Elhmadany *et al. Journal of Big Data* (2024) 11:6
https://doi.org/10.1186/s40537-023-00871-9

Journal of Big Data

RESEARCH Open Access

Instance segmentation on distributed deep learning big data cluster

Mohammed Elhmadany[1], Islam Elmadah[1] and Hossam E. Abdelmunim[1*]

*Correspondence: hossameldin.hassan@eng.asu.edu.eg

[1] Computer and Systems Engineering, Faculty of Engineering, Ain Shams University, 11517 Cairo, Egypt

Abstract

Distributed deep learning is a promising approach for training and deploying large and complex deep learning models. This paper presents a comprehensive workflow for deploying and optimizing the YOLACT instance segmentation model as on big data clusters. OpenVINO, a toolkit known for its high-speed data processing and ability to optimize deep learning models for deployment on a variety of devices, was used to optimize the YOLACT model. The model is then run on a big data cluster using BigDL, a distributed deep learning library for Apache Spark. BigDL provides a high-level programming interface for defining and training deep neural networks, making it suitable for large-scale deep learning applications. In distributed deep learning, input data is divided and distributed across multiple machines for parallel processing. This approach offers several advantages, including the ability to handle very large data that can be stored in a distributed manner, scalability to decrease processing time by increasing the number of workers, and fault tolerance. The proposed workflow was evaluated on virtual machines and Azure Databricks, a cloud-based platform for big data analytics. The results indicated that the workflow can scale to large datasets and deliver high performance on Azure Databricks. This study explores the benefits and challenges of using distributed deep learning on big data clusters for instance segmentation. Popular distributed deep learning frameworks are discussed, and BigDL is chosen. Overall, this study highlights the practicality of distributed deep learning for deploying and scaling sophisticated deep learning models on big data clusters.

Keywords: Distributed deep learning, Big data cluster, BigDl, Spark, Instance segmentation, YOLACT, OpenVINO, ONNX, Azure databricks

Introduction

In our modern world, a massive amount of data is generated daily. This requires a unique approach to process it effectively. Deep Neural Networks (DNNs), despite their state-of-the-art results on tasks like image classification, object detection, natural language processing, and machine translation, face a significant challenge: they require large amounts of data to train and achieve high accuracy. This is due to the large number of parameters in DNNs that need to be learned from the data. As shown in Fig. 1 the model learns from more data, it becomes better at understanding complex patterns and relationships within the information [1]. Distributed Deep Learning (DDL) addresses this challenge by training and deploying DNNs on multiple machines, distributing the computational

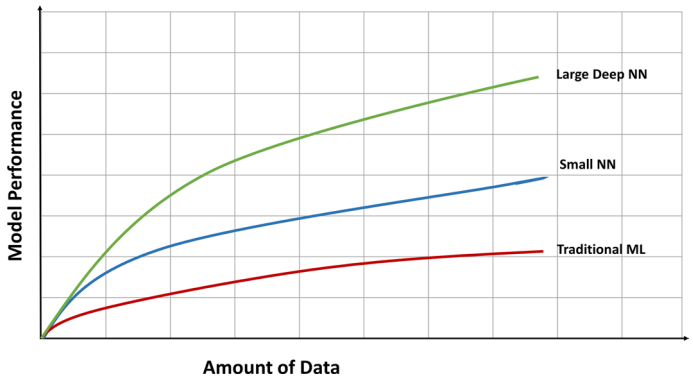

Fig. 1 The relationship between model performance and the size of the trained data is generally positive, with larger amounts of training data leading to improved performance in deep learning models [1]

workload and data across them for faster, more scalable, and cost-effective training and inference. DDL can handle terabytes of data that cannot be stored on a single machine by distributing the data and computational workload across multiple machines. Each machine stores a portion of the data and computes the gradients for that portion. The gradients are then exchanged between machines to update the model parameters. Large Language Models (LLMs) like Transformer, which are so large and complex that they cannot be stored or trained on a single machine, can be trained and deployed using DDL. For instance, GPT-3, a large language model based on the Transformer architecture, has 175 billion parameters and was trained on a dataset of 100 trillion words. Thus, DDL is a powerful technique for training LLMs and DNNs on large datasets.

DDL has recently gained a lot of attention because of its effectiveness in various applications. DDL is important for building and training large-scale deep-learning models. As the size and complexity of these models continue to grow, DDL is likely to become an increasingly important tool in the field of artificial intelligence. Deep Neural Networks can use Distributed Deep Learning to spread out their processing tasks over multiple Central Processing Units (CPUs) or Graphics Processing Units (GPUs). DDL can be implemented either on-premises or on the cloud, catering to an organization's specific needs and requirements. Moreover, the integration of DDL with cloud computing leverages the scalability and flexibility of cloud infrastructure for training and deploying DNNs. Cloud computing is a cost-effective method to use a variety of computer resources like virtual machines, storage, and networking whenever you need them. Combining this with DDL provides a powerful way to handle the needs of large deep-learning models [2, 3]. These benefits of employing DDL will be further explored in the Motivation section. There are several cloud-based platforms and services, such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and IBM Cloud, that offer distributed deep learning capabilities. These platforms provide a variety of tools and services that enable users to train deep neural networks on a large scale. This includes distributed computing infrastructure, pre-configured deep learning frameworks, and automated machine learning pipelines [4]. For example, using cloud-based DDL on Azure Databricks allows companies to avoid the hassle of creating and maintaining their own deep learning systems. This lets them focus on their main business goals and use the scalability and flexibility of cloud computing to train and deploy

Deep Neural Networks (DNNs). This research suggests that DDL on Azure Databricks is a promising method for creating and deploying high-performance DNNs on large datasets.

In this research, YOLACT, a complex real-time instance segmentation model, was chosen to test the effectiveness of DDL on Azure Databricks. Various techniques were used to optimize YOLACT, including converting the original PyTorch model to ONNX and OpenVINO, to make it lighter and decrease prediction time. The most efficient model was then deployed on a distributed big data cluster.

This paper is divided into nine sections, each focusing on a different aspect of Distributed Deep Learning (DDL). Section A survey of distributed deep learning frameworks provides an overview of the main frameworks used in DDL. Section Motivation and addressing challenges in the utilization of distributed deep learning on big data clusters explores the motivations behind the adoption of DDL, with a specific focus on its implementation within big data clusters. This section also examines the advantages and obstacles associated with the utilization of DDL in such an environment. Section Bigdl end-to-end, distributed AI on big data cluster offers a practical guide to implementing BigDL in a distributed AI system, shedding light on its deployment in real-world scenarios. Section Instance segmentation deep learning model introduces instance segmentation deep learning models, explaining their importance and various applications. Sections Optimizing the YOLACT model for improved inference throughput and Comparative analysis of Speed and performance across various YOLACT models are dedicated to the optimization of the YOLACT model for inference, providing a detailed evaluation of the speed and performance of different YOLACT models. The implementation of YOLACT OpenVINO on a big data cluster is demonstrated in Sections YOLACT OpenVINO distributed inference on big data clusters using virtual machines and YOLACT OpenVINO distributed inference on big data clusters on azure databricks: experimental results. Section 7 presents experimental results from running BigDL on Spark clusters hosted on virtual machines, offering a comprehensive demonstration of this implementation in both standalone and YARN cluster modes, along with an overview of the results and insights obtained from these experiments. Section YOLACT OpenVINO distributed inference on big data clusters on azure databricks: experimental results focuses on the performance of YOLACT OpenVINO on a big data cluster using Azure Databricks, providing insights into its efficiency. Finally, Section Conclusions and future work concludes the paper by summarizing the insights gained and suggesting potential future research directions to further advance the field of DDL.

A survey of distributed deep learning frameworks

Distributed deep learning is a powerful approach to training and deploying large-scale deep learning models on distributed clusters of CPUs or GPUs. This allows for the training of much larger and more complex models than would be possible on a single machine, and can also significantly reduce training time. GPUs are commonly used in deep learning because they are highly efficient at performing the matrix multiplication operations fundamental to deep learning algorithms. DDL frameworks provide the necessary tools and interfaces for developers to efficiently build, train, and deploy their deep learning models in a distributed environment. These frameworks handle the challenges

of coordinating the training process across multiple machines, such as data parallelism, model parallelism, and distributed communication.

In this study, BigDL [5] was selected as the distributed deep learning framework for CPU clusters, providing a cost-effective and scalable solution for running high-scale deep learning models on large datasets. BigDL, an open-source distributed deep learning library developed by Intel and built on Apache Spark, enables organizations to leverage their existing CPU clusters to run deep learning workloads without the need for expensive GPUs. It also simplifies the data loading from big datasets stored in the Hadoop cluster. BigDL achieves significant performance gains on Intel CPUs by leveraging Intel's Math Kernel Library (MKL). With its distributed computing capabilities, BigDL can accelerate deep learning computations and distribute workloads across multiple CPUs in a cluster, delivering high performance and scalability for deep learning inference. BigDL is a valuable solution for industries where Intel CPUs are widely used. The following section will discuss BigDL in more detail, including its implementation in different environments, the techniques used in DDL, and the challenges faced during implementation.

In order to expand our understanding of different distributed deep learning frameworks, let's explore another widely-used framework known as Horovod. By examining different DDL frameworks, this exploration helps in gaining insights into the available options and essential considerations for making an informed choice. Horovod is a distributed deep-learning framework that was originally developed by Uber Technologies and is now maintained by the LF AI Foundation. It is a strong contender in the DDL field because of its design and capabilities. Horovod provides a simple and easy-to-use interface for distributed training of deep learning models. It supports multiple deep learning frameworks such as TensorFlow, PyTorch, and MXNet, and it can be used with various types of deep learning models such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformer Networks. Horovod is a powerful and efficient distributed deep-learning framework that can help researchers and data scientists scale their models to large clusters of GPUs and CPUs. Its use of Ring-Allreduce and MPI helps to reduce communication overhead and improve the scalability of distributed training, making it a popular choice for large-scale deep-learning workloads [6].

The main difference between BigDL and Horovod is framework architecture: BigDL is built on top of Apache Spark, which is a distributed computing framework for large-scale data processing. In contrast, Horovod is built on top of MPI (Message Passing Interface) [7], which is a standard for parallel computing on distributed systems, BigDL may be more accessible to users who are already familiar with Apache Spark and distributed computing, while Horovod may be more appealing to users who are primarily focused on GPU acceleration and have experience with MPI. In addition, BigDL's All-Reduce algorithm has similar performance characteristics to Horovod's Ring-AllReduce algorithm for aggregating gradients from multiple nodes in a distributed system during the back-propagation step of deep learning training [8]. In short, both BigDL and Horovod are robust distributed deep learning frameworks, each with its unique strengths. BigDL is a good choice for projects that require both large-scale data processing and analytics alongside deep learning. BigDL's design allows it to run directly on top of existing Spark or Hadoop clusters, which can be a significant advantage for projects that need to leverage these resources. BigDL also provides a unified data analytics and AI

pipeline, which can simplify the development process and reduce the complexity of the system. Horovod is a good choice for projects that need to train and deploy large-scale deep-learning models. It is highly efficient and can scale to thousands of GPUs. Horovod supports a wide range of deep learning frameworks, including TensorFlow, PyTorch, and MXNet, but it does not currently support running OpenVINO models. BigDL was also chosen for this study because an optimized OpenVINO model achieves the best inference throughput performance, and to explore the advantages of deploying deep learning models on big data Spark clusters. Another approach extended to Horovod standalone is a Horovod on Spark that combines the power of Horovod on CPUs or GPU clusters with the distributed computing capabilities of Apache Spark. The main advantage of Horovod on Spark over Horovod is its ability to scale deep learning model training across a large cluster of computers. This can be especially useful for training large models on massive datasets. However, Horovod on Spark also adds some complexity to the training process [9]. In terms of fault tolerance if compared Horovod on Spark with BigDL, BigDL has a more robust and integrated approach than Horovod on Spark, as it uses Spark's fault tolerance mechanisms directly. However, Horovod on Spark can be used with additional tools to add fault tolerance, such as job schedulers or check-pointing when the fault tolerance feature is significant for rescheduling the failed task on a different worker node instead of restarting the entire job from the beginning [10].

Ray also provides fault tolerance mechanisms to handle worker failures during model training. When a worker fails, Ray is a distributed computing framework that provides a set of high-level APIs for building scalable and fault-tolerant applications that can automatically reschedule the failed task on a different worker node, and it can also use check-pointing to save the current state of your model and optimizer to disk periodically during training, which can be used to recover from failures. Ray can be used with several popular deep learning frameworks, including TensorFlow, PyTorch, and Keras, to scale model training across a cluster of machines [11].

In addition, TensorFlow offers various distributed training strategies to cater to different hardware configurations and accelerate machine learning workloads. OneDeviceStrategy is ideal for single-device setups, placing all variables and computations on a single device. For multi-GPU synchronous training on a single device, MirroredStrategy creates model replicas on each GPU, mirroring variables across all replicas. MultiWorkerMirroredStrategy extends this synchronous approach to multiple devices, each with one or more GPUs. TPUStrategy leverages Google's specialized Tensor Processing Units (TPUs) to significantly boost training speed compared to traditional CPUs and GPUs, although it currently exclusively supports TensorFlow. TPUStrategy aligns with the distributed training methodology of MirroredStrategy, enabling synchronous distributed training across multiple TPU cores [12].

Also, PyTorch Distributed is an integral library in PyTorch designed for building distributed training applications, allowing for efficient model training across multiple GPUs or distributed machines. Its key components, including Distributed Data-Parallel Training (DDP), RPC-Based Distributed Model Parallel, and the Collective Communication Library (c10d), work together to synchronize model training, enable parallel processing, and facilitate seamless communication between processes, ultimately reducing training time and enhancing performance for large-scale deep learning models [13–15].

In conclusion, the best distributed deep learning framework depends on your specific needs. If you're already using Apache Spark, BigDL is a good choice. If you want a user-friendly library that supports multiple deep-learning frameworks, Horovod is a good choice. If you need a distributed computing framework for machine learning applications, including distributed deep learning, Ray is a good choice.

Motivation and addressing challenges in the utilization of distributed deep learning on big data clusters

Distributed deep learning is motivated by the increasing volume and complexity of data in real-world applications, and the need to train and deploy large and complex deep learning models. DDL efficiently processes and learns from this data by distributing computational tasks across multiple machines [16]. DDL is better than single-machine deep learning, DDL offers several advantages over traditional single-machine deep learning, especially when deploying large and complex deep learning models on big data clusters these advantages include **Scalability**, DDL can scale to train and deploy models on very large datasets and more complex models, which is essential for many deep learning applications;**Fault tolerance**, Big data clusters are typically designed to be fault-tolerant, meaning that they can continue to operate even if some of the nodes fail. This makes them a good choice for deploying DDL applications, as it can help to reduce the risk of training failures [17]; **Speeding Up Training Time**, By distributing the computational workload across multiple machines, DDL can significantly reduce the time required to train models and inference of deep learning models, which can be critical for time-sensitive applications; **Cost-effectiveness**, DDL can help to reduce the cost of training and deploying deep learning models by using distributed computing resources. DDL reduces the cost of training and deploying deep learning models by distributing the workload across commodity machines in a cluster, which improves utilization of available resources; and **Resource sharing**, By sharing resources among multiple applications, organizations can make better use of their existing resources and avoid the need for additional hardware investments [18]. DDL on Big Data clusters provides a scalable, efficient, and robust solution for training deep learning models on large datasets As the demand for deep learning continues to grow, DDL is expected to become even more widely adopted in the future. DDL offers significant benefits for training machine and deep learning models, including faster training times, improved accuracy, and increased scalability. DDL works by distributing computational tasks across multiple machines in a big data cluster. This allows DDL applications to scale to very large datasets and complex models, such as VGG networks [19] or Inception Resnet network [20]. DLL significantly reduces the training time and improves the accuracy of very large models on very large datasets. While GPUs are preferred for training due to their high performance, CPUs are sufficient and more attractive for data preprocessing and inference, which are less resource-intensive. There is a growing interest in developing CPU-optimized deep learning frameworks and algorithms, as CPUs are more widely available and less expensive.

Moreover, techniques like hyperparameter tuning and neural architecture search can substantially impact model accuracy and complexity, but training in neural architecture search can be time-consuming. taking many computing hours [21]. By applying distributed hyperparameter tuning (DHPT) provides several advantages in optimizing

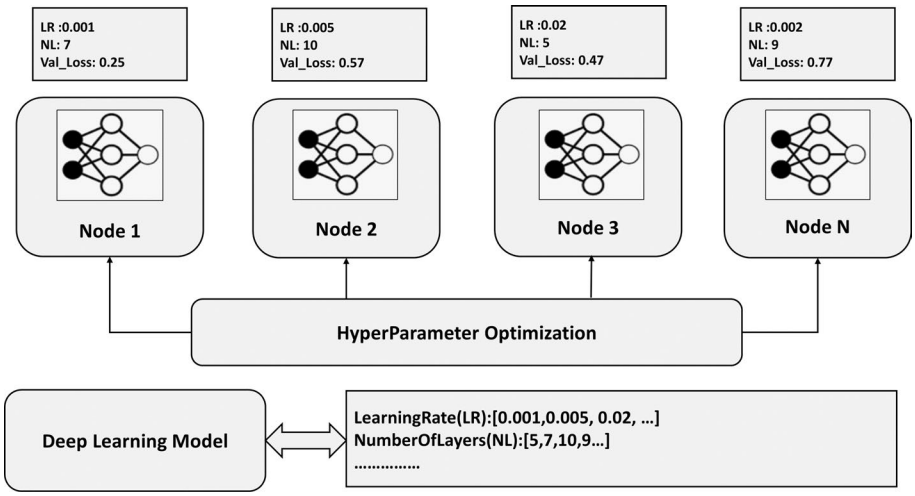

Fig. 2 Hyper-parameter tuning in a distributed deep learning setting, where each node can evaluate a subset of the hyper-parameter combinations, and the results are aggregated to select the best set of hyper-parameters, can offer several benefits for accelerating the search for the best set of hyper-parameters in large-scale machine learning models. By distributing the search among multiple nodes, it is possible to reduce the search time and explore a wider range of hyperparameter combinations

deep learning models. One of the key benefits is faster convergence. By evaluating multiple hyperparameter combinations in parallel, DHPT accelerates the training process, leading to quicker model convergence and more efficient utilization of computational resources. DHPT proves particularly beneficial for large deep-learning models with numerous hyperparameters. It optimizes these models more effectively and efficiently, making it a powerful tool in the field of deep learning. as shown in Fig. 2

Distributed deep learning takes advantage of both data and model parallelism. When the dataset or model is too large to fit on a single machine, distributing them across multiple machines becomes a necessity. In data parallelism, the training data is split and distributed across several machines, with each machine training the same model. On the other hand, model parallelism involves dividing the model itself across multiple machines, where each machine is responsible for training different parts of the model. Overall, the motivation behind DDL lies in its ability to handle large datasets and complex models efficiently, making it a key player in the future of deep learning applications. in the case of **Data Parallelism** where data is distributed across multiple machines. This can be used to speed up training or in cases where the amount of data is too large to fit on a single machine by assigning each part of the data set to one GPU/CPU. each node independently computes the gradients for its assigned part of the data and sends the gradients to a master node. The master node aggregates the gradients from all the GPUs/CPUs and updates the model parameters accordingly. as shown in Fig. 3

Model parallelism, on the other hand, allows the model to be split across multiple machines if that model is too large or complex that would not fit on a single machine so that a single layer can be fit into the memory of a single node in the cluster whereas forward and backward propagation involves the communication between the outputs from one node to another in a serial fashion [22] The division of the model in model parallelism requires careful consideration to ensure that the computational load is evenly

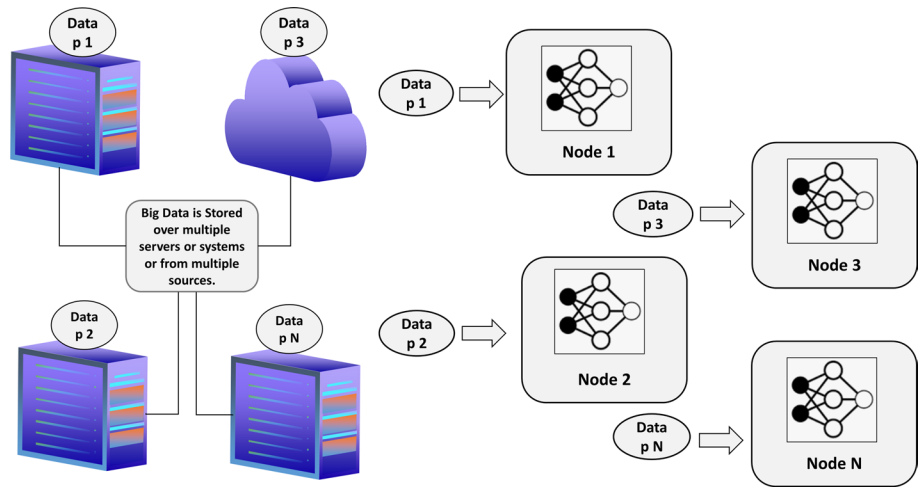

Fig. 3 In Data Parallelism, the training data may be stored in distributed storage systems such as HDFS (Hadoop Distributed File System) or other cloud-based storage solutions. By distributing the data across multiple nodes, each node can work on a subset of the data in parallel, allowing for faster training times

distributed among the different machines or devices during forward propagation, each machine or device computes the forward pass for its assigned segment or layer of the model by processing the input data and computing intermediate results or activations. These intermediate results or activations are then passed to the next machine or device for further computation. This process continues until the final output of the model is obtained.

Similarly, during backward propagation, each machine or device computes the gradients of the loss function with respect to the locally assigned segment or layer of the model. This involves computing the gradients based on the local activations and local model parameters. The computed gradients are then passed to the previous machine or device, which computes the gradients for the previous segment or layer of the model. This process continues until the gradients for all segments or layers of the model are computed as shown in Fig. 4 Proper load balancing and coordination between machines or devices are critical in model parallelism to ensure that the computational workload is distributed evenly and that the intermediate results and gradients are passed accurately between machines or devices. Careful consideration of the model architecture and the distribution of computational tasks among different machines or devices is essential to achieve efficient and effective model parallelism in distributed deep learning. as shown in Fig. 4

However, model parallelism can help reduce the memory requirements and improve the training time of very large models. But it does have some serious technical limitations concerning model splitting. For instance, there can be difficulty in load balancing when dividing a large model into smaller sub-models, and it can be challenging to balance the workload evenly across all the devices or processors. This can lead to some sub-models being overburdened while others are underutilized, resulting in sub-optimal performance It is difficult for the convolution neural network algorithm to select

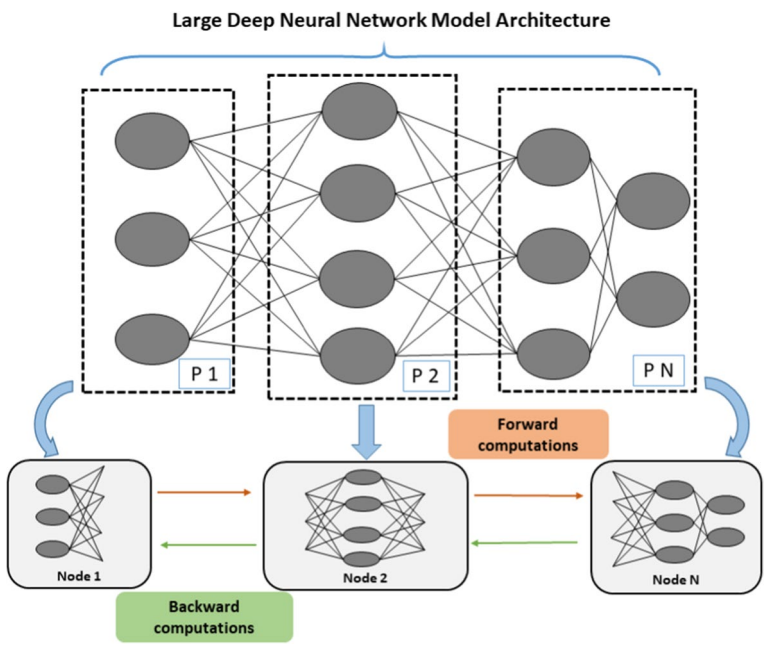

Fig. 4 In Model parallelism, different parts of the large model are assigned to different nodes or machines. The intermediate results or activations need to be exchanged between machines or devices during forward propagation, and the gradients need to be passed backward between machines or devices during backward propagation

the appropriate optimization method and the optimal time also challenges required in implementing especially when implemented in a heterogeneous system, which means it cannot leverage existing the advantages of the heterogeneous system's computing resources. Furthermore, when the hardware condition changes, the training algorithm cannot dynamically adapt to the computing resources, resulting in low training efficiency.

Data parallelism and Model parallelism can both be categorized as either synchronous or asynchronous parallelism, depending on how the computation is organized and coordinated across the parallel workers [23]. In **synchronous parallelism** training, multiple machines work together on the same model at the same time. They all update their model parameters simultaneously after each iteration, using the same information requires the use of a synchronization barrier to force all nodes to update model parameters. This method requires strong communication and coordination between the machines one limitation of this parallel method is that the faster nodes wait for the other slower nodes for each iteration, which greatly affects the model training speed so the converge time for synchronous parallel training can also be longer than asynchronous parallel training, especially when the dataset is large and the number of worker nodes is high. as shown in Fig. 5 in Synchronous parallel training, the master node is often used as a parameter server. The master node is responsible for aggregating the gradients from each worker node, computing the overall gradient, and updating the model parameters. The worker nodes are responsible for computing the gradients for their portion of

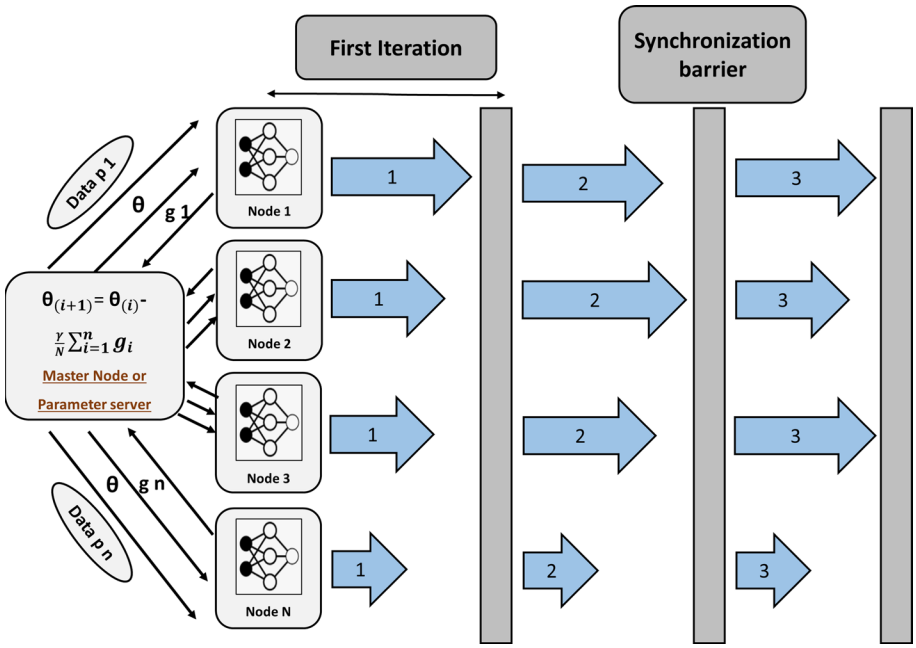

Fig. 5 Synchronous parallelism is a widely used method in distributed deep learning using stochastic gradient descent (SGD) optimization. In this method, multiple worker nodes compute the gradients on different subsets of the data and send them to a parameter server. The parameter server aggregates the gradients and updates the model weights synchronously, which means all workers update their weights simultaneously. This process is repeated for a number of epochs until the model converges

the data and sending them to the master node for aggregation. This approach helps to ensure that all worker nodes are updating the same model parameters at each iteration, leading to better model convergence. In Fig. 5 describes the steps involved in the synchronous parallel training process: (1) The master node initializes the model parameters and distributes copies of the model to all worker nodes. (2) Each worker node receives a batch of training data and calculates the gradients for that batch using the current model parameters. (3) The gradients from all worker nodes are aggregated and averaged at the master node to obtain a single set of gradients. (4) The master node updates the model parameters using the averaged gradients and distributes the updated model copies to all worker nodes. (5) Steps 2–4 are repeated for multiple epochs until the model converges. One of the benefits of using synchronous parallel training is **the synchronization barrier**, which ensures that all worker nodes update their model copies using the same set of gradients. This helps to keep the models synchronized and improves the quality of the trained model. Additionally, synchronous parallel training can reduce the training time as all worker nodes can work in parallel and share the computational load.

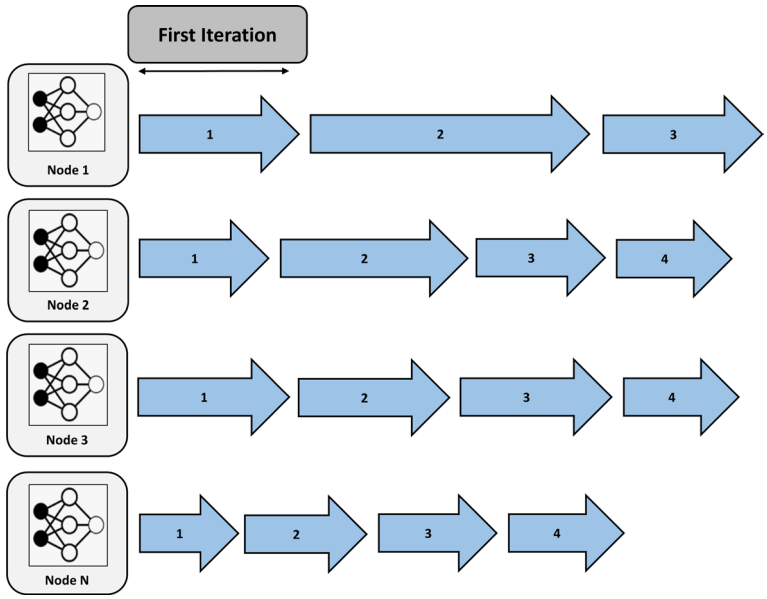

Fig. 6 Training a deep learning model using the asynchronous parallel method with stochastic gradient descent. The method involves multiple worker nodes independently training different subsets of the data and updating the model parameters asynchronously without coordination. The training process can suffer from slower convergence and increased variability due to the asynchronous updates. However, it can also make more efficient use of computational resources compared to synchronous parallel methods

Here is the pseudo-code for a synchronous update method in Algorithm 1:

Algorithm 1 Synchronous SGD with Parameter Server Pseudocode for Distributed Training

Require: Training data D, batch size B, number of workers N, learning rate γ
Initialize model parameters θ randomly
Distribute θ to all worker nodes
for $t = 1$ to maximum number of iterations **do**
  for $i = 1$ to N in parallel **do**
    Worker i samples a mini-batch of size B from D
    Worker i computes gradients g_i using mini-batch and current θ
    Worker i sends g_i to the parameter server
  end for
  Parameter server aggregates gradients from all workers:
  $\hat{g} = \frac{1}{N} \sum_{i=1}^{N} g_i$
  Parameter server updates θ:
  $\theta \leftarrow \theta - \frac{\gamma}{N} \hat{g}$
  Distribute updated θ to all worker nodes
end for

On the other hand, **Asynchronous parallel** