Georgios Atmatzidis
11 April 2023

Survey on Distributed Machine Learning

Abstract

Distributed Machine Learning is the idea of training machine learning models across multiple devices, rather than on a single machine. This technique is particularly useful for training models that generalise from a very big number of data, such as SVMs and Neural networks. Distributed ML allows faster and more-efficient training of these models. However, the distribution of data and computational load brings new challenges such as communication overhead, data inconsistency and fault tolerance. Companies have been attempting to address these issues by developing new frameworks and modifying distributed system technologies already in use. In this survey study, we will go over the existing issues with distributed training in more detail before examining some of the ideas that have been proposed to deal with the above problems.

Introduction

What's all the fuss about?

Traditionally, Machine Learning algorithms analyse datasets to produce solutions to real-world problems such as image and speech recognition, natural language processing, autonomous vehicles, and many others.
In the recent years, an explosion of data - the so called Big Data phenomenon- is observed due to the rapid development of new technologies and the growth rates of online systems. But the performance of traditional machine learning systems does not scale up when working with Big Data, because the datasets that are used to train the models can be in order of terabytes or even petabytes. So, there is a new need in the field for distribution. Machine Learning can benefit from distributed systems due to the advantages these systems provide such as parallel computation, data distribution and failure resilience. Therefore, new algorithms that take advantage of the above advantages should be proposed.
These new technologies should efficiently perform advanced machine learning without having to rely on expensive super machines.

Trying to design a distributed machine learning system

On the one hand there are a lot of different real-world problems that can be solved via machine learning and on the other hand there are a lot of methods, algorithms and options that can be used to solve these problems. Therefore when

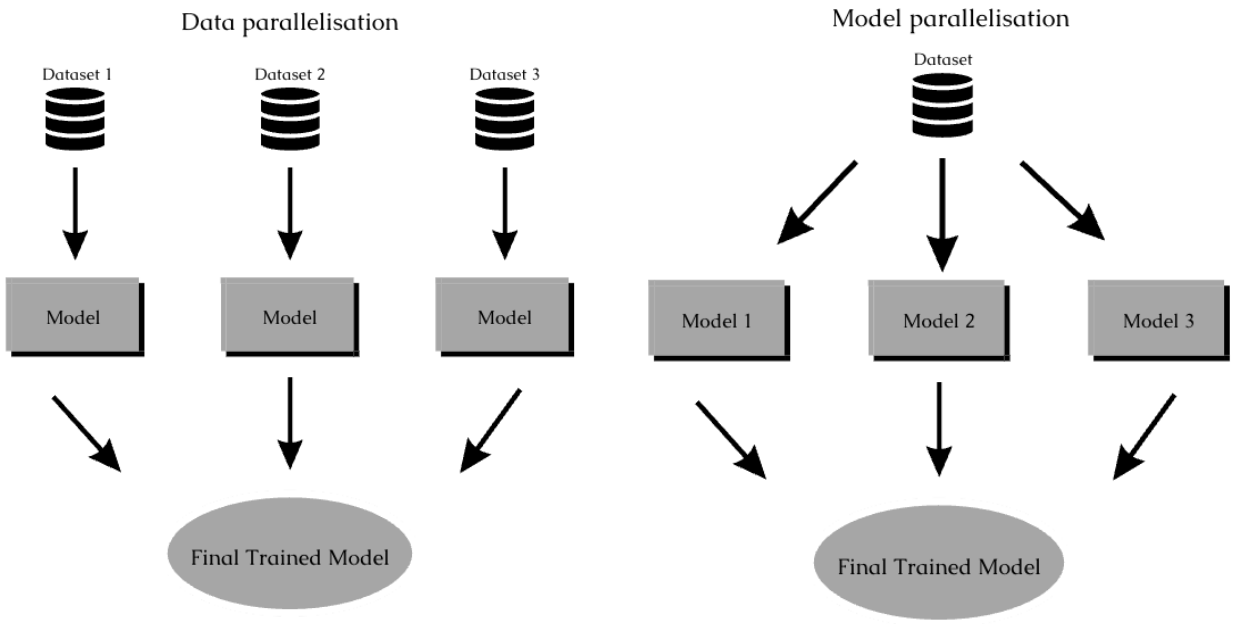
trying to design a distributed machine learning system one need to address these issues and make some architectural decisions based on its needs.

Parallelism in distributed machine learning

First, we need to decide which method of distribution we will use. Generally, there are two main ways to distribute on machine learning: parallelising the data or the model.

Using data parallelisation the data is partitioned to smaller datasets. The number of these datasets is proportional to the number of available workers. All workers execute the same algorithm to an equal-sized partition of the entire training data. The same model is available to all workers through centralisation or replication. The technique can be used with every ML algorithm known with the assumption that there is an independent and identical distribution assumption over the datasets.

Using model parallelisation all workers operate on different parts of the model but the entire dataset is available to them (copying the entire dataset). The final model is an aggregation of all separated models. A worker computes the gradients for a server based on its assigned model partition and data. This technique cannot be used to every machine learning algorithm, because the model parameters generally cannot be split up to sub models.



Choosing a machine learning algorithm

There is a wide criteria of ML algorithms to choose from. These algorithms can be categorised by the type of feedback needed while learning such as Supervised, Unsupervised or Semi-supervised learning. Also, ML algorithms can be

categorised based on the purpose they serve such as Classification, Clustering or Regression and much more. Last but not least, these algorithms can be categorised based on the method they use such as SVMs, Perceptrons, Neural Networks and plenty more.

Each category has its own specialties but in this particular paper an extensive analysis will not be made, as they are getting out of our subject.

Hyperparameter Optimization

The right selection of the parameter values used in ML algorithms has a significant impact on how well these algorithms perform. One must decide the optimal batch size, learning rate, epochs, initialisation and other factors for each different problem domain.

