Parallel and Distributed Machine Learning Techniques for Anomaly Detection Systems

Iulia Khlevna and Bohdan Koval

Taras Shevchenko National University of Kyiv, Volodymyrska str., 60, Kyiv, 01033, Ukraine

Abstract

Efficient processing of extensive datasets is crucial in data-driven applications, particularly for anomaly detection. This article explores the application of parallel and distributed machine learning techniques to enhance anomaly detection systems. Parallel training, including model and data parallelism, significantly reduces processing times in data preprocessing, feature engineering, and model training. Experiments demonstrate notable efficiency improvements, especially for extensive datasets. Distributed training is crucial in scenarios with time-intensive training, storage constraints, data localization, or RAM limitations. The approach to parallelize learning process has been demonstrated, which allowed to speed up the processing time by 2x via allocating 4x resources. Also, it depicts how the process can be spread across multiple machines using distributed techniques. In conclusion, this article emphasizes the pivotal role of parallel and distributed machine learning techniques in accelerating the development of anomaly detection systems. These techniques empower organizations to address challenges posed by extensive datasets and real-time data streams effectively. As data-driven applications evolve, these advanced computing methods promise more effective and timely responses to anomalies, solidifying their significance in the era of big data.

Keywords [1]

machine learning, distributed computing, parallel processing, big data, scalable systems

1. Introduction

In traditional machine learning approaches, data processing typically revolves around a single computational process. For instance, in widely-used machine learning libraries like scikit-learn, the default setting often employs a single processing core. While this method works effectively for relatively small datasets, typically measured in kilobytes or megabytes, it becomes inefficient and impractical when dealing with substantial volumes of data, particularly in the gigabytes or even terabytes range. Notably, the primary issue here is the prolonged processing times, especially during the training phase.

In modern data-driven applications, the need to analyze and extract insights from large datasets has become increasingly common. Anomaly detection systems, which are essential for identifying subtle irregularities or deviations, require the capability to process vast amounts of data.

An urgent scientific task based on the above rises, which consists of the development of the concept to tackle the significant computational challenges posed by such large datasets. This has given rise to the concept of parallel and distributed machine learning, which leverages the power of multiple processing units or distributed computing resources to speed up the analysis of massive datasets and enhance the overall efficiency of anomaly detection systems.

In this article, we explore the world of parallel and distributed machine learning and its application in anomaly detection systems. We will examine the theoretical foundations, practical implementations,

Information Technology and Implementation (IT&I-2023), November 20-21, 2023, Kyiv, Ukraine
EMAIL: yuliya.khlevna@gmail.com (Iulia Khlevna); bohkoval@gmail.com (Bohdan Koval)
ORCID: 0000-0002-1807-8450 (Iulia Khlevna); 0000-0002-3757-0221 (Bohdan Koval)

and the numerous advantages it brings to the complex task of detecting anomalies in extensive datasets. By adopting these advanced computational techniques, our goal is to alleviate the time and resource constraints that have traditionally hindered real-time anomaly detection in large-scale data environments.

2. Literature overview

Ben-Nun and Hoefler, in "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis" [1], offer a profound exploration into the intricacies of parallel and distributed deep learning. Their work dissects the critical issue of achieving concurrency in distributed settings, a fundamental aspect relevant to anomaly detection systems.

Aach, Inanc, Sarma, Riedel, and Lintermann, in "Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks" [2], delve into the performance analysis of distributed deep learning frameworks. With a focus on convolutional neural networks, this study investigates the scalability and efficiency of distributed deep learning algorithms - a pivotal area of study for implementing anomaly detection systems at scale.

In the context of real-time data streams, Esmaeilzadeh, Salajegheh, Ziai, and Boote, in "Abuse and Fraud Detection in Streaming Services Using Heuristic-Aware Machine Learning" [3], provide practical insights into the application of heuristic-aware machine learning techniques for abuse and fraud detection. Their work demonstrates the importance of heuristic-aware approaches in identifying anomalies in dynamic data sources, a crucial component of anomaly detection systems.

"Parallel and Distributed Training of Deep Neural Networks: A brief overview" by Attila Farkas, Gábor Kertész, Róbert Lovas [17] focuses on the parallel and distributed training of deep neural networks, a fundamental topic in modern machine learning. The authors likely discuss techniques and methods for training deep neural networks across multiple processors or nodes to expedite the training process and handle large datasets efficiently. Parallelization and distribution are crucial for making deep learning models practical for real-world applications.

