apache Spark or Hadoop clusters. This allows them to directly process the production data and become a part of the end-to-end pipeline for development and deployment. BigDl implements distributed data-parallel training directly on top of the coarse-grained functional compute model of Apache Spark. Each worker uses a replica of the model to train on a fraction of the dataset and computes local parameters. BigDL implements a parameter server style architecture in view of synchronizing the parameter values computed by the worker nodes in the cluster. BigDL has been optimized to run on Intel's Xeon CPU based architectures by leveraging Intel's Math Kernel Library (MKL) and is found to provide a performance that is several orders of magnitude faster than the single node performance.

Objective

- To take a look at BigDL, a framework that has been used by many users in the industry for performing deep learning in a distributed fashion on production big data systems.
- To understand the drawbacks of the other frameworks out there and the motivation behind developing BigDL.
- To describe the key features of BigDL. There are a lot of deep learning frameworks out there. Then why BigDL? We try to understand what makes BigDL on Spark special when compared to the other competing frameworks.
- To understand the distributed execution model adopted by BigDL (the model follows the practice of data parallel training for big data), which is a viable alternative for distributed model training (model parallel training which is employed by other conventional deep learning frameworks).
- To understand the key limitations and certain known issues of BigDL.

Generic Details

1st Version	Latest Version	Development Status	Software License Category	Maintained by	Categorization
2016	0.10.0 (2019)	Active	Apache license 2.0	Intel Analytics	Distributed memory system

Introduction

Technology is growing in leaps and bounds everyday. With the explosion of mobile networks and cloud-based computing growing in popularity everyday, there has been an incomprehensible increase in the amount of data being produced and handled. The term "big data" has been used to refer to such humongous amounts of data that inundates the world. Big data is generally characterized by 3 V's: Volume, Velocity and Variety, as shown in Figure 1. Volume refers to the large quantity of data in question. Velocity refers to the rate at which data is generated, processed or consumed. Finally, Variety suggests the fact that data comes in all formats - structured, numeric data like those in traditional databases to the unstructured data like emails, videos, images, etc.

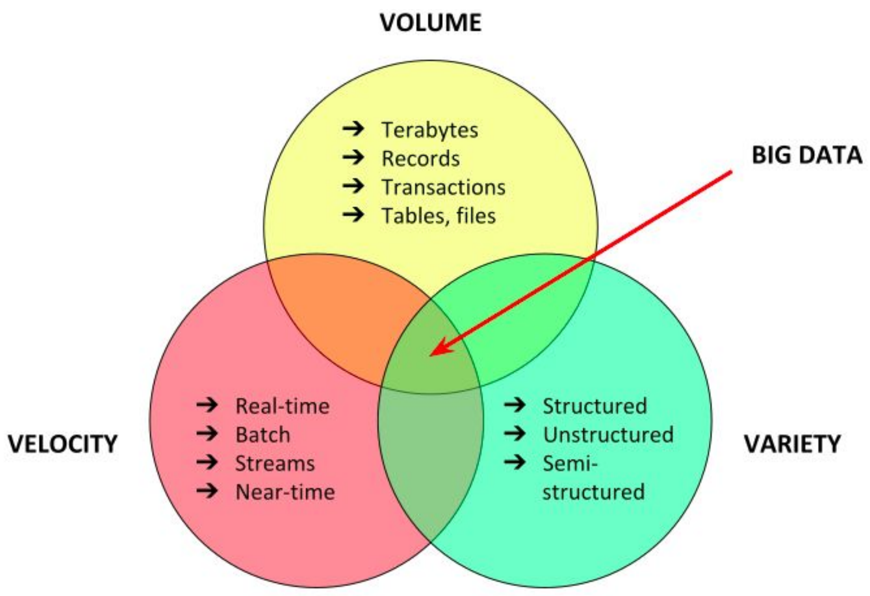


Figure 1: 3V's of big data

Sometimes, 2 more V's are added - Veracity and Value. Veracity is a measure of the authenticity of the data or degree to which a piece of data can be trusted. Value refers to the business value and utility of the data in question. Having large amounts of data is all well, but it can be useful only if we can handle it properly. Big data does not fit into the architectures of the conventional database systems and exceeds their processing capacity. Thus, a new way to process big data was needed. This led to the field of Big Data Analytics. Big Data Analytics comprises the algorithms, tools, etc required to handle large amounts of dynamic data. Big data analytics opens up a plethora of opportunities to companies. It helps them have a competitive edge over their counterparts. For example, real-time analysis of the market trends helps companies make crucial timely decisions. Technologies like column-oriented databases, schema-less databases, MapReduce, Hive, Chukwa, etc evolved as a result of the efforts to store, retrieve and process large amounts of data.

- Columnar databases store data in columns and allow large data compression and very fast query times.
- Schema-less databases or NoSQL databases are used to store semi-structured or unstructured data.
- MapReduce is a processing technique and programming model for distributed computing based on Java. MapReduce consists of 2 phases - Map (input dataset is mapped to a set of key-value pairs or tuples) and Reduce (several of the outputs of the map phase are combined to form a reduced set of couples).
- Hive was developed to be an SQL-like bridge that allows applications to run queries against a Hadoop cluster.
- Apache Chukwa provides an end-to-end delivery model that leverages local log files. It provides a way to display, monitor and analyze such data.

Continued advancements in the field of artificial intelligence brought deep learning to the forefront of the data analytics field. The existing deep learning frameworks such as Caffe, Torch and Tensorflow evolved as a result. As the requirements and models expanded, new architectures and systems beyond the conventional frameworks started to evolve. In particular, there was a demand from the companies to integrate deep learning technologies with their existing machine learning pipelines. This would enable the deep learning workloads to become mainstream. Applying deep learning to production is however a tough challenge because of the fact that real world data is generally dynamic and very messy. Instead of running the data processing only once, real-world data analytics is an iterative and recurrent process. It involves back and forth development and debugging, incremental model update with new data, etc. Hence, it is highly inefficient to run such workloads on separate big data and deep learning systems. For example, processing the data on a Spark cluster and then exporting the processed data to a separate TensorFlow cluster for training or predictions is very expensive. The data transfer between the two systems may end up bottlenecking the operation productivity.

One way to resolve this issue is to adopt a connector approach (as used in CaffeOnSpark, TensorFlowOnSpark, etc). Here proper interfaces are developed to connect the data processing and the deep learning components. However, such a solution suffers from large overheads in the form of inter-process communication, data persistency, etc. Further, deep learning systems and data processing systems generally have different distributed execution paradigms. Big data tasks are considered to be embarrassingly parallel and independent of each other. If for example, a Spark worker fails, the system just relaunches the worker. Deep learning tasks however need to coordinate with each other as they tend to be dependent on each other.

To solve this problem and meet these demands, BigDL was developed. It was developed as a distributed deep learning framework for big data. It is implemented as a library on top of Apache Spark and it allows the users to write their applications as Spark programs and run them on top of the existing Hadoop or Spark clusters. Figure 2 shows where BigDL is located in the Spark ecosystem.

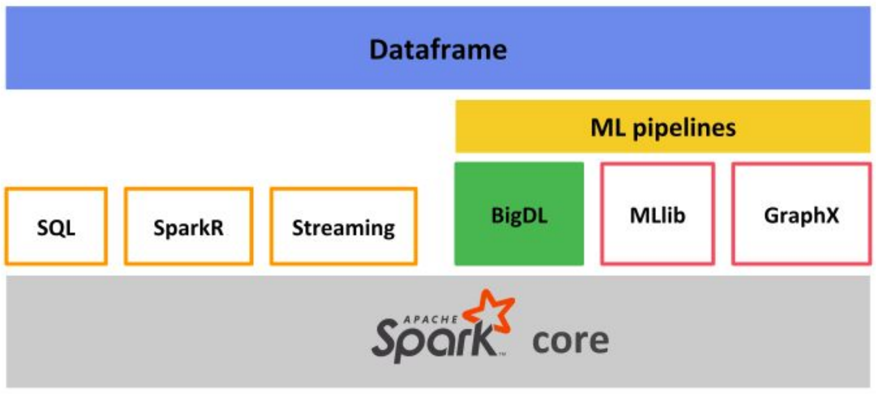


Figure 2: Apache Spark ecosystem

The native Spark ecosystem does not have any support for deep learning applications. This is why BigDL was built. BigDL is a library with deep learning features, complementary to MLlib. From a functional point of view, it has feature parity with Caffe and Torch. It supports an API similar to Keras and Torch for constructing neural network models and allows large-scale distributed training and inference by leveraging the scale-out architecture (running across hundreds of servers in a distributed fashion efficiently) of the underlying Apache Spark framework.