Optimisation is a stand-alone field in computer science that study algorithms that can be used to automatically optimise the parameters of these ML algorithms. These algorithms include: first and second order techniques, coordinate descent and several more.

The implementation of these algorithms are beyond the subject of this particular article and therefore will not be examined further.

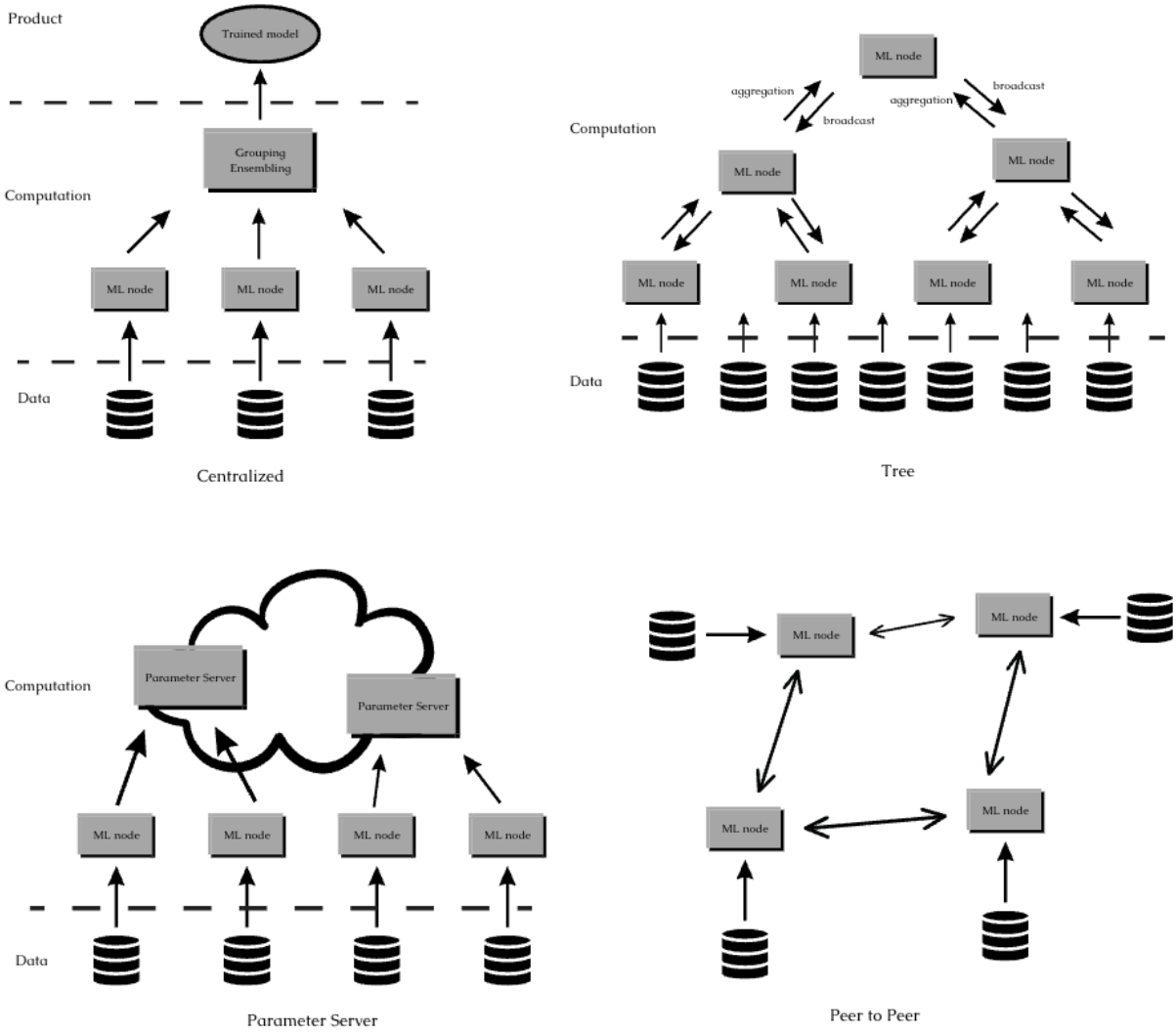
Topologies

One of the most important considerations when trying to build a distributed machine learning system is the topology in which the cluster of computers is organised. First of all a centralised solution is not even an option when data are inherently distributed or too big to store on single machines. So, we need a distributed organisation of the clusters.

The most popular topologies for distributed machine learning are: **Trees**, **Rings**, **Parameter Server** and **Peer-to-Peer**. There is a special mention in Parameter Server because various algorithms that will be analysed later in this paper use this technology:

Parameter Server: In the parameter server framework, a centralised server (or group of servers) maintains global shared parameters of a machine-learning model while the data and computation of calculating updates are distributed over workers. All model parameters are accessed through a key-value store. Workers upload their computed updated by the push operation and fetch up-to-date global weights by pull operation. The parameter server supports both model and data parallelism

An advantage is that all model parameters are in a global shared memory, which makes it easy to inspect the model. A disadvantage of this topology is that the parameter servers can form a bottleneck, because they are handling all communication. To solve this issue, there has been a development in techniques for



Different topologies on distributed machine learning

bridging computation and communication, some of which will be presented in detail later in this paper.

Communication

As it seems, distributing the data evenly across several machines gives the user a number of benefits regarding the speed of training and the accuracy of the outcome. The choice of distribution, however, can differ and it has direct impact on the amount of communication required to train the model.

When we are trying to build a distributed machine learning system, we aim for the best accuracy with the lowest computation and communication cost. Parallelising the learning phase can reduce the computation time, as long as the communication cost between the machines is not dominating.

There is always a trade-off between accuracy and communication cost when trying to solve this issue. For example, if one splits the data across the workers and each worker trains a separate model on a separate part of the dataset then the accuracy of the model will be reduced. Differently, by synchronising the separated models during training can increase the accuracy of the final model but leads to an increase of communication cost between the machines as the model size increases.