In "Toward Parallel and Distributed Learning by Meta-Learning" by Philip K. Chan and Salvatore J. Stolfo [18], the authors explore the concept of meta-learning in the context of parallel and distributed machine learning. Meta-learning involves developing models that can adapt to new tasks quickly, and the paper likely discusses how this concept can be used to enhance the scalability and efficiency of machine learning in parallel and distributed systems. It may propose methods and frameworks for leveraging meta-learning to improve anomaly detection.

The work authored Ron Bekkerman, Mikhail Bilenko, John Langford named "Scaling up Machine Learning: Parallel and Distributed Approaches" [19] deals with the scaling up of machine learning using parallel and distributed approaches. The authors probably discuss a wide range of techniques and strategies for handling large datasets and complex models in parallel and distributed environments. Such methods are essential for anomaly detection systems, which often deal with massive amounts of data and require efficient algorithms to identify anomalies.

The research "Boosting Algorithms for Parallel and Distributed Learning" by Aleksandar Lazarevic and Zoran Obradovic [20] delves into the application of boosting algorithms in the context of parallel and distributed learning. Boosting is a machine learning ensemble technique that combines multiple weak learners to create a strong learner. The paper probably explores how boosting algorithms can be adapted and optimized for parallel and distributed systems to improve anomaly detection performance.

Collectively, these works contribute to our comprehensive understanding of parallel and distributed machine learning. They address various facets of the topic, ranging from the optimization of concurrency to patterns, performance analysis, and real-world applications in distributed environments. As the foundation for our investigation into parallel and distributed machine learning for anomaly detection systems, these studies inform the development of scalable and efficient solutions to the challenges posed by large-scale datasets and real-time data streams.

The examination of existing literature has revealed the critical importance of research endeavors that focus on accelerating the training process of machine learning models, particularly in the context of today's era characterized by the prevalence of big data and data-intensive applications. This underscores

the necessity of exploring techniques and strategies aimed at expediting the learning phase, as it directly addresses the challenges posed by the vast volumes of data that modern applications deal with.

3. Purpose and objectives of the research

The purpose of this research is to explore how parallel and distributed machine learning techniques can be harnessed to make anomaly detection systems operate with greater speed and efficiency. We seek to comprehend how these techniques can be practically integrated into various critical domains such as fraud detection, cybersecurity, and healthcare, where identifying anomalies is of paramount importance. Our primary objectives in this endeavor encompass a comprehensive understanding of parallel machine learning, which involves studying the methods through which computational tasks are divided and assigned to multiple processors or nodes. This distributed approach holds the promise of expediting the training process and enabling the scalability of anomaly detection systems.

Furthermore, we aim to delve into the intricacies of distributed machine learning, which extends beyond the mere distribution of computational workloads. It entails the dispersion of data and computation across multiple machines or clusters, ultimately contributing to more efficient learning processes, scalability, and adaptability. A crucial aspect of our research is the quantitative assessment of the actual speed improvement achieved through the application of parallel and distributed machine learning techniques. This necessitates rigorous benchmarking and performance evaluation to precisely measure the efficiency gains and the reduction in processing time.

To achieve this goal, the following tasks must be solved:

1. Differentiate parallel and distributed machine learning.
2. Formulate parallel machine learning technique and analyze its performance
3. Elaborate on distributed machine learning technique (where the process is spread not across multiple cores, but physical machines over network) and how we can apply it.

Table 1

Comparison of parallel and distributed computing.

Parallel Computing	***Distributed Computing***
Concurrency is achieved through simultaneous execution of numerous operations	System components are geographically distributed across distinct locations
A solitary computing unit is sufficient to execute the tasks	Utilizes a network of multiple distinct computing units
Concurrent operations are performed by multiple processors within a single system	Concurrent operations are distributed across multiple discrete computing systems
May encompass shared or distributed memory resources	Solely employs distributed memory resources
Inter-processor communication typically occurs via a shared memory bus	Communication between computing units relies on message passing protocols
Enhances the overall performance of a system	Enhances system scalability, fault tolerance, and resource sharing capabilities

When dealing with a large dataset and long processing times using classic machine learning techniques, there are several strategies you can employ to significantly reduce processing time and improve efficiency, some of which are:

- *Parallelism*: Utilize parallel processing or multi-threading capabilities available in your programming environment. Libraries like scikit-learn have options for parallel processing. You

can make use of this feature to distribute the computation across multiple CPU cores, which can drastically reduce processing time.

- *Distributed Computing*: If your dataset is extremely large and cannot fit in memory, consider using distributed computing frameworks like Apache Spark. These frameworks can distribute the data and computation across a cluster of machines.

Below it the comparison of these 2 approaches (Table 1):

In the realm of machine learning, parallel and distributed computing techniques have garnered increasing attention due to their significant impact on the efficiency and scalability of anomaly detection systems.

4. Parallel machine learning algorihms

4.1. Understanding parallel training

Parallelism, at its core, is a strategic framework aimed at overcoming the limitations posed by the size of large machine learning models and enhancing the overall training efficiency. Within this framework, there are two primary types of parallelism, each tailored to distinct objectives. The first is model parallelism, which focuses on dividing a large model into smaller, manageable components that can be trained concurrently. The second is data parallelism, a technique where subsets of the dataset are distributed to multiple processing units for simultaneous training [4].

When you parallelize a task, the idea is that if you double the number of processing elements, it should take only half the time, and if you double it again, the time should halve once more. However, in reality, only a few parallel algorithms can achieve this perfect speedup. Most of them show nearly linear speedup when you have a small number of processing elements, but as you add more, the speedup tends to level off and remains relatively constant.

The possible increase in the speed of an algorithm when it runs on a parallel computing platform is determined by Amdahl's law:

$$S_{latency}(s) = \frac{1}{1 - p + p/s} = \frac{s}{s + p(1 - s)},$$

where $S_{latency}$ represents the potential reduction in the time it takes to complete the entire task, s the reduction in time specifically for the part of the task that can be done in parallel, p is the portion of the total task time that is spent on the part that can be parallelized before parallelization [16].

Since $S_{latency} < 1/(1 - p)$, it indicates that a small portion of the program that cannot be parallelized will restrict the maximum speedup achievable through parallelization. A program that tackles a substantial mathematical or engineering problem usually includes both parts that can be parallelized and parts that must be executed sequentially. If the non-parallelizable segment of a program makes up 10% of the total runtime (with p = 0.9), the speedup cannot exceed 10 times, regardless of how many processors are added. This sets a limit on the benefit of adding more parallel execution units. In simple terms, if a task cannot be divided due to sequential constraints, putting in more effort won't accelerate the schedule.

The picture (Fig. 1) provides an insight into the strategies employed for data and weight partitioning within the context of Google Switch Transfers. The Switch Transformer represents an innovative adaptation of the traditional transformer architecture by introducing the concept of a switch feed-forward neural network (FFN) layer. The fundamental departure from convention lies in the fact that, unlike the typical transformer FFN layer, the switch layer incorporates multiple FFNs, which are referred to as "experts." In the operation of this layer, each individual token undergoes a distinct process. It commences its journey by traversing a router function, responsible for directing the token to a specific FFN expert. Therefore, this process can be parallelized [6].

In this representation, each 4×4 grid demarcated by dotted lines symbolizes a collective of 16 cores. The shaded squares within these grids represent the data residing on each core, which can include either model weights or a batch of tokens. This illustration serves to elucidate how model weights and data tensors are partitioned for each strategy [7].

The first row in this visual representation offers a glimpse into the division of model weights among the cores. Different-sized shapes in this row signify the presence of larger weight matrices, notably in

the Feed Forward Network (FFN) layers, denoted by more extensive "dff" sizes. Each color attributed to the shaded squares designates a distinct weight matrix. It's noteworthy that the number of parameters allocated to each core remains constant, although larger weight matrices entail a more substantial computational workload for each token.

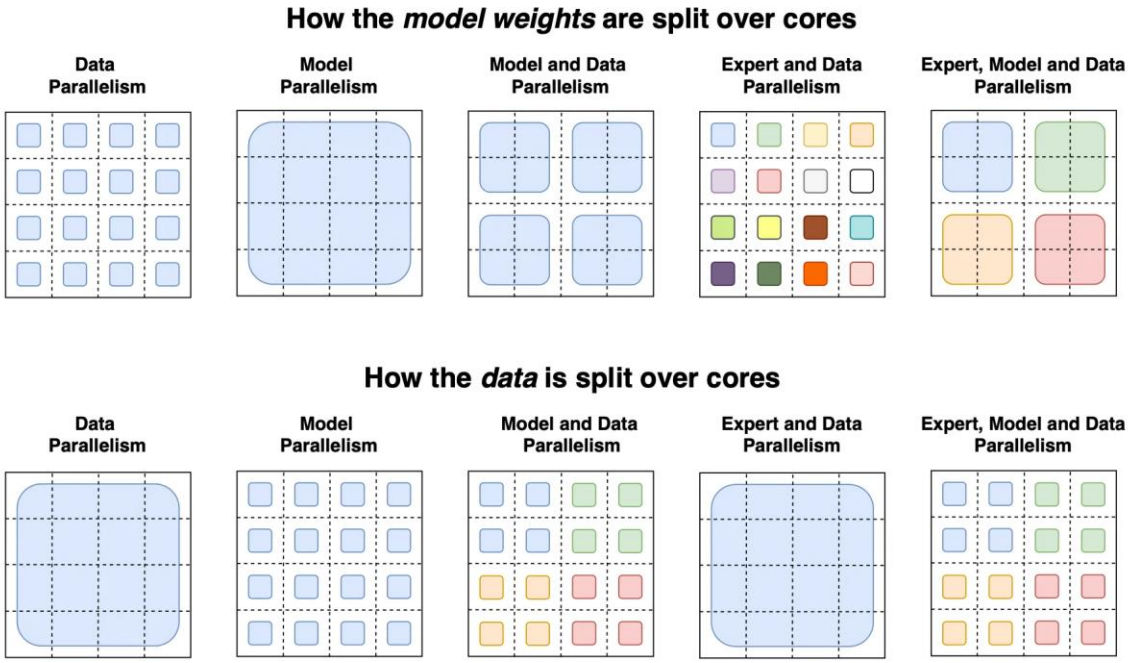

Figure 1: Examples of model weights and data parallelization [5].

Moving on to the second row, it unveils the manner in which a data batch is distributed across the cores. In this case, each core accommodates an identical number of tokens, ensuring a uniform memory utilization across all the partitioning strategies. These strategies exhibit varying properties, allowing for each core to harbor tokens in the same or different colors, resulting in distinctive arrangements across the cores [8].

This depiction serves as a visual guide to comprehend the intricacies of data and weight partitioning strategies in Google Switch Transfers. It showcases the flexibility and versatility in distributing model weights and data, crucial for optimizing the efficiency and performance of deep learning models.

4.2. Applying parallel training

Let’s apply parallel machine learning and compare to sequential processing. Speed is of the essence when developing and fine-tuning machine learning models, especially in constrained environments like kernels competitions. Faster iterations over various ideas can lead to better-performing models. In the realm of anomaly detection, the need for rapid data processing and model training is particularly critical. This is where parallel and distributed computing comes to the rescue.

In the forthcoming example, we will undertake a comparison between the sequential execution of a model and its parallel counterpart utilizing four threads. This comparison will be conducted within the Python environment, with particular emphasis on its C implementation.

Parallel processing can significantly enhance the speed of data pipelines, and in Python, this can be easily achieved by dividing the main input data into parts and concurrently applying functions to these segments. Two powerful libraries, “multiprocessing” and “joblib”, facilitate this parallelism. Let's take a closer look at how parallel data processing can be implemented in a kernel environment.

In the kernel environment, where efficiency is paramount, a timer function is employed to measure the execution time of specific operations. This allows us to precisely gauge the impact of parallel processing on our data processing pipelines. The *split_df* function is used to divide the main DataFrame

(*df_*) into multiple smaller DataFrames, equal in size. This makes it easier to apply parallel functions to these mini-DataFrames.

```
def timer(name):
    t0 = time.time()
    yield
    print('{0} done in {1:.3f} s.'.format(name, time.time() - t0))

def split_df(df, num_splits):
    # Split the DataFrame into 'num_splits' equal parts
    # ...
    return df_list
```

For instance, you can split the main DataFrame into four parts, apply functions to these sub-DataFrames in parallel, and later reassemble them to ensure the integrity of the data.

Next, we implement the following functions:

- `datetime_proc`: Converts the 'time' column into datetime format for more efficient time-based analysis.
- `create_time_resolutions`: Creates features that capture information about the time dimension, such as hour, day, and week of the year.
- `rename_columns`: Renames columns after grouping to facilitate easier merging and access.
- `create_grouped_df` and `create_grouped_df_shifted`: These functions create grouped features based on specific criteria, allowing for more detailed analysis and anomaly detection.

By using parallel processing, we can significantly reduce the time it takes to transform and prepare your data, allowing you to experiment with various anomaly detection techniques more efficiently.

Next, let's examine the process of creating datetime features, comparing sequential and parallel approaches:

```
with timer('datetime processing:'):
    df_ = create_time_resolutions(df_)
```

Datetime processing: done in 19.341 seconds.

```
with timer('pool datetime processing:'):
    with Pool(processes=4) as pool:
        pool.map(create_time_resolutions, dfs_split)

with timer('pool datetime processing threads:'):
    with ThreadPool(processes=4) as pool:
        pool.map(create_time_resolutions, dfs_split)
```

Pool datetime processing: done in 9.120 seconds.

Pool datetime processing threads: done in 19.125 seconds.

```
with timer('joblib parallel datetime processing:'):
    Parallel(n_jobs=4)(delayed(create_time_resolutions)(i)  for  i
in dfs_split)

with timer('joblib parallel datetime processing threads:'):
      Parallel(n_jobs=4,prefer='threads')
```

Joblib parallel datetime processing: done in 16.912 seconds.

Joblib parallel datetime processing threads: done in 19.213 seconds.

Final comparison can be seen on Table 2.

Table 2
Data datetime processing performance comparison.

Processing approach	***Result***
Regular (sequential) processing	19.341 seconds
Pool parallel processing	**9.120 seconds**
Pool parallel processing threads	19.125 seconds
Joblib parallel processing	**16.912 seconds**
Joblib parallel processing threads	19.213 seconds

Therefore, parallel processing is typically faster than sequential processing. However, it's worth noting an interesting observation here: when using threads as the multiprocessing backend, parallel processing can be noticeably slower. In some cases, it might even be slower than sequential processing. In theory, both joblib and pool should yield similar results. It appears that kernel variability plays a significant role. For example, in one run, multiprocessing was much quicker (9 seconds vs. 19 seconds), while in the next run they showed almost identical results (19 seconds vs. 19 seconds).

Now, let's delve into parallel feature engineering. In this scenario, we'll be creating several groupings that will act as inputs for the pandas.groupby function. The exciting part is that these groupings will be processed concurrently, demonstrating the power of parallel processing.

```
  with timer('grouping features:'):
      for i in groupings:
    dfs_proc.append(create_grouped_df(df_time,           group_by=i,
columns_set=market_cols_agg_num))

  with timer('grouping features shifted:'):
      for i in groupings:
        dfs_proc_shift.append(create_grouped_df_shifted(df_time,
group_by=i, columns_set=market_cols_agg_num))
```

Grouping features: done in 1.721 seconds.

Grouping features shifted: done in 1.981 seconds.

For multiprocessing we depict the example of code for pool grouping features parallel. For other 3 methods the code is identical, only with different parameters.

```
  with timer('pool grouping features parallel:'):
      with Pool(processes=4) as pool:
          pool.starmap(#datasets go here as params)
```

Pool grouping features parallel: done in 2.820 seconds. Pool grouping features parallel threads: done in 1.101 seconds. Pool grouping features shifted parallel: done in 2.521 seconds. Pool grouping features shifted parallel threads: done in 1.226 seconds. Joblib grouping features parallel threads: done in 1.009 seconds. Joblib grouping features shifted parallel threads: done in 1.182 seconds (Table 3).

The disparities between parallel and sequential processing seem less pronounced for these groupings. This could be attributed to the inherent efficiency of the grouping operation, making the performance improvements less evident. It's advisable to evaluate the performance of the parallel version before incorporating it into parallel processing methods. An intriguing observation is that, in this context, the use of threads as the backend appears to be notably faster for multiprocessing.

Table 3
Parallel feature engineering performance comparison

Feature engineerings processing approach	***Result***
Grouping features (sequential)	1.721 seconds
Grouping features shifted (sequential)	1.981 seconds
Pool grouping features parallel	2.820 seconds
Pool grouping features parallel threads	**1.101 seconds**
Pool grouping features shifted parallel	2.521 seconds
Pool grouping features shifted parallel threads	**1.226 seconds**
Joblib grouping features parallel threads	**1.009 seconds**
Joblib grouping features shifted parallel threads	**1.182 seconds**

Also, we can notice the advantage of ThreadPool over Pool, which comes from their nature:

- *Lower Overhead*: The key distinction between ThreadPool and Pool is the way they manage parallel execution. Pool uses separate processes, while ThreadPool uses threads within a single process. Creating and managing processes is generally more resource-intensive and time-consuming than threads. Threads have lower overhead, as they share memory space and other resources, making them quicker to start and manage.
- *Shared Memory*: Threads within the same process share the same memory space, which can lead to more efficient data sharing and communication. In contrast, processes have their own memory space and need to use inter-process communication mechanisms to share data. This shared memory in ThreadPool can be advantageous for tasks that require a lot of data sharing.
- *GIL (Global Interpreter Lock)*: In the context of Python, the Global Interpreter Lock (GIL) is a mutex that allows only one thread to execute in the Python interpreter at a time. This means that even in a multi-threaded environment, only one thread can execute Python bytecode at a time. While this might seem like a limitation, it's not as significant when you're dealing with tasks that are I/O-bound (e.g., file I/O, network requests), as the GIL is often released during I/O operations. In such cases, using threads with ThreadPool can be more efficient.
- *Task Nature*: The choice between ThreadPool and Pool can also depend on the nature of the tasks you're parallelizing. If you have CPU-bound tasks (tasks that require a lot of computation), then using separate processes (Pool) may provide better performance because of true parallelism on multi-core CPUs. On the other hand, for I/O-bound tasks or tasks that are heavily reliant on efficient data sharing, threads (ThreadPool) might be faster due to the reasons mentioned above.

Overall, we can sum up that parallel processing can benefit the training speed in both cases, but the bump in the speed depends on the input size (the bigger size - the better speed improvement). Generally speaking, we can see 2x speed boost with allocating 4x resources.

5. Distributed machine learning

Distributed training is the practice of training machine learning algorithms across multiple machines, with the primary objective of enhancing scalability. In essence, it enables the handling of larger datasets by integrating more machines into the training infrastructure [9].

The benefits of distributed training are evident in the increased availability of CPUs and greater bandwidth, facilitating the processing of substantial volumes of data. Nonetheless, harnessing this added capacity effectively remains a significant challenge.

Distributed training becomes a necessity under the following circumstances:

1. *Time-Intensive Training*: When the training process consumes a substantial amount of time, especially in scenarios with vast datasets such as large text corpora, extensive image datasets, video data for autonomous vehicles, medical imaging data, or satellite imagery.
2. *Storage Constraints*: When storing the data on a single machine is unfeasible due to its size.
3. *Data Localization*: When data accessibility is confined to specific locations, often due to security requirements or the immense size of datasets (common in large enterprises or astro data).
4. *RAM Constraints*: When the data must be entirely accommodated in RAM for preprocessing, making it necessary to employ distributed training for efficient handling [10].

Distributed training is a powerful approach but not without its complexities. Let's delve into the specifics of data parallelism, a technique focusing on the distribution of data across multiple machines. Note that we won't explore model parallelism in this context, as it deals with distributing machine learning models across several machines [11].

The process of data parallelism typically involves the following steps:

1. *Model Initialization*: Initiate the machine learning model on the primary server, which serves as the central point for model management.
2. *Model Distribution*: Workers, the machines responsible for computation, retrieve a copy of the model from the primary server. Each worker works on a subset of the data.
3. *Gradient Calculation*: Workers perform calculations on their respective batches of training data, computing the gradients of the loss function with respect to the model's parameters.
4. *Gradient Communication*: After calculating the gradients, workers transmit this information back to the primary server, which serves as a hub for collecting and processing these updates.
5. *Model Update*: The primary server receives the gradients from all workers and utilizes them to update the model weights, implementing changes based on the collected gradient information.
6. *Iterative Process*: The process repeats, with workers fetching the updated model, recalculating gradients, and sending them for further model refinement. This iterative cycle continues until the training process reaches the desired performance level [12].

These steps collectively constitute the data parallelism approach for distributed training, allowing for efficient model training on vast datasets with the distributed computational power of multiple machines.

Numerous methods and techniques are available to effectively manage gradient computations when dealing with distributed data across multiple machines.

Synchronous updates, in which the main server waits for each worker to complete their gradient computations before executing a model update and redistributing it to all workers for the next iteration, can be a somewhat slow process. This is primarily due to the process being bottlenecked by the slowest worker's progress. To address this issue, several approaches can be employed [13]:

1. *Backup Workers*: This strategy involves having backup workers available. In this setup, the main server ceases to wait as soon as a certain number, N, of workers have completed their assigned tasks. While this approach accelerates the training process by reducing the wait time for slower workers, it comes at the cost of increased gradient variance. Smaller batch sizes result from this approach, potentially affecting the model's convergence speed. Nevertheless, the net effect is a more efficient training process due to the faster execution of additional iterations compared to waiting for slower workers.
2. *Flexible Rapid Reassignment (FlexRR)*: FlexRR is another technique that can be employed to mitigate the slowdown caused by a worker lagging behind. In this approach, a slower worker can delegate a portion of its tasks to faster workers. This redistribution of workload allows the slower worker to catch up with the overall progress of the other workers, ultimately reducing the training time.

These strategies are aimed at addressing the inherent challenges of synchronous updates in distributed training, improving overall training efficiency, and mitigating delays caused by slower participants in the process [14].

Asynchronous updates operate on a different principle. In this approach, the main server updates and serves the model to all workers each time a worker delivers a gradient update. However, this method can introduce race conditions because updates are computed on a worker using a model state that may

already be obsolete. The feasibility of asynchronous updates depends on the extent to which the weights updated by concurrent workers are distinct.

Fortunately, this approach proves effective in scenarios involving very large models, where only a small fraction of the model's weights are updated with each gradient update. In such cases, the convergence achieved with asynchronous updates is comparable to that of synchronous updates. What sets asynchronous updates apart is their higher computing efficiency, as they allow for concurrent updates from workers, optimizing the use of computational resources [15].

While synchronous updates might initially seem sluggish, their speed can be significantly improved with some additional engineering efforts, enabling them to match the efficiency of asynchronous updates. In contrast, asynchronous updates are simpler to implement but hinge on the sparsity of weight gradients.

Successful distributed training revolves around meticulous parameter tuning. Once you've settled on your model update strategy, fine-tuning the worker's batch size and the learning rate is of paramount importance. This process differs from the gradient descent algorithm on a single machine. In the case of synchronous updates, the model remains unaltered until all batches are collected from the workers. This effectively creates a virtual batch size (B) calculated as B = N x b, where N represents the number of workers and b signifies the batch size on a single machine. It's crucial for b to be sufficiently large to compute substantial gradients, and N can also be substantial, especially when tackling large-scale problems.

This reduces the number of steps required in an epoch without significantly accelerating the loss reduction process. To counteract this, you can use a higher learning rate, but this might come at the expense of convergence stability. Thus, meticulous tuning becomes crucial to fully harness the advantages of parallelization.

In the context of selecting a gradient update strategy, the next pivotal step is the choice of a framework to facilitate the integration of parallelization techniques within your machine learning algorithm. For the purpose of this discussion, we will focus on the utilization of the Horovod framework.

Horovod stands as a distributed deep learning framework that is compatible with popular libraries such as PyTorch and Keras. It boasts adaptability to various infrastructure backends, including Kubernetes, Ray, and Spark.

Our analysis will revolve around an illustrative example drawn from the Horovod repository, demonstrating MNIST training.

In the Horovod framework, several essential functions should be employed to enable distributed model training:

- `hvd.DistributedOptimizer`: This distributed optimizer encapsulates a Torch optimizer, offering a synchronized method for gathering and subsequently reducing gradients. Ultimately, it facilitates model updates.
- `hvd.broadcast_parameters`: This function serves to broadcast parameters (including state_dict, named_parameters, and parameters) to ensure uniform model updates across worker processes.
- `hvd.broadcast_optimizer_state`: It extends the capability to broadcast the optimizer itself to the nodes, guaranteeing that gradient updates are computed with the correct learning rate.
- `hvd.allreduce`: This function is integral for performing averaging or summation across all processes running on the worker nodes. It plays a critical role in the DistributedOptimizer and should be invoked whenever a value necessitates computation on worker processes and subsequent reduction on the primary server.
- `hvd.rank`: This function provides a unique identifier for the process running on the worker, facilitating differentiation between various worker processes.
- `hvd.size`: It offers insight into the total number of Horovod processes distributed across all worker nodes.

A crucial aspect of the data distribution process is managed by the PyTorch DistributedSampler:

- `torch.utils.data.distributed.DistributedSampler`: This component undertakes the task of sampling a subset of the data, specifically associated with the process rank.

With these tools at our disposal, we are well-equipped to construct a distributed training workflow for a PyTorch script. Furthermore, we may explore other distributed training approaches tailored to specific machine learning frameworks, such as PyTorch's distributed data parallel training with nn.parallel.DistributedDataParallel or Tensorflow/Keras' tf.distribute framework.

Horovod offers synthetic benchmarks for TensorFlow v1, TensorFlow v2, and PyTorch, facilitating performance and scalability evaluation in your specific environment. These benchmarks serve as a valuable tool for assessing Horovod's capabilities and diagnosing performance issues. It's recommended to run synthetic benchmarks initially to isolate potential problems not related to the training script itself.

6. Conclusion

In the world of machine learning, distributed training stands as a powerful approach for enhancing scalability, allowing machine learning algorithms to effectively process massive datasets across multiple machines. The benefits of distributed training are particularly evident in scenarios involving extensive time-consuming training processes, storage constraints, data localization requirements, and RAM limitations. With the increased availability of computational resources and greater bandwidth, distributed training has emerged as a crucial solution to address these challenges.

Data parallelism facilitates the collaborative training of machine learning models by distributing data across multiple machines. This approach includes various key steps such as model initialization, model distribution, gradient calculation, gradient communication, and model updates. Data parallelism enables efficient model training on extensive datasets, harnessing the collective computational power of multiple machines. Our experiments have consistently shown that parallel processing significantly enhances the speed and efficiency of data pipelines, datetime feature creation, and feature engineering. This speed improvement is most pronounced when dealing with computationally intensive operations. Our results have demonstrated that parallelism offers a promising avenue for addressing the scalability and efficiency challenges in modern data-driven applications.

While distributed training offers immense benefits, it also brings inherent complexities. Strategies for handling synchronous and asynchronous updates, such as the use of backup workers and Flexible Rapid Reassignment (FlexRR), play a critical role in mitigating delays caused by slower workers and optimizing training efficiency. Successful distributed training requires meticulous parameter tuning, involving adjustments to the worker's batch size and learning rate, to fully harness the advantages of parallelization. Parallel processing can benefit the training speed in both cases, but the bump in the speed depends on the input size (the bigger size - the better speed improvement). Generally speaking, we can see 2x speed boost with allocating 4x resources.

Choosing a gradient update strategy is a pivotal step in the distributed training process. The selection of an appropriate framework can significantly impact the integration of parallelization techniques into your machine learning algorithm. The Horovod framework, compatible with popular libraries like PyTorch and Keras, offers essential functions for distributed model training, such as DistributedOptimizer, broadcast_parameters, broadcast_optimizer_state, allreduce, rank, and size. These functions streamline the process of data distribution, enabling a seamless distributed training workflow for machine learning tasks. This technique can assist to spread (parallelize) the learning process across multiple machines. These techniques are essential for achieving rapid data processing, model training, and feature engineering, enabling organizations to develop more effective anomaly detection solutions and machine learning models in shorter timeframes.

7. References

[1] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. ACM Comput. Surv. 52, 4, Article 65 (July 2020), 43 pages. https://doi.org/10.1145/3320060

[2] Aach, M., Inanc, E., Sarma, R. et al. Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks. J Big Data 10, 96 (2023). https://doi.org/10.1186/s40537-023-00765-w
[3] Esmaeilzadeh, Soheil & Salajegheh, Negin & Ziai, Amir & Boote, Jeff. (2022). Abuse and Fraud Detection in Streaming Services Using Heuristic-Aware Machine Learning.
[4] Hegde, Vishakh and Sheema Usmani. “Parallel and Distributed Deep Learning.” (2016).
[5] William Fedus, Barret Zoph, & Noam Shazeer. (2022). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity.
[6] Low, Yucheng, et al. "Graphlab: A new framework for parallel machine learning." arXiv preprint arXiv:1408.2041 (2014).
[7] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, & Rafal Jozefowicz. (2017). Revisiting Distributed Synchronous SGD.
[8] Wang, Hao, Di Niu, and Baochun Li. "Distributed machine learning with a serverless architecture." IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, 2019.
[9] Li, Mu. "Scaling distributed machine learning with system and algorithm co-design." Santa Clara, CA, USA: Intel (2017).
[10] Peteiro-Barral, Diego, and Bertha Guijarro-Berdiñas. "A survey of methods for distributed machine learning." Progress in Artificial Intelligence 2 (2013): 1-11.
[11] Konečný, Jakub, et al. "Federated optimization: Distributed machine learning for on-device intelligence." arXiv preprint arXiv:1610.02527 (2016).
[12] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, & Jan S. Rellermeyer (2020). A Survey on Distributed Machine Learning. ACM Computing Surveys, 53(2), 1–33.
[13] Xing, Eric P., et al. "Strategies and principles of distributed machine learning on big data." Engineering 2.2 (2016): 179-195.
[14] Kraska, Tim, et al. "MLbase: A Distributed Machine-learning System." Cidr. Vol. 1. 2013.
[15] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems 25 (2012).
[16] Almasi, G., & Gottlieb, A. (1989). Highly Parallel Computing. Benjamin-Cummings Publishing Co., Inc..
[17] A. Farkas, G. Kertész and R. Lovas, "Parallel and Distributed Training of Deep Neural Networks: A brief overview," 2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES), Reykjavík, Iceland, 2020, pp. 165-170, doi: 10.1109/INES49302.2020.9147123.
[18] Chan, P., & Stolfo, S. (1993). Toward Parallel and Distributed Learning by Meta-Learning. In Proceedings of the 2nd International Conference on Knowledge Discovery in Databases (pp. 227–240). AAAI Press.
[19] Bekkerman, R., Bilenko, M., & Langford, J. (2011). Scaling up Machine Learning: Parallel and Distributed Approaches. In Proceedings of the 17th ACM SIGKDD International Conference Tutorials. Association for Computing Machinery.
[20] Lazarevic, A., Obradovic, Z. Boosting Algorithms for Parallel and Distributed Learning. Distributed and Parallel Databases 11, 203–229 (2002). https://doi.org/10.1023/A:1013992203485