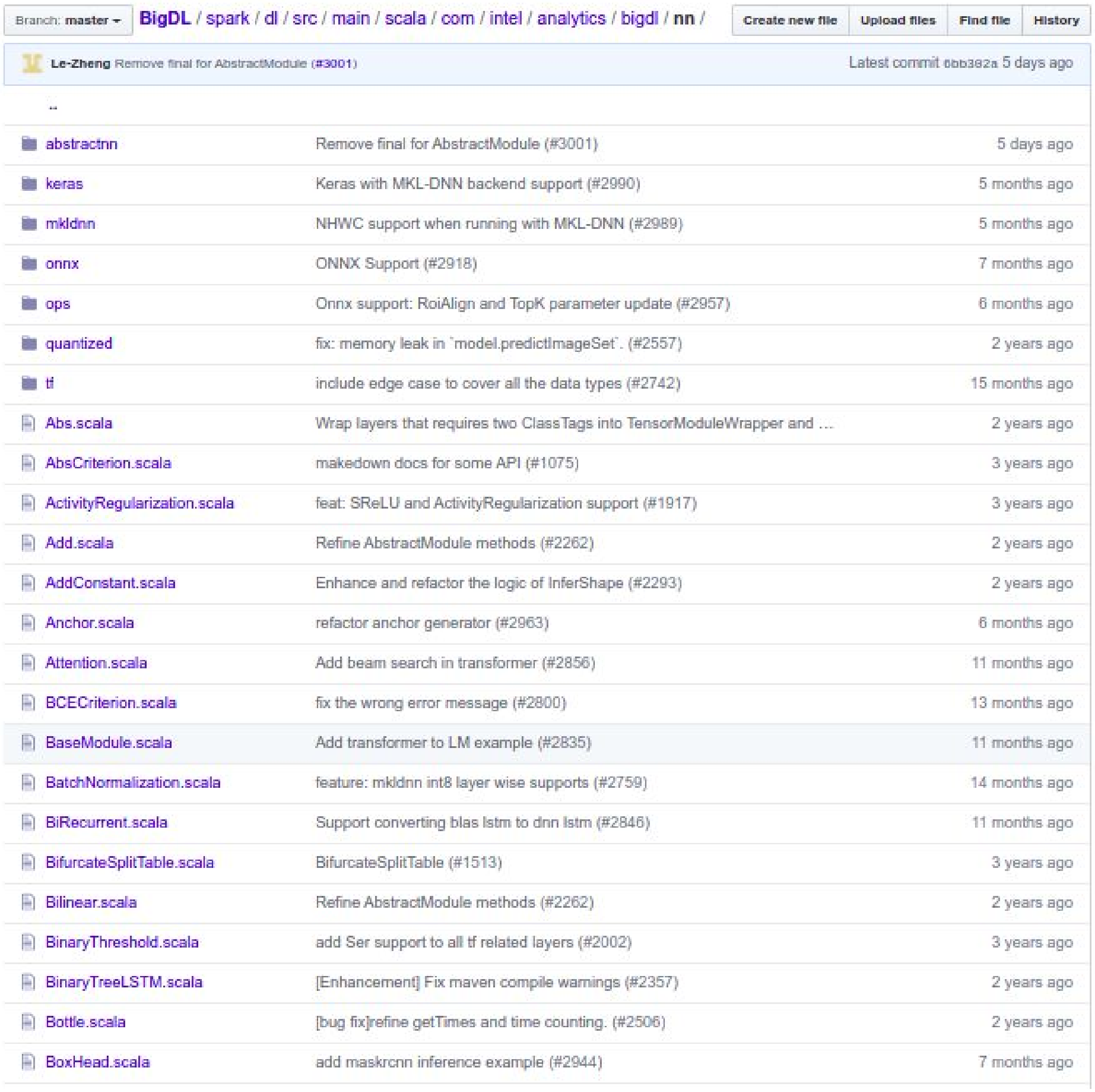
According to [1], BigDL provides an expressive data-analytics integrated deep learning programming model. Within a unified data analysis pipeline, users can very efficiently process large volumes of data, feed a distributed dataset to a neural network and perform training and inferencing on top of the underlying Apache Spark framework. Specifically, BigDL provides data-parallel training on top of the coarse-grained functional compute model of Apache Spark. This is contrary to the conventional machine learning community's wisdom that fine-grained data access is needed for efficient training in a distributed scenario. By unifying the execution models of a traditional neural network model and big data analysis, BigDL allows complex algorithms to be seamlessly integrated into the production pipelines. These can be easily deployed, monitored and managed in a single unified big data platform.

Features [Minimum 1000 words, bullet points with headings]

1. **Rich deep learning support for Spark**

   BigDL has been implemented as a standalone library on top of Apache Spark. BigDL has been modeled after Torch and it has extensive support for deep learning including numeric computing (via tensors) and high-level neural networks. As far as deep learning applications go, it is critical that they run where the data is stored and form a part of the end-to-end pipeline. Further, big data from real-world scenarios is often huge and messy. BigDL allows the users to write deep learning applications as standard Spark programs using Scala or Python and run them directly on top of the existing Apache Hadoop or Spark clusters. In addition to this, it provides feature parity with the popular open-source deep learning framework such as Caffe, Torch, Tensorflow, etc.

   BigDl has implemented over a 100 layers which can be used to write complex models. These layers include most of the layers available in other popular deep learning frameworks. Figure 3 is a snapshot from BigDL’s git repository, showing different layers implemented. BigDL has implemented over 20 loss functions to give an estimate of how good your model is. In addition to this, several optimization methods have been written to train the model. Figure 4 is a snapshot from BigDL’s git repository, showing different optimizers available to the user.



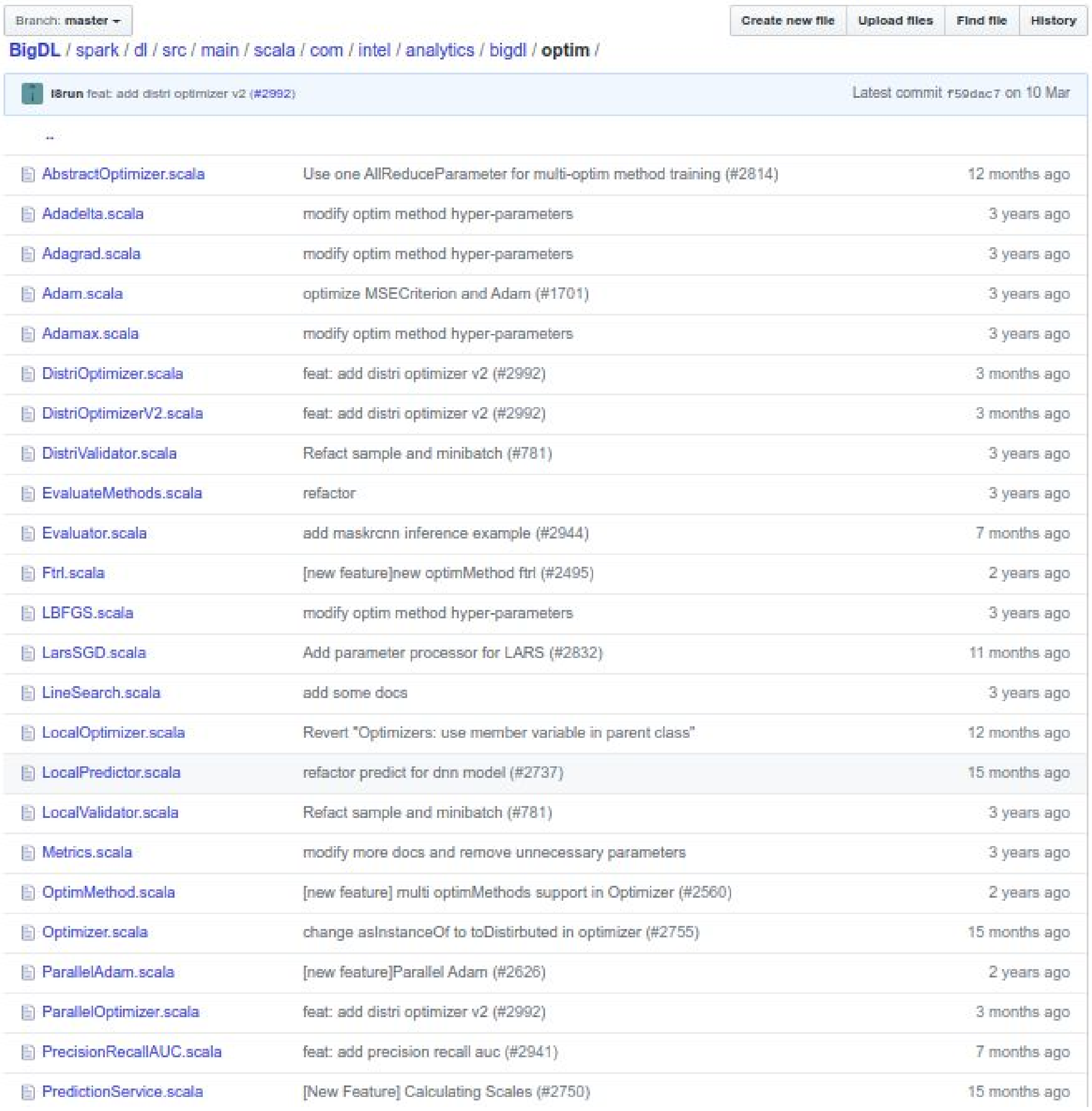
Branch: master ▾ BigDL / spark / dl / src / main / scala / com / intel / analytics / bigdl / nn /

Le-Zheng Remove final for AbstractModule (#3001) Latest commit 6bb392a 5 days ago

..

abstractnn	Remove final for AbstractModule (#3001)	5 days ago
keras	Keras with MKL-DNN backend support (#2990)	5 months ago
mkldnn	NHWC support when running with MKL-DNN (#2989)	5 months ago
onnx	ONNX Support (#2918)	7 months ago
ops	Onnx support: RoiAlign and TopK parameter update (#2957)	6 months ago
quantized	fix: memory leak in `model.predictImageSet`. (#2557)	2 years ago
tf	include edge case to cover all the data types (#2742)	15 months ago
Abs.scala	Wrap layers that requires two ClassTags into TensorModuleWrapper and …	2 years ago
AbsCriterion.scala	makedown docs for some API (#1075)	3 years ago
ActivityRegularization.scala	feat: SReLU and ActivityRegularization support (#1917)	3 years ago
Add.scala	Refine AbstractModule methods (#2262)	2 years ago
AddConstant.scala	Enhance and refactor the logic of InferShape (#2293)	2 years ago
Anchor.scala	refactor anchor generator (#2963)	6 months ago
Attention.scala	Add beam search in transformer (#2856)	11 months ago
BCECriterion.scala	fix the wrong error message (#2800)	13 months ago
BaseModule.scala	Add transformer to LM example (#2835)	11 months ago
BatchNormalization.scala	feature: mkldnn int8 layer wise supports (#2759)	14 months ago
BiRecurrent.scala	Support converting blas lstm to dnn lstm (#2846)	11 months ago
BifurcateSplitTable.scala	BifurcateSplitTable (#1513)	3 years ago
Bilinear.scala	Refine AbstractModule methods (#2262)	2 years ago
BinaryThreshold.scala	add Ser support to all tf related layers (#2002)	3 years ago
BinaryTreeLSTM.scala	[Enhancement] Fix maven compile warnings (#2357)	2 years ago
Bottle.scala	[bug fix]refine getTimes and time counting. (#2506)	2 years ago
BoxHead.scala	add maskrcnn inference example (#2944)	7 months ago

Figure 3: Snapshot of BigDL got repo - different layers implemented



Branch: master ▾ | Create new file | Upload files | Find file | History

BigDL / spark / dl / src / main / scala / com / intel / analytics / bigdl / optim /

i8run feat: add distri optimizer v2 (#2992) — Latest commit f59dac7 on 10 Mar

..		
AbstractOptimizer.scala	Use one AllReduceParameter for multi-optim method training (#2814)	12 months ago
Adadelta.scala	modify optim method hyper-parameters	3 years ago
Adagrad.scala	modify optim method hyper-parameters	3 years ago
Adam.scala	optimize MSECriterion and Adam (#1701)	3 years ago
Adamax.scala	modify optim method hyper-parameters	3 years ago
DistriOptimizer.scala	feat: add distri optimizer v2 (#2992)	3 months ago
DistriOptimizerV2.scala	feat: add distri optimizer v2 (#2992)	3 months ago
DistriValidator.scala	Refact sample and minibatch (#781)	3 years ago
EvaluateMethods.scala	refactor	3 years ago
Evaluator.scala	add maskrcnn inference example (#2944)	7 months ago
Ftrl.scala	[new feature]new optimMethod ftrl (#2495)	2 years ago
LBFGS.scala	modify optim method hyper-parameters	3 years ago
LarsSGD.scala	Add parameter processor for LARS (#2832)	11 months ago
LineSearch.scala	add some docs	3 years ago
LocalOptimizer.scala	Revert "Optimizers: use member variable in parent class"	12 months ago
LocalPredictor.scala	refactor predict for dnn model (#2737)	15 months ago
LocalValidator.scala	Refact sample and minibatch (#781)	3 years ago
Metrics.scala	modify more docs and remove unnecessary parameters	3 years ago
OptimMethod.scala	[new feature] multi optimMethods support in Optimizer (#2560)	2 years ago
Optimizer.scala	change asInstanceOf to toDistirbuted in optimizer (#2755)	15 months ago
ParallelAdam.scala	[new feature]Parallel Adam (#2626)	2 years ago
ParallelOptimizer.scala	feat: add distri optimizer v2 (#2992)	3 months ago
PrecisionRecallAUC.scala	feat: add precision recall auc (#2941)	7 months ago
PredictionService.scala	[New Feature] Calculating Scales (#2750)	15 months ago

Figure 4: Snapshot of BigDL got repo - different layers implemented

2. **Lower costs**

   BigDL allows users to write the DL applications as standard Spark programs and use the existing Spark or Hadoop clusters. This is done without explicitly requiring any specialized hardware. Hence the costs are significantly reduced.

3. **Extremely high performance**

   BigDL uses the optimized Intel's MKL (Math Kernel Library) and multi-threaded programming for each task. BigDL is optimized for Xeon. Hence, running BigDL applications on Xeon processor-based infrastructure (distributed) is orders of magnitudes faster than the conventional Caffe, Torch or Tensorflow models running on a single node. The single node performance on a Xeon E5-26XX V3 is comparable to that of the mainstream GPU's.

4. **Efficiently scale-out**

   BigDL supports large-scale distributed training on top of Apache Spark. It allows applications to seamlessly scale-out across several Intel Xeon processor-based nodes, by leveraging the standard Spark model and efficiently implementing synchronous SGD and an all-reduce communications model.

   More specifically, BigDL performs data-parallel training to train a neural network across the cluster. Each iteration of the training process runs a couple of Spark jobs. Each of these runs some tasks. Each task computes the local gradients using the current mini-batch of data. A single update is made to the shared key-value store. The updated values are read in the subsequent iteration.

5. **Loading pre-trained models**

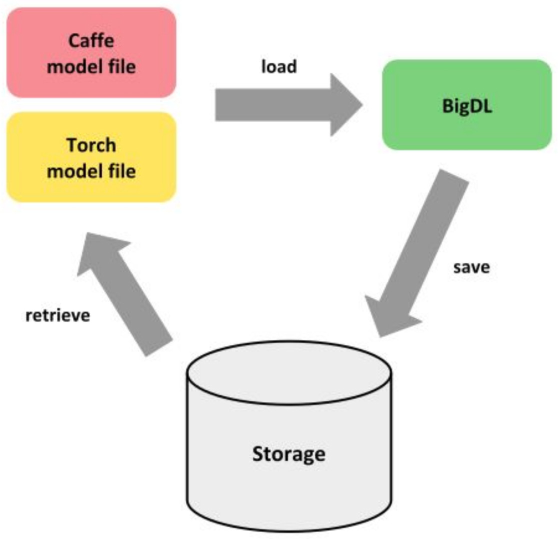


Figure 5: Loading pre-trained models

Pre-trained models are those that are trained on large benchmark dataset, to solve a problem similar to the one under consideration, Users can load pre-trained Caffe or Torch models into Spark programs using BigDL. Figure 5 shows the flow diagram for the same.

The provision to import pre-trained models is highly useful. The pre-trained models usually contain the weights learnt for the network for some classification task. Such a network can be used for other similar tasks because, at the higher level, features extracted will have some inherent similarity. If such a pre-trained network is employed, the number of steps taken to converge to the output decreases drastically.

Using pre-trained models, we can train on different datasets for different purposes. Fine tuning can be performed to predict finer features of the model. For example in Figure 6, the Caffe image set model is loaded into the BigDL model. Fine tuning can be performed to predict the image style rather than the image type. The model was initially successful in distinguishing facial images from non-facial images. After fine tuning the same model, it goes to a state where it can predict the emotions that are being displayed .i.e. The model moves from a state of recognizing faces to a state where it can make inferences based on the underlying contours within it.

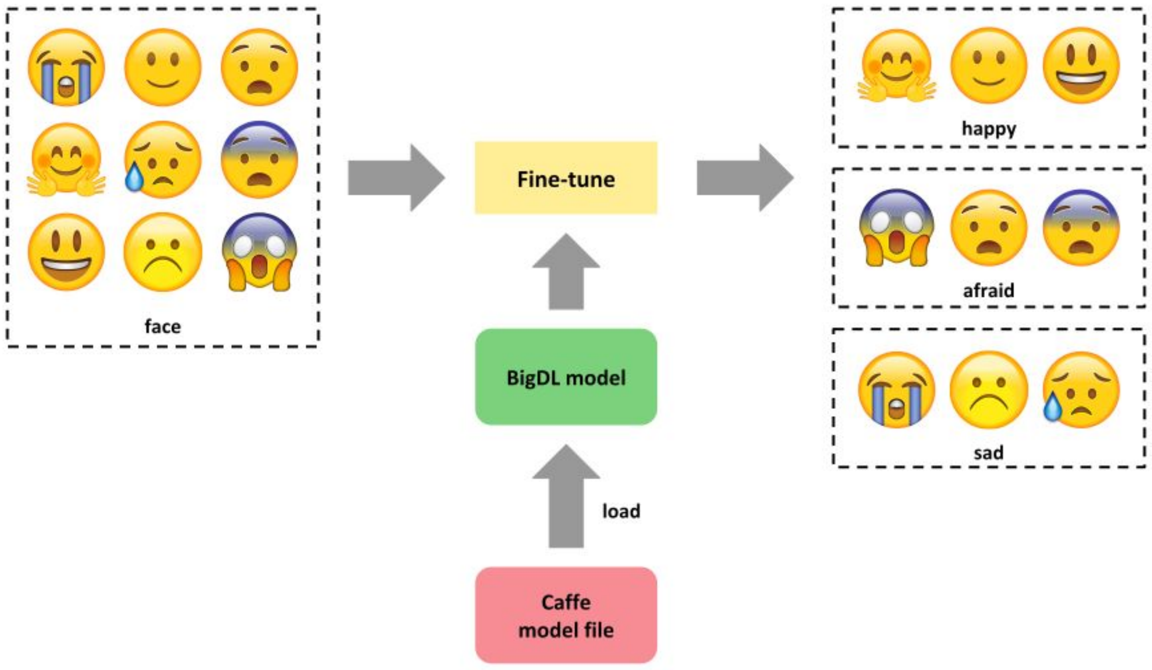


Figure 6: Fine tuning using a pre-trained model.

6. **Model snapshots**

   BigDL supports model snapshots. A snapshot of the job is saved periodically to an index (or timestamp) within the cluster. This is done to provide resiliency in the event of a system failure. There are several advantages of model snapshots:

   - When the training process spans over a long period of time, snapshots can be pretty useful to stop and resume training at a later point of time.
   - Snapshots are very useful for performing inferences at a later point of time.
   - Provide a way to share models with others.

   The amount of time required to create and save these snapshots is proportional to the size of the model in memory. The age of a snapshot is a measure of how stale the snapshot is. Put another way, it is computed as the timestamp of any snapshot relative to the most recent timestamp. If there are snapshots older than a day from the most recent timestamp, they are automatically deleted, except for the first snapshot for each day. All snapshots older than ten days are destroyed. Snapshot retention and deletion settings can be varied. Snapshots consume space on the cluster and if they are not deleted, we may run the risk of exhausting the available space.

7. **Integrates with the rest of the Spark ML pipeline**

   BigDL integrates nicely with Spark's ML/ DL pipeline and this helps a lot in developing complex end-to end ML or DL applications. For example, Figure 7 shows how BigDL integrates with Spark streaming for runtime training and prediction.

   The streaming data from any one of the data sources like Flume, Kafka, Kinesis, Twitter, etc is processed by Spark Streaming to convert the incoming data into RDD's. During the training phase, we can train the models using these RDD's. During the prediction phase, the RDD's can be loaded for runtime prediction on the trained model. The output can then be published to the storage or the event system using StreamWriter.

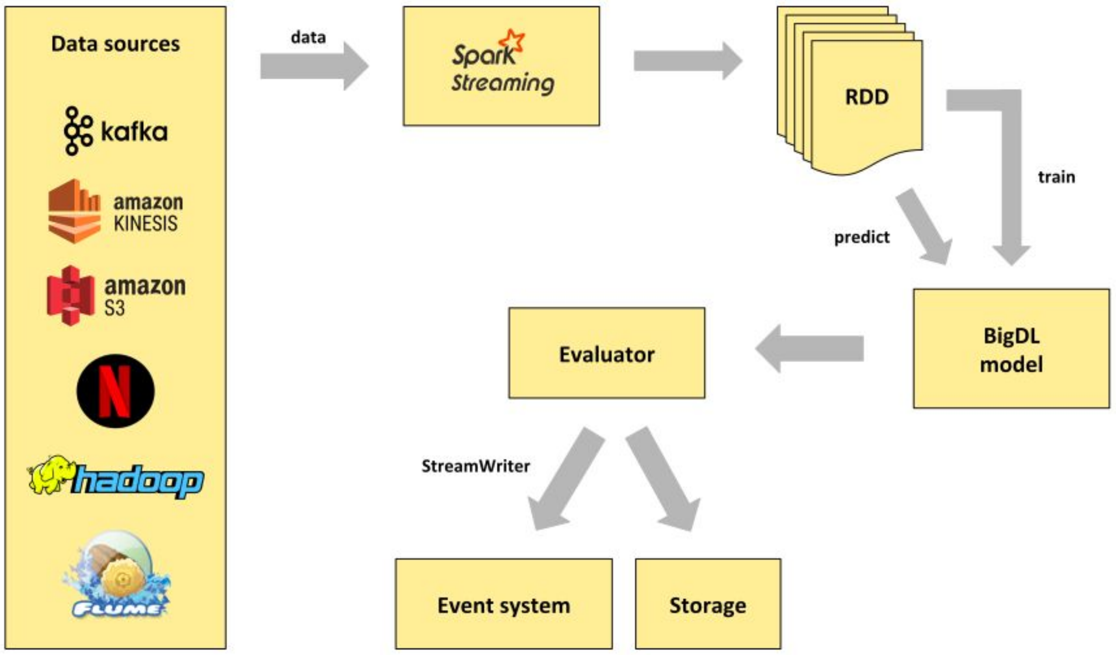


Figure 7: Integration with Apache Spark for run-time training and prediction.

Detailed Architecture Design

This section explains how BigDL supports large-scale distributed training on top of Apache Spark. As already mentioned, BigDL is implemented as a standalone library on Spark. The conventional machine learning community believes that fine-grained data access is needed for a highly efficient implementation. However, big-data systems such as Spark adopt a functional, coarse-grained (applying same operations to all data items) compute model. The key challenge handled in BigDL is the efficient implementation of parallel training on Spark's coarse-grained model.

The Spark execution model is similar to other big data systems in several ways. A Spark cluster comprises a single driver node and multiple workers as shown in Figure 8. At the higher level, the driver is responsible for coordinating the workers and scheduling the tasks in a Spark job. The actual computation occurs in the form of Spark tasks running within the worker nodes.

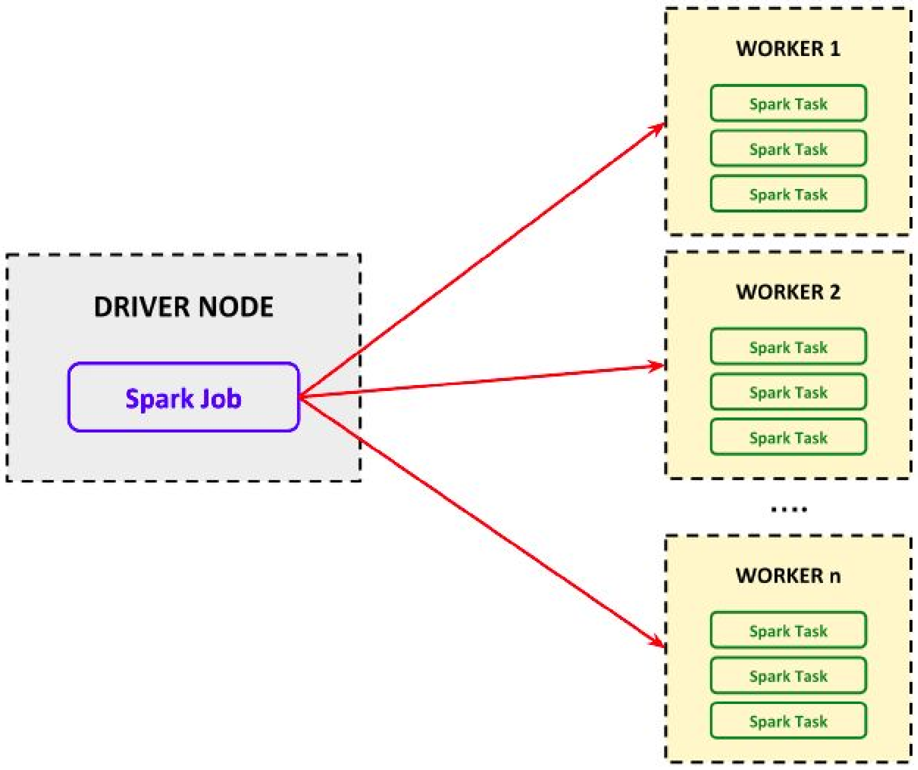


Figure 8: The standard Spark execution model - high-level view

Deep learning applications are written as standard Spark programs and run on top of the existing Hadoop or Spark clusters. On the driver side, the program calls the BigDL library. During training, there are many iterations and each of these is run as a standard Spark job launched by the BigDL library. Following the standard Spark execution model, several tasks are spawned inside the workers. Each of these tasks leverages Intel's Math Kernel Library (MKL) to speed up the training process. This flow is depicted in Figure 9.

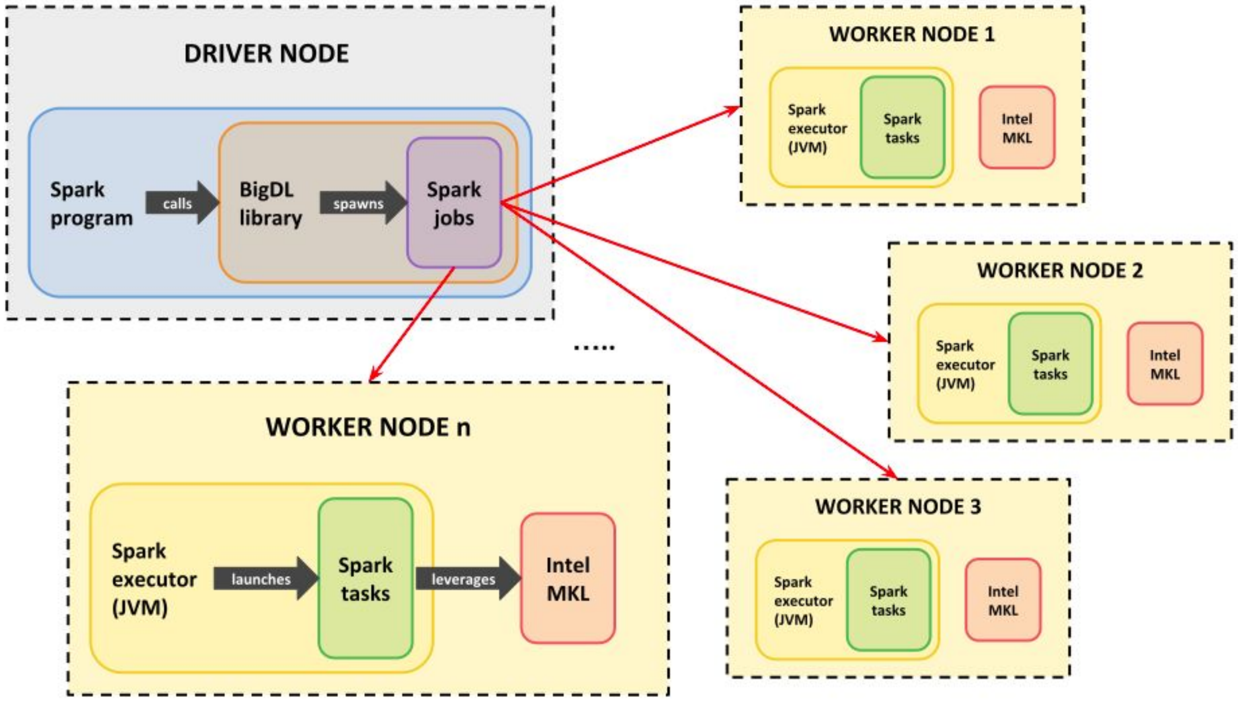


Figure 9: The standard Spark execution model - low-level view

In Spark, the fundamental data structure is a RDD (Resilient Distributed Dataset). RDD is an immutable distributed collection of objects (partitioned across a cluster). It is fault-tolerant and can be operated on in a parallel fashion. In Spak, RDD's support two types of operations - transformation and action. Transformation operation transforms RDD from one form to another. This is done using Spark operations such as *map*, *filter*, *reduce*, etc. Transformation, however, does not modify the existing RDD. During actions, computation is performed on the partitioned dataset to obtain the required results. The obtained results are then transmitted back to the driver node. RDD operations are represented in Figure 10 below.

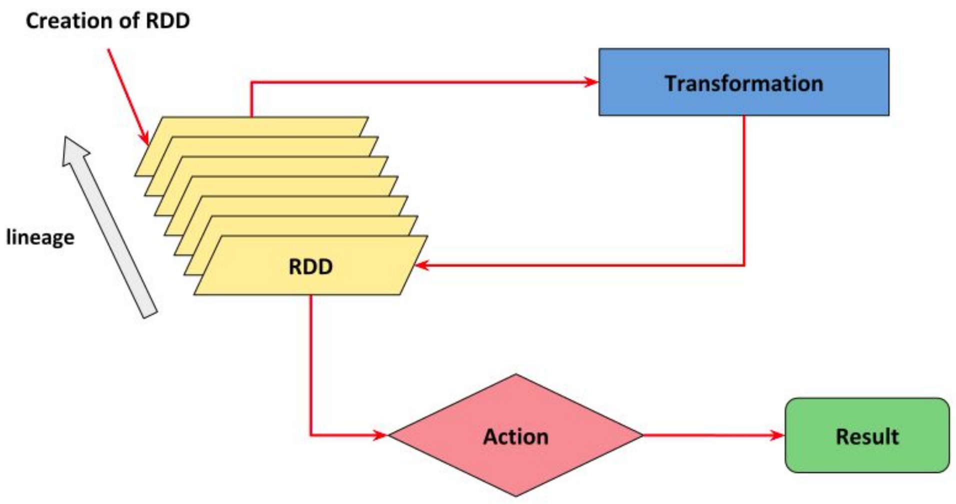


Figure 10: RDD operations

Distributed training can be done in one of the two ways - data parallelism or model parallelism. In data-parallel training (Figure 11), the RDD of the sample data is partitioned across the different workers. Each worker gets a replica of the full model RDD and uses it to perform training on the obtained sample partition. In model-parallel training (Figure 12), the RDD of the model data is partitioned across the different workers. Each worker gets a replica of the sample data and uses it for training using the obtained model partition. Put another way, some computation is performed locally in each worker using the full model, but only for part of the dataset.

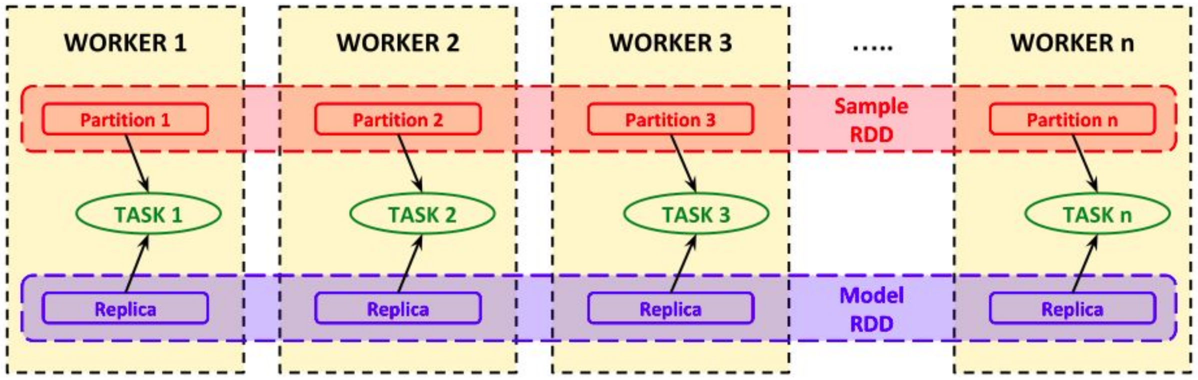


Figure 11: Data parallelism

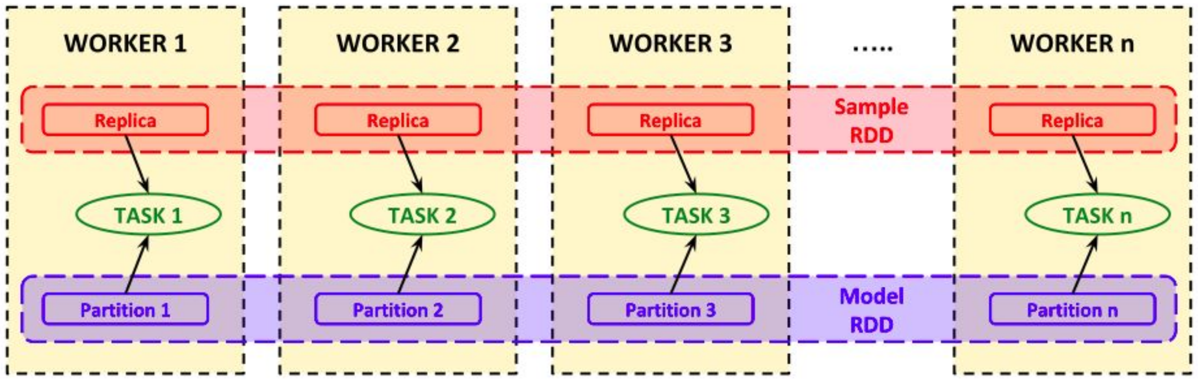


Figure 12: Model parallelism

BigDL performs synchronous data-parallel training. The RDD of samples is partitioned across the network. An RDD of models is constructed, each of which is a replica of the original. Before training starts, both the model and sample RDD's are cached in the memory and partitioned or co-located across the cluster as shown in Figure 11.

The implementation of data-parallel training on standard Spark is very expensive as communications become a serious bottleneck (see Figure 13).

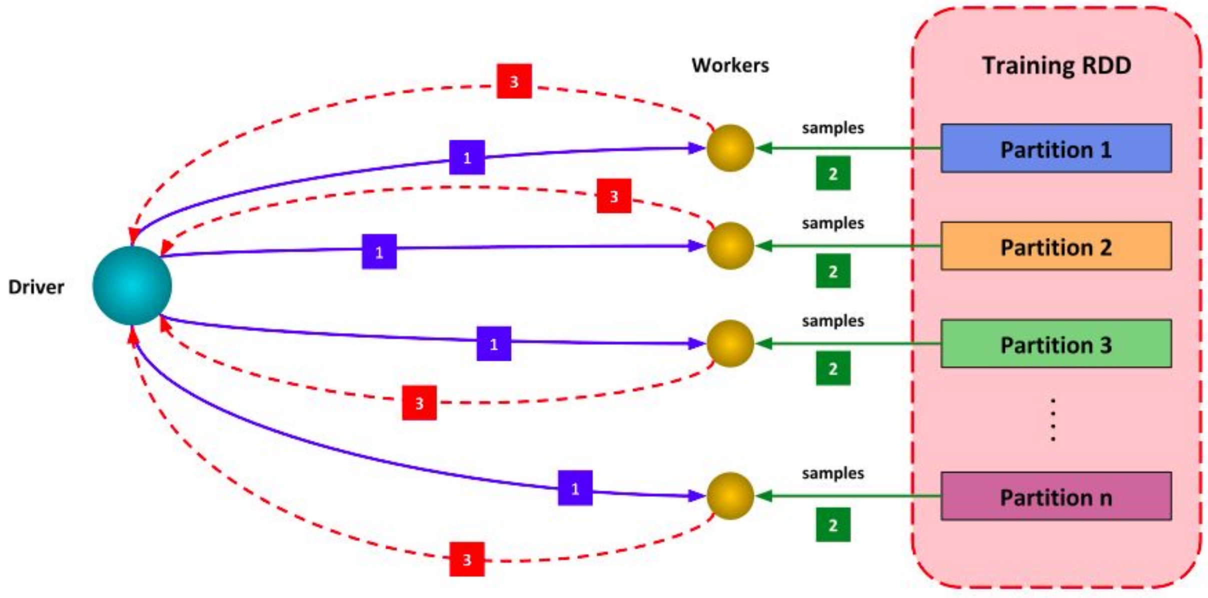


Figure 13: Communications in data-parallel training on Spark

The driver communicates the global parameters to the worker nodes. The worker nodes compute the local gradients and communicate back with the driver node. The driver then updates the parameters and sends them back to the workers. This cycle repeats. Here, the training process ends up becoming communication-bound rather than compute-bound. As the number of workers increases, the workload on the driver increases drastically and the driver node soon becomes a bottleneck.

This bottleneck has been handled in BigDL. The driver node is not involved in the training step. Distributed training in BigDL is an iterative process. Each iteration launches a couple of Spark jobs. The first step involves each worker, computing the local gradients using the model replica and the current mini-batch of data. The worker is responsible for fetching the global parameter from the Parameter Server, which is essentially a key-value store. The next step is a single update to the parameters of the network model. Hence, the number of interactions/ communications has been drastically minimised. Figure 14 represents the all-reduce communications model involved in training using the parameter server.

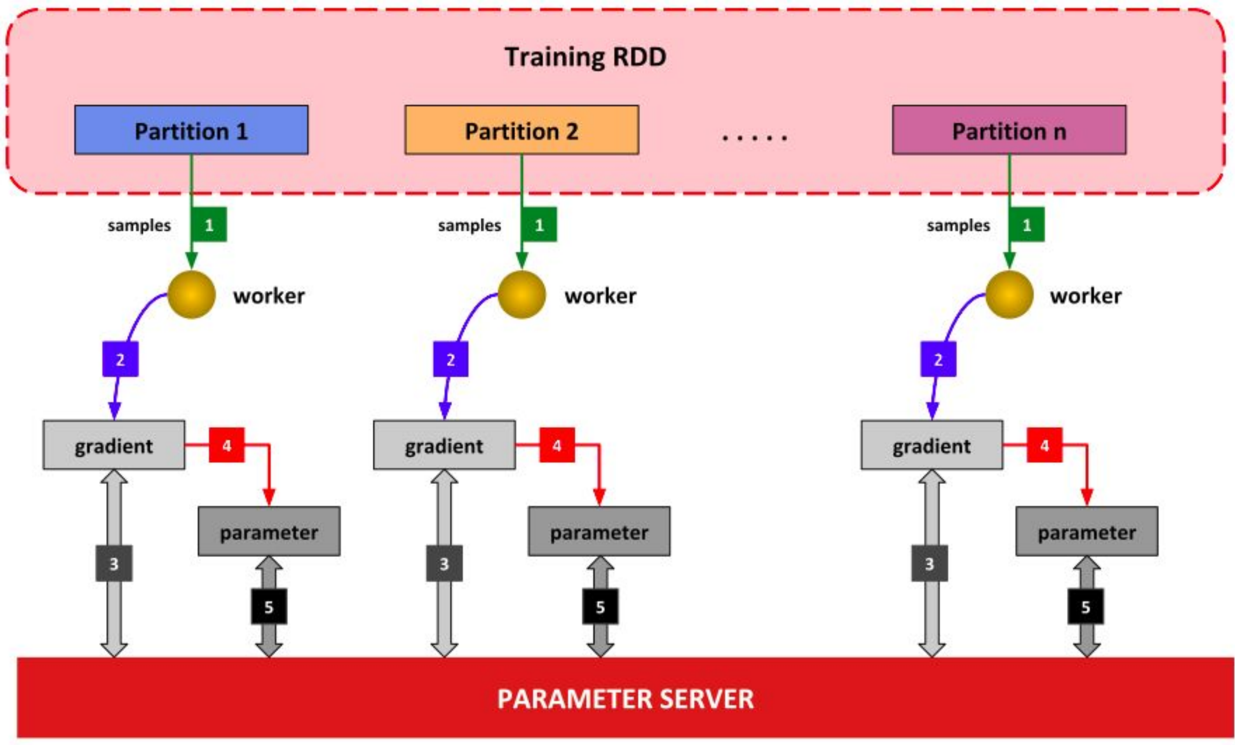


Figure 14: Communications in data-parallel training using the parameter server

Parameter synchronization is a very critical step as far as the performance is concerned. Most of the deep learning frameworks implement the above parameter server architecture. BigDL stands out from the rest in the way it performs parameter synchronization. BigDL uses standard Spark operations like shuffle and broadcast for achieving this. Figure 15 shows how parameter synchronization is achieved in BigDL.

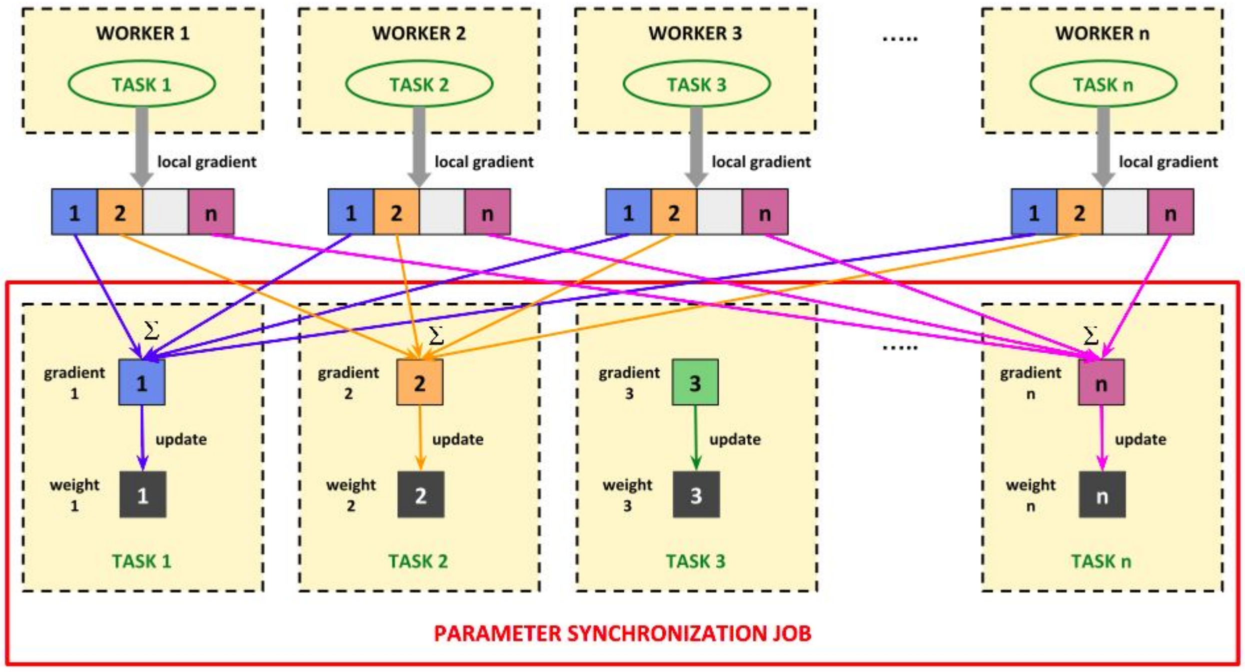


Figure 15: Parameter synchronization in BigDL

The Spark job has n tasks, each of which is assigned a unique ID ranging from **1** to **n**. After each task has finished computing the local gradients, it uniformly splits it into **n** parts as shown in the upper part of Figure 8. Now, another parameter synchronization job is launched with n tasks. Each task **i** of this parameter synchronization job is responsible for managing the **i**th partitions of the parameters. More specifically the **i**th partitions of the local gradients are aggregated by task **i** of the parameter synchronization job. Task **i** then updates **i**th the partition of the weights as depicted. As the next step, each task broadcasts the updated value of the weights and hence, the main job can see the updated values of the weights in the next iteration.

Let’s now take a look at the pseudo-code for Data-parallel training in BigDL is as follows:

```
for i = 1 to M do
    // model job
    for each task in the Spark job do
        read the latest weights
        get a random batch of data from local Sample partition
        compute local gradients using the local model replica on the
        sample partition
    end for
    // "parameter synchronization" job
    aggregate (sum up) all the gradients
    update the weights per specified optimization method
end for
```

The first 3 lines of the inner for loop are implemented using functional operations like map, filter and reduce, that are supported by Apache Spark. It is important to note that these operations are both data parallel (applied to each of the data partitions in parallel) and coarse grained (applied to multiple data items at once).

The pseudo-code for the parameter synchronization job is as follows:

```
for each task n in the "parameter synchronization" job do
    shuffle the n_th partition of all gradients to this task
    aggregate (sum up) these gradients
    update the n_th partition of the weights
    broadcast the n_th partition of the updated weights
end for
```

The shuffle and broadcast operations at the task side are implemented on top of the distributed in-memory storage of Spark. The tasks store the local gradients and the updated weights in the in-memory storage unit. The Spark tasks can remotely read this data with a very low latency.

The scalability of distributed training is heavily dependent on the efficiency of its parameter synchronization job. The efficiency of a parameter server is inversely proportional to the associated overheads. In an experiment carried out by Intel, the parameter synchronization overhead was computed by running ImageNet Inception v1 model training using BigDL on a large number of Intel Xeon servers. The results were very satisfactory. The overheads were approximately of the order of 7% of the average model computation.

In another study conducted by Cray, the number of worker nodes in the cluster were modified in the range of 16 to 256 to study the effect on performance. It was observed that there was a nearly linear increase in the speed up. The speed up obtained with 96 workers was approximately equal to 5.3 times the speed up observed with 16 worker nodes.

A class inheritance diagram has been added in the Appendix section.

Limitations

BigDL has numerous advantages, as discussed in the features section. The BigDL project is relatively new and suffers from having a few key limitations. A couple of known issues have been resolved as bug fixes. Some of them however persist. In this section, we will take a look at the limitations and a few known issues with BigDL.

- BigDL does not support model parallelism (.i.e. There is no distribution of the model among the different workers). In model parallelism, the same sample data is sent to all cores. Each of the worker nodes is responsible for the computation of a different parameter(s) or for handling different parts of the network. The worker nodes then exchange the parameter(s) values to finalize on the right estimate for all values. On the other hand, BigDL follows a data parallel training scheme where a replica of the network model is trained on a partition of the dataset.

  This data parallel approach is useful only when the number of nodes in the cluster is low. This could be attributed to the large scheduling overheads that come into the picture when we increase the number of worker nodes operating in the cluster. If there are too many nodes where there is computation happening (as is the case with BigDL), we may have to significantly reduce the learning rate to get a smooth training process. Put another way, increasing the computational nodes would increase the training time. However, this is not actually a serious limitation in practice. BigDL runs on Intel's Xeon CPU servers which have a large memory size. Hence, there won't be any problem in storing replicas of the large model at each worker.

- Currently, BigDL uses synchronous mini-batch stochastic gradient descent (SGD) for the training process. In mini batch gradient descent, the training set is divided into small batches. These batches are then used to compute the model error and update the coefficients. BigDl assumes that the size of the training set is an integral multiple of the total worker cores used for the job.

- As of now, BigDL loads all the training and validation data into the main memory of the worker nodes. The worker nodes then run the risk of going out of space should the dataset be significantly large. In most of the cases, the full training data is not needed for the process. A random subsample would be enough for training the model to obtain a comparable inferencing accuracy.

- Earlier, BigDL had a problem when the dataset size was too small. If the training data is cached before all the executor resources are allocated, Spark may not distribute the training data partitions in an even fashion. This leads to an imbalance in the tasks allocated. This was resolved in a bug fix, by increasing the waiting time of the scheduler before the resources were cached.
- The current BigDL design does not support dynamic resource allocation. Dynamic resource allocation has several advantages - optimum resource allocation and usage, reduced execution time, to name a few. In BigDL on Apache Spark, the number of workers remains the same after initialization. Consider for example, a Spark streaming application that meets certain timing performance in a dynamic fashion. In such a case, it would be beneficial if it was possible to dynamically alter the number of executors (worker nodes) depending on the performance of the previous batch.
- Currently, BigDL is only optimized for running on Intel's Xeon based CPU infrastructure. This is partly because of its dependence on Intel's MKL (math kernel library) for obtaining significant speedup.

Conclusion

In this report, we have described BigDL, a framework for distributed deep learning. The data parallel execution model of BigDL on Spark has been explained in detail in the section on Architectural design. A short note has been provided on its performance related aspects. BigDL allows users to build deep learning applications for big data using a single unified big data pipeline. This entire pipeline can be run directly on top of the existing big data systems in a distributed fashion. This is unlike the other conventional deep learning frameworks as it provides efficient and scalable training on top of the functional compute model of Spark. On Intel Xeon based infrastructure with multiple cores, use of BigDL is expected to provide a speed which is several orders of magnitude faster than its other counterparts. BigDL is a work in progress and the Analytics team at Intel is actively engaged in the project. BigDL however has a few limitations as mentioned in the Limitations sections. We can expect these to be fixed in the upcoming bug fixes. As of now, BigDL is highly successful and it is used by a lot of users to build new analytics and deep learning applications for the big data pipelines.

References

1. Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, and Guoqiong Song. 2019. BigDL: A Distributed Deep Learning Framework for Big Data. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '19). Association for Computing Machinery, New York, NY, USA, 50–60. (DOI: https://doi.org/10.1145/3357223.3362707)

2. Singh, Neelam & Garg, Neha & Mittal, Varsha. (2014). Big Data – insights, motivation and challenges.

3. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, USA, 10.

4. Jason (Jinquan) Dai, and Ding Ding. Very large-scale distributed deep learning with bigdl. o'reilly ai conference, san francisco. (2017).

5. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12). USENIX Association, USA, 2.

6. Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. in Proceedings of the 22nd ACM international conference on Multimedia. MM'14.

7. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. OSDI'16.

8. Collobert, Ronan and Kavukcuoglu, Koray and Farabet, Clément. Torch7: A matlab-like environment for machine learning. in BigLearn, NIPS workshop. (2011).

9. Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. International Conference on Machine Learning (ICML). (2018).

10. Tokui, Seiya and Oono, Kenta and Hido, Shohei and Clayton, Justin Chainer: a next-generation open source framework for deep learning in In Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS). (2015).

11. Kim, Hanjoo et al. “DeepSpark: Spark-Based Deep Learning Supporting Asynchronous Updates and Caffe Compatibility.” ArXiv abs/1602.08191 (2016): n. Pag.

12. Zhang,H., Zheng,Z., Xu,S., Dai,W., Ho,Q., Liang,X., Hu,Z., Wei,J., Xie,P., and Xing,E.P. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. in 2017 USENIX Annual Technical Conference (USENIX ATC 17). (2017).

Appendix

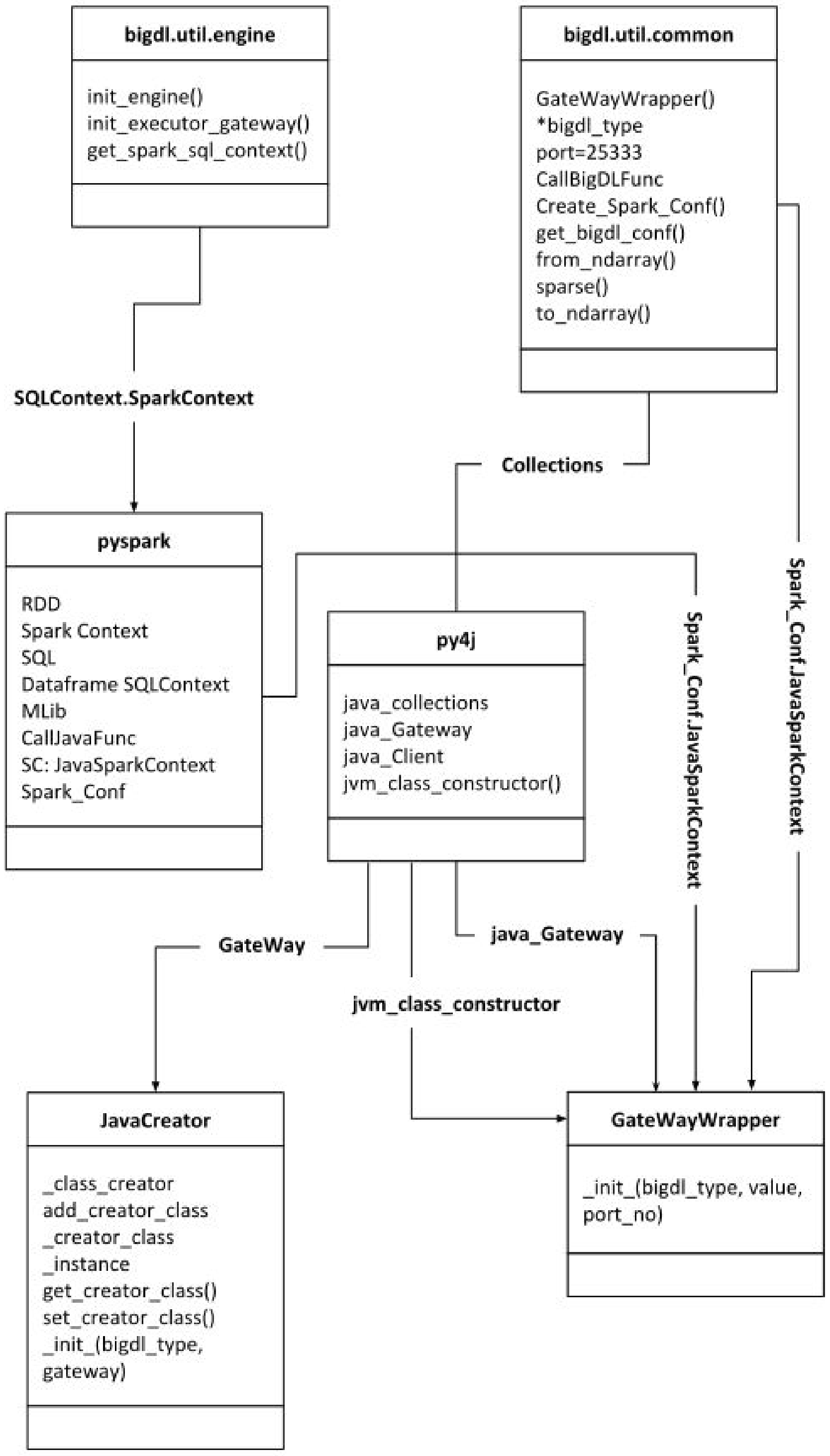


Figure 16: Class inheritance diagram