There are several techniques that enable the interleaving of parallel

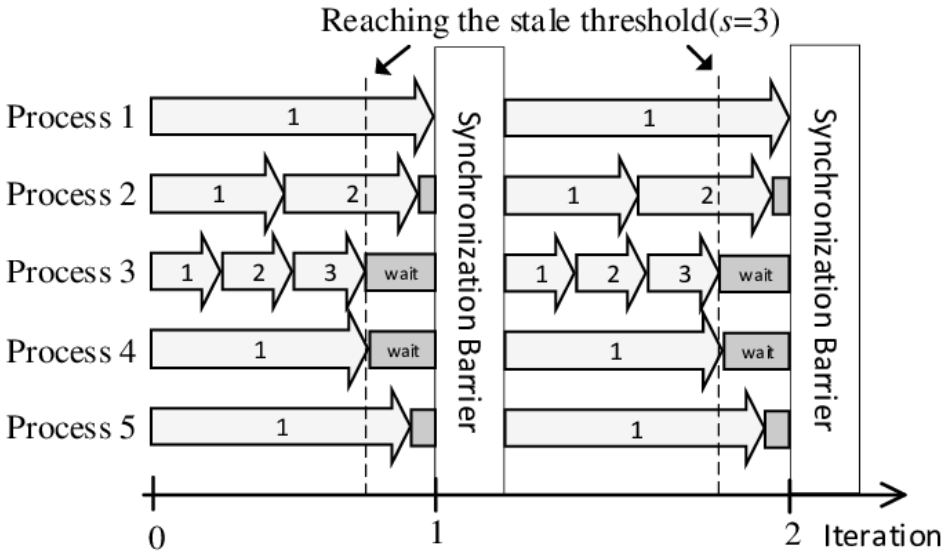
computation and inter-worker communication. These techniques trade off fast/correct model convergence with faster/fresher updates. Indicatively, some of these techniques are: **Bulk Synchronous Parallel (BSP)**, **Stale Synchronous Parallel (SSP)**, **Total/Barrierless Asynchronous Parallel.**

Below these is a summary for these techniques because various algorithms that will be analysed later in this paper rely on these techniques:

Bulk Synchronous Parallel (BSP): it is a simple technique where all workers send out their computed gradients to the parameter server and wait for each other at the end of every iteration for synchronisation. The parameter server receives the gradients and updates the global weights and then sends them back to the workers before the start of the new iteration. At the start of each iteration, all workers have the same global weights. BSP has the best accuracy but takes the most training time due to synchronisation at the end of each iteration.

Total/Barrierless Asynchronous Parallel (ASP): a simple technique that gives the permission to workers to work on their own and communicate without waiting on each other. Every worker send their computed gradients to the parameter server at each iteration but no synchronisation is required at all. At the start of each iteration, the global weights are inconsistent to all workers. Slower workers bring delayed or staled gradients to globally shared weights on the server. Therefore the delayed gradients introduce errors to the model. This technique unlocks the best possible speedup of the cluster and has the least training time but can converge slowly or incorrectly because errors can grow when the staled updates are from very old iterations.

Stale Synchronous Parallel (SSP): allows faster workers to move ahead for a certain number of iterations defined by the user (threshold s). If the number is exceeded then all workers are paused and synchronisation is forced on faster workers. The policy is to restrict the number of iterations between the fastest workers and the slowest workers. Workers train their models on cached versions of the data and commit changes on the end of a task cycle, which can cause slower workers to work on stale data. This technique relaxes the synchronisation overhead and can lead to a strong model convergence if done right. If not, then the convergence rates can quickly decline. SSP is an intermediate solution between BSP and ASP. It is faster than BSP and guarantees convergence, leading to a more accurate model than ASP.



Visualisation of the SSP

Natively Distributed Machine Learning Systems

The growing use of machine learning in numerous applications has led to the development of a number of natively distributed machine learning systems, that are based on certain distribution models. Below, we will talk about the most popular implementations and what optimisations have been proposed the last years.

Before we start

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimisation algorithm that is commonly used machine learning. The main idea is to find the parameters that will minimise a cost function by tweaking them iteratively. The optimal convergence is $O(1/\sqrt{K})$.

Synchronous Parallel Stochastic Gradient Descent

The parameter server keeps the model saved and every worker fetches the model and computes a small portion (batch) of stochastic gradients. After their computation, they push the computed stochastic gradients back to the parameter server. After that, the parameter server synchronises all the stochastic gradients and update their average into the saved model. The convergence rate is proven to be $O(1/\sqrt{nK})$.

Parallel Asynchronous Stochastic Gradient Descent and Parameter Servers

DistBelief and afterwards TensorFlow are two big projects that use this technique. In DistBelief, neural networks are represented as computational graphs. Each node implements an operation transforming the input to output. Every workers

executes the training of a small number of nodes and communication is only required when the output of a node is the input of another node (thus Asynchronous). The performance of this technique depends on the partition of the nodes to clusters. Locally connected models lend themselves better for model-parallelism because of limited cross-partition communication. For data parallelism, DistBelief uses a centralised Parameter Server to keep parameters and share them to worker's model replicas. The convergence rate is proven to be the same as S-PSGD, $O(1/\sqrt{nK})$.

AllReduce-SGD

This algorithm's philosophy is similar to the Synchronous-PSGD. However, it differs because there is no parameter server and all workers use the AllReduce method to synchronise the stochastic gradients. In one iteration, only $O(1)$ amount of gradient is sent/received per worker, but $O(n)$ handshakes are needed on each worker. The convergence rate is the same as S-PSGD. The idle time is also the same as S-PSGD because synchronisation still exists. Plus, due to $O(n)$ handshakes AllReduce is slow on high latency networks.

Decentralised Stochastic Gradient Descent

Every worker has its own local copy of the model. In each iteration, all workers compute stochastic gradients locally based on their own model and at the same time average their model with their neighbours. In the end of the iteration, the newly computed stochastic gradients are updated into the local models. All workers are connected with a network that forms a connected graph G. In an iteration, the busiest worker only sends/receives O(deg(G)) models and has O(deg(G)) handshakes per iteration. The idle time is still high in D-PSGD because all workers need to finish updating before stepping into the next iteration.

Parallel Asynchronous Decentralised Stochastic Gradient Descent

Why?

In a heterogeneous environment, asynchronous algorithms using parameter servers perform poorly because of two reasons. One, there is a communication bottleneck at parameter servers when workers are too many, and two, the algorithm converges very badly when traffic to parameter server is congested. Next up, there will be a extensive presentation of Parallel Asynchronous Decentralised SGD, an algorithm that is solid on heterogeneous environments, communication efficient and has the best possible convergence.

Problem definition

The distributed topology consists of n number of workers, that are represented as an unidirectional graph $G = (V, E)$ where set $V = \{1,2,..,n\}$ is the set of workers and set $E \subseteq VxV$ is the connections between the workers. Each worker represents a machine/CPU or even a sensor that has local data.

The loss function of each node is: $f_i(x) = E_{\xi \sim D_i} F_i(x; \xi)$

where:

i: is the index of a worker (thus worker i)

D_i: is a distribution of the local data of worker i

ξ: is a sampled data point in the distribution D_i

The overall optimisation problem that AD-PSGD algorithm has to solve is:

$$min_{x \in R^N} = E_{i \sim I} f_i(x) = \sum_{i=1}^{n} p_i f_i(x)$$

where p_i's define a distribution that is $p_i \geq 0$ and $\sum_i p_i = 1.$

The main idea is that the algorithm minimises the loss function $f(x)$ so that the faster workers have higher p_i values and therefore bigger percentage of updates. The abstract is that the algorithm exploits faster workers by letting them run more epochs that the slower workers (stragglers) in the same amount of time and as a result have a bigger impact.

Moreover, a point of interest in the above study is the distribution D_i. There are two strategies that can be applied to appropriately distribute data so that the algorithm can solve the above optimisation problem.

Strategy 1: Let $D_i = D$. In this way all workers can access the same all data.

Strategy 2: Split the data in a way where the portion of data is p_i on worker i and define D_i to be the uniform distribution over the assigned data samples.

Overview of the algorithm

Each worker i has a copy of the model x stored in his local memory and repeats the below operations:

1. Sample data: Sample a small batch of size M from the training data denoted by $\{\xi_m^i\}_{m=1}^M$.
2. Calculate gradients: Compute the stochastic gradient $\sum_{m=1}^{M} \nabla F(\hat{x}^i; \xi_m^i)$, using the sampled data and the local model of worker i.
3. Gradient update: Update the local model using the newly computed gradients by $x^i \leftarrow x^i - \gamma \sum_{m=1}^{M} \nabla F(\hat{x}^i; \xi_m^i)$, where γ is the learning rate.

4. Averaging: Select a random neighbour i' (using the unidirectional graph) and average the local model with the other worker's model. Specifically, $x^i \leftarrow \frac{x^i + x^{i'}}{2}$. Moreover, both model on both workers are updated to the averaged model, which means $x^i = x^{i'}$.

Notes

Note 1: In Step 2 pay attention to the fact that x^i and $\hat{x}^i$ are not particularly the same because x^i may be modified by other workers in step 4.

Note 2: The above procedure is asynchronous as since each worker runs on its own without the need on global synchronisation in the cluster. The model is constantly learning due to fast workers that compute and update gradients even if there are stragglers on the network.

Deadlock Avoidance

In step 4, the averaging procedure involves the update of two separate workers and therefore can cause a deadlock, if precautions are not taken.

How is that possible? Given three workers A, B and C, A sends its local model x_A to B and waits for x_B model from B. But B has already sent out x_B to C and waits of C's response. Finally, C has sent out x_c to A and waits for x_A from A.

Prevention: The topology graph is now bipartite and it consists of 2 disjoint worker sets A and P. The first set A is called Active and the second P is called Passive. Any edge of the graph connects one worker in A and one worker in P. Due to this technique all neighbours of an active worker can be passive workers and on the other side also neighbours of an passive worker can be active workers.
This implementation avoids deadlock and is consistent with the general algorithm.

Communication Topology

Variations: The basic topology of the algorithm can be ring-based. To improve the communication between workers the algorithm can converge equally good in a topology where each active worker communicates with a passive worker that is $2^i + 1$ hops away in the ring, where i is an integer from 0 to $log(n-1)$ (n is the number of learners).

Complexity: When using the ring topology the complexity for any pair to exchange information is $O(n)$. Using the above suggested topology the complexity improves to $O(log(n))$.

Improving the topology can lead to better scalability of the algorithm and better robustness in a slow or failed network because there are multiple routes for a worker to exchange its information.

Algorithm AD-PSGD

Required: Initialise local models $\{x^i\}_{i=1}^n$ with the same initialisation, learning rate γ, batch size M, and total number of iterations K.

1. For $k = 0,1,...,K$ do
2. Randomly sample a worker i_k of the graph G and randomly sample an averaging matrix W_k which can be dependent on i_k.
3. Randomly sample a batch $\xi_k^{i_k} = (\xi_{k,1}^{i_k}, \xi_{k,2}^{i_k}, \ldots, \xi_{k,M}^{i_k})$ from local data of the i_k-th worker.
4. Compute the stochastic gradient locally $g_k(\hat{x}_k^{i_k}; \xi_k^{i_k}) = \sum_{j=1}^{M} \nabla F(\hat{x}_k^{i_k}; \xi_{k,j}^{i_k})$.
5. Average local models by $[x_{k+1/2}^1, x_{k+1/2}^2, \ldots, x_{k+1/2}^n] \leftarrow [x_k^1, x_k^2, \ldots, x_k^n] W_k$
6. Update the local model $x_{k+1}^{i_k} \leftarrow x_{k+1/2}^{i_k} - \gamma g_k(\hat{x}_k^{i_k}; \xi_k^{i_k})$, $x_{k+1}^j \leftarrow x_{k+1/2}^j$, $\forall j \neq i_k$

Note: Steps 4 and 5 can run in parallel.

Convergence and Performance

It can be proven mathematically that if the iteration number is big enough the AD-PSGD's convergence rate is $O(1/\sqrt{K})$, which is the same or better than SGD and its other variations. This means that AD-PSGD is consistent with SGD. Also, It can be proven that linear speedups are achievable with respect to batch size, number of workers and staleness.

Concerning the speedup w.r.t. number of workers n workers will make the iteration number advance n times faster in the sense of wall-clock time, which means we will converge n times faster.

Concerning the speedup w.r.t. staleness, linear speedup is achievable as long as the staleness is bounded by $O(K^{1/4})$ and all other parameters are considered constants.

Comparison to other algorithms

To compare AD-PSGD to other algorithms we use experimental methodologies that have been conducted to evaluate AD-PSGD's convergence rate and speedup w.r.t epochs and runtime with other algorithms. For reference, the datasets used are CIFAR10 and ImageNet-1K, Torch-7 as the deep learning framework and MPI for communication. For CIFAR10 we evaluate VGG (communication intensive) and ResNet-20 (computation intensive). For ImageNet-1K we evaluate ResNet-50

Epochs: Using the same hyper-parameters for all algorithms, in communication intensive workloads and computation intensive workloads AllReduce-SGD, D-PSGD and AD-PSGD converge similar, while ASGD converges worse all w.r.t epochs. Moreover, AD-PSGD does not sacrifice test accuracy. For

bigger datasets, AD-PSGD converges similarly to AllReduce-SGD w.r.t epochs and better than D-PSGD.

Runtime: AD-PSGD converges faster than all other algorithms w.r.t runtime and achieves the best speedup, regardless of workload type (computation intensive or communication intensive) or communication networks (fast or slow).

Heterogeneous environment: In a heterogeneous environments things can get more complicated because the performance of computation or communication may vary due to different architectural decisions, resource-sharing and hardware malfunctions. Powerful algorithms like AllReduce-SGD and D-PSGD perform poorly when computation or communication speeds vary. Moreover, centralised asynchronous algorithms like A-PSGD perform also poorly when the parameter server's network links slow down. But, AD-PSGD localises the impact of slower workers or network links. The below experiments show that on AD-PSDG is robust against both heterogeneous computation and heterogeneous communication:

- **Slow computation:** When there is a very slow worker in the cluster AD-PSGD's convergence is robust. AD-PSGD can converge faster than AllReduce-SGD and D-PSGD by orders of magnitude.
- **Slow communication:** AD-PSGD is also robust when one worker is connected to slower network links, in contrast to centralised asynchronous algorithms where they use a larger communication period to overcome slower links, which significantly slows down the convergence.

Dynamic Stale Synchronous Parallel

Why?

Following the rising of Big Data, training a deep network has become a very time-consuming job. A single machine can't handle the learning of a large model over a large dataset. Distributed training and parallelisation come to solve this problem by using a parameter server. As it stands, when one is trying to train a distributed machine learning model, staleness threshold must be set along with other hyper-parameters by the user and must be found via trial-and-error. A DNN model involves many other hyper-parameters. When these parameters change, the same searching trials have to be repeated again to fine-tuning the staleness threshold. Also, a single fixed value may not be suitable for the whole training process. **Dynamic Stale Synchronous Parallel** improves the current Stale Synchronous Parallel (SSP) model, that we covered earlier, by dynamically determining the staleness threshold at the run time based on the statistics of the real-time processing speed of distributed computing resources.

Overview

Dynamic Stale Synchronous Parallel (DSSP) takes a range $[s_L, s_U]$, where s_L is the lower bound of given range and s_U is the upper bound, for the staleness threshold as input and computes an optimal threshold $s^* \in [s_L, s_U]$ for each worker during the run-time of the training. Therefore, each worker may have different threshold. DSSP tries to minimise the waiting time for the worker to synchronise with others, based on the iteration time taken from each worker. An iteration interval of a worker is the time period between two consecutive updates the server receives from the specific worker.

Algorithm DSSP

First, we propose the algorithm's implementation for worker p:

Worker p at iteration t_p

1. Wait until receiving OK from Server.
2. $pull$ weights w^s from Server.
3. Replace local weights w^{t_p} with w^s.
4. Gradient $g^{t_p} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \delta l_{loss}((x_i, y_i), w^{t_p})$
5. $push$ g^{t_p} to Server.

Note: In step 4 m is the size of mini-batch M and $(x_i, y_i) \in M$.

After receiving OK from the Server, worker p fetches the global weights w^s and replace the local ones w^{t_p} with the global ones. Next, a mini-batch of the partition is used to compute the gradients with the new weights. After computation is finished, worker p sends the gradients to the Server.

Secondly, we propose the algorithm's implementation for the server:

Server at iteration t_s

1. $t_p = t_p + 1$
2. Update the server weights w^{t_s} with g^{t_p}. If some other workers send their updates at the same time, their gradients are aggregated before updating w^{t_s}.
3. if $(r_p > 0)$ then
   $r_p = r_p - 1$
   Send OK to worker p
4. else
   Find the slowest and fastest workers based on array t
   if $(t_p - t_{slowest} \leq s_L)$ then
      Send OK to worker p

else

If t_p is the fastest worker then

$r_p \leftarrow \text{synchronisation\_controller}(clock_p^{push}, r_p)$

If $(r_p > 0)$ then

Send OK to worker p

Wait until the slowest worker sends the next push request(s) so that $t_p - t_{slowest} \leq s_L$. After updating the server weights w^{t_s} with (aggregated) gradients, send OK to worker p

Note 1: t_i stores the number of push requests received from worker i so far.

Note 2: r_p stores the number of extra iterations worker p is allowed beyond s_L, initialised to zero at the very beginning.

The server waits for push requests from workers. Once a push request from a worker p has arrived, it updates its weights with the gradients that came from worker p and then it checks if the worker p can continue working. To determine if a worker is allowed to proceed, the server keeps the number of push requests received from each worker t_p and finds the slowest worker. The server allows worker p to continue only if his t_p is no more than s_L iterations away from the slowest worker. Otherwise, that means that worker p is the fastest worker and the server calls the synchronisation controller procedure to determine whether it allows worker p to continue with extra iterations.

Next up, we propose the algorithm's implementation for synchronisation controller:

Synchronisation controller

1. $A[p][1] \leftarrow A[p][0]$
2. $A[p] \leftarrow push_p^t$
3. Find the slowest worker from table A
4. Compute the length of the latest iteration interval of worker p:
   $I_p \leftarrow A[p][0] - A[p][1]$
5. Compute the length of the latest iteration interval of the slowest worker:
   $I_{slowest} \leftarrow A[slowest][0] - A[slowest][1]$
6. Simulate next r_{max} iterations for worker p based on I_p and $A[p][0]$ by storing the r_{max} simulated timestamps in array Sim_p so that:
   - $Sim_p[0] \leftarrow A[p][0]$
   - $Sim_p[i] \leftarrow Sim_p[0] + ixI_p$, where $0 < i \leq r_{max}$

7. Repeat the above step for the slowest worker and store the r_{max} simulated timestamps in array $Sim_{slowest}$ with $Sim_{slowest}[0] \leftarrow A[slowest][0] + I_{slowest}$
8. Find the simulated time point r^* for the index of $sim_p[r]$ that minimises $|sim_{slowest}[k] - sim_p[r]|$ for all $k \in [0,r_{max}]$ and $r \in [0,r_{max}]$.
9. Return r^*

Note 1: Synchronisation controller accepts as **input** $push_p^t$**,** which is the timestamp of push request of worker p for sending its iterations t's update to the server.
Note 2: Synchronisation controller calculates a time point r^* in the range of $[0,r_{max}]$, which is the number of extra iterations worker p is allowed to run. r^* minimises the simulated waiting time of worker p.
Note 3: Table A stores the timestamps of two latest push requests by all workers, where $A[i][0]$ stores the timestamp of the latest push request by worker i and $A[i][1]$ stores the timestamp of the second latest push request by worker i.

Synchronisation controller uses the table A to find the length of the latest iteration interval of worker p and of slowest worker. Next, it simulates the next r_{max} iterations of worker p and the slowest worker, where r_{max} is the maximum number of extra iterations allowed for a worker to be ahead of the slowest worker beyond the lower bound of the staleness threshold and saves them in array Sim_p and $Sim_{slowest}$ respectively. Then, it computes a time point r^* that minimises the difference in simulations between the slowest worker and worker p.

Differentiation between workers

When worker p sends a push request, if $r_p > 0$ the server sends the confirmation to continue working back to the worker and decreases its r_p by 1. In this way, even if worker p is not the fastest worker in the current time period, as long as its $r_p > 0$ (due to being the fastest worker in a previous iteration), it can still perform extra iterations beyond s_L. Therefore DSSP has workers that have different thresholds, that change over time.

Convergence

It can be proven mathematically via the theorem of SSP that DSSP has a bound on regret. DSSP supports SGD convergence following the same conditions

as SSP and moreover has the same regret bound $O(\sqrt{T})$ as SSP, where T is the number of iterations. DSSP converges in theory as long as the range is constant.

Performance

To see the performance of the algorithm we study specific experiments that were conducted on DSSP and the other three distributed paradigms BSP, ASP and SSP. The experiments used CIDAR-10 and CIDAR-100 datasets on four IBM POWER8 servers with 4 NVIDIA P100 GPUs on each server, 512GB RAM and 2x10 cores. The connectivity was possible with Infiniband EDR.

The DNN models are AlexNet, ResNet-50 and ResNet-110. Based on the experiments, we observe two opposite trends of ASP, DSSP, SSP and BSP with respect to their performance. The trends can be classified by the architecture of DNNs: ones that contain fully connected layers and ones that do not:

For fully connected DNNs: DSSP generally converges faster than BSP, ASP and SSP with a higher accuracy even though BSP can eventually reach the highest accuracy if it is given more training time.

For not fully connected DNNs: DSSP features a distinct better accuracy than BSP and slightly better accuracy than SSP.

DSSP is in general more stable on either homogeneous or heterogeneous environments compared to ASP, BSP and SSP, because it has shown that is more stable on both environments. In a heterogeneous environment, DSSP gives significant improvement than SSP and BSP with mixed models of GPUs, converging much faster to a higher accuracy. DSSP ensures the convergence of DNNs by limiting the staled delays, unlike ASP.

General Purpose Distributed Frameworks

Distributed Machine Learning can be seen as a problem of processing a large set of data on a cluster of machines. This problem is more general and has been studied further throughout the years. So, there are general-purposed distributed platforms that address this problem with practical implementations and can be a solid foundation for distributed machine learning. Below there will be an extensive reference to Apache Spark and especially to its distributed machine learning library called MLlib.

Apache Spark

Apache Spark is a popular open-source platform for processing large-data. It is a powerful and efficient in-memory solution for distributed processing. It has low latency and can offer powerful parallel processing. It is fast becoming the mainstream distributed engine of advanced data analytics on big data sets.

Why Spark? Apache Hadoop and MapReduce, which is another distributed computational framework, present high latency due to disk persistent write

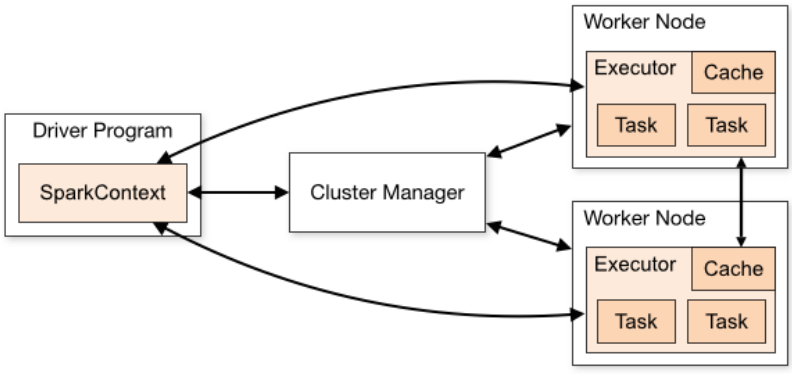
operations during the phases of the execution. In iterative workloads, such as Machine Learning, intermediate computational results have to store on disk and this fact greatly increases disk accesses making the execution slower. Furthermore, one must take into account the map jobs overhead of MapReduce and the fault-intolerance due to insufficient memory capacity. Hence, Hadoop lacks the ability to perform real-time analytics and therefore machine learning iterative algorithms.

Spark comes into play to overcome the inabilities of Hadoop. The crucial difference of Spark is the data structure that it uses called Resilient Distributed Dataset (RDD). An RDD is a set of elements that are separated across nodes of the cluster that can be operated on in parallel. RDDs can be created by being referenced to any Hadoop-supported file system such as HDFS, Amazon S3 or HBase and transforming them to RDD. Users can select to keep a RDD in memory, allowing it to be reused across parallel operations.

Moreover, in the case of node failures, RDDs automatically recover. Each RDD keeps a graph that shows what transformations have been executed on it. This lineage graph ensures that, if some data are lost, Spark can trace the path the RDD has followed from the lineage graph and recalculate any lost data. It is important that the lineage graph does not contain cycles (is a Directed Acylic Graph).

Sometimes, a variable needs to be shared across tasks, or between the driver program. Spark allows users to use shared variables, that can be used in parallel operations. There are two types of shared variables: *broadcast variables*, which can be used to cache a value in memory on all nodes, and *accumulators*, which are variables that are only “added” to, such as counters and sums.

The workflow of Spark consists of the driver program, the cluster manager and the worker nodes. SparkContext is an object that runs on the driver program and connects to a cluster manager (either Spark’s standalone or YARN on Hadoop) to allocate resources across applications. If the connection is successful then Spark demands workers to compute and store data for the driver program. The key factor of Spark is that it runs every job as an independent set of processes on a cluster.



Example of a Spark cluster

MLlib

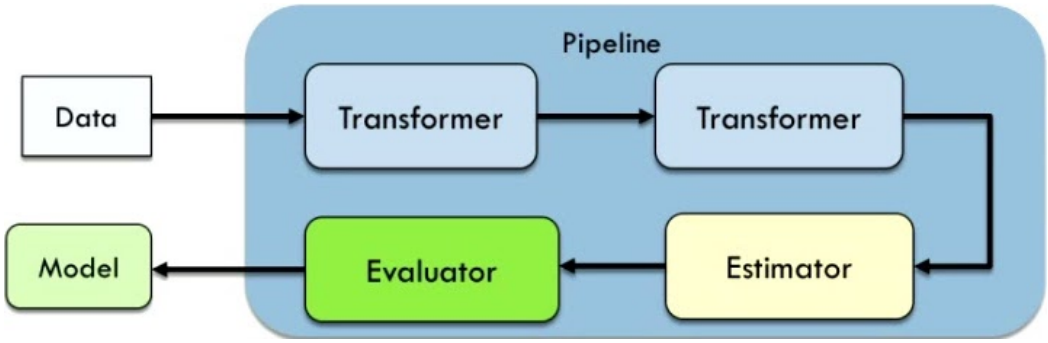
Features

Following the explanation of Apache Spark, we introduce MLlib, Spark's library for distributed machine learning.

Methods: MLlib consists of standard machine learning algorithms used for classification, regression, decision trees, clustering, and topic modelling. The framework uses faster, distributed versions of linear models, naive Bayes, decision trees, alternating least squares with feedback for filtering, k-means and dimensionality reduction. It can also offer a wide option of statistical analysis, distributed linear algebra and optimization algorithms.

Optimisations: The main advantage of MLlib is that it includes many optimisations to standard algorithms for more efficient predictions. For example, Alternating Least Square (ALS) is a matrix factorisation algorithm that is used for collaborative filtering problems. MLlib makes careful use of blocking in ALS to reduce JVM garbage collection overhead and to leverage higher-level linear algebra operations. Another example is in decision trees where MLlib executes a discretisation in data to reduce communication costs. Moreover, in tree ensemble, which is a machine learning technique for supervised learning, MLlib parallelise learning both within trees and across trees, so the weak learners can be found faster and the aggregation is faster and more accurate. In linear regression, workers use fast C++ based linear algebra to parallelise gradient computation for optimisation algorithms for the model production. Last but not least, many other algorithms benefit from the parallel nature of Spark minimising the distribution between driver and workers cost.

Pipeline: Another key feature of MLlib is its pipeline API. MLlib revolves around the concept of pipelines, which allows users to combine multiple algorithms represented as an array of stages in a single pipeline. MLlib distributes these stages to clusters of machines for computation.



Pipeline workflow of MLlib

In machine learning data pre-processing is a known procedure and it often involves a sequence of data pre-processing, feature extraction, model fitting, and validation

stages. MLlib includes a package to resolve these concerns named spark.ml. This package contains a high-level unified API, that simplifies the transformation of data before the actual processing. Also, it allows users to use their own specialised algorithms.

Member of Spark: MLlib enjoys many benefits from Spark, since it is a part of it. Improvements in low-level components of Spark often translate into performance gains in MLlib, without any direct changes to the library itself. Moreover, due to Spark's iterative nature the development of implementations of large-scale machine learning algorithms becomes easy and efficient since they are also typically iterative.

Performance and Scalability

In the next section we demonstrate various experiments that were conducted to measure the performance on speed and scalability of MLlib and to justify its existence and use. The experiments contain several machine learning algorithms that execute on very large databases.

First of all, the experiments use a common strategy in the training of the model: The datasets are split randomly and 70% of the dataset is used for training and the other 30% for evaluation. Moreover, all training and testing operations were executed in-memory due to Spark' advantages that were mentioned earlier.

The pipeline, as mentioned earlier, consists of two stages:

- **Stage 1: Classification Evaluator**, that uses indexes to evaluate the classification model.
- **Stage 2: Classifier**, that specifies which machine learning algorithm will be used.

The machine learning algorithms that will be used are Decision Tree, Logistic Regression and Naïve Bayes. To evaluate the performance of a classifier the experiments use the Positive Predictive Value (PPV) metric:

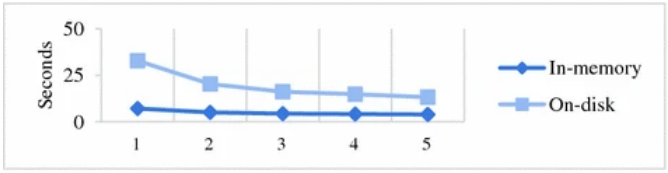
$$PPV = \frac{TP}{TP + FP}$$

where TP=True Positive and FP=False Positive

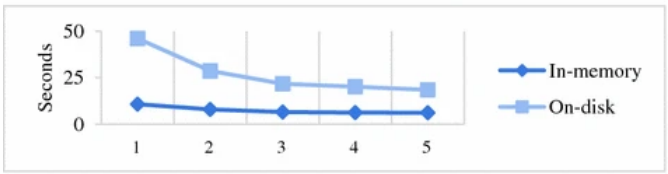
The experiments use EC2 instances of Amazon Web Services for worker nodes, that are memory optimised. A general-purposed driver exists on the cloud-based Databricks platform. Spark's hardware instances used in the experiments are identical and are not changing throughout the experiments.

The first experiment uses the H-1B visa petitions dataset (3 million records of data), while the other uses the Fire Department calls for the Service of San Francisco (4.6 million records).

The first figures show the difference in time between one to five worker nodes and between in-memory and on-disk execution:

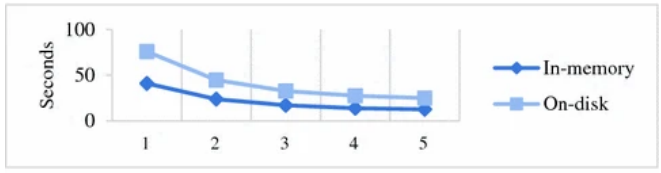


(a) Logistic Regression

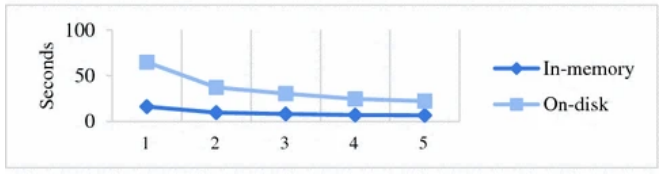


(b) Decision Tree

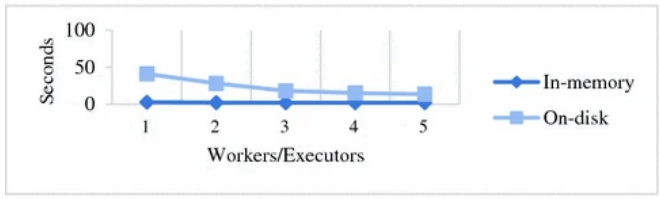
(c) Naïve Bayes



(a) Logistic Regression



(b) Decision Tree



(c) Naïve Bayes

Experiment 1		Experiment 2	
Learning Model	**Classification Accuracy**	**Learning Model**	**Classification Accuracy**
Decision Tree	87%	Decision Tree	73%
Logistic Regression (multinomial) Maximum iteration = 3	87%	Logistic Regression (multinomial) Maximum iteration = 20	69%
Naïve Bayes (multinomial) Smoothing = 1.0	56%	Naïve Bayes (multinomial) Smoothing = 500.0	52%

In conclusion, MLlib in Databricks using instances of EC2 of AWS is a proposed system for distributed machine learning on Big Data. The above experiments demonstrated that parallelising executions on-memory can drastically improve the speed of the system. Moreover, in terms of scalability the timing analysis shows that satisfactory speedups can be attained by increasing the number of workers in the cluster.

References

(1) Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. ACM Comput. Surv. 53, 2, Article 30 (March 2021), 33 pages. https://doi.org/10.1145/3377454

(2) X. Zhao, A. An, J. Liu and B. X. Chen, "Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning," 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 2019, pp. 1507-1517, doi: 10.1109/ICDCS.2019.00150.
(3) Lian, X., Zhang, W., Zhang, C. & Liu, J.. (2018). Asynchronous Decentralized Parallel Stochastic Gradient Descent. Proceedings of the 35th International Conference on Machine Learning, in Proceedings of Machine Learning Research 80:3043-3052 Available from https://proceedings.mlr.press/v80/lian18a.html.
(4) Abdel Hai, A., Forouraghi, B. (2018). On Scalability of Distributed Machine Learning with Big Data on Apache Spark. In: Chin, F., Chen, C., Khan, L., Lee, K., Zhang, LJ. (eds) Big Data – BigData 2018. BIGDATA 2018. Lecture Notes in Computer Science(), vol 10968. Springer, Cham. https://doi.org/10.1007/978-3-319-94301-5_16
(5) Meng, Xiangrui, et al. "Mllib: Machine learning in apache spark." *The Journal of Machine Learning Research* 17.1 (2016): 1235-1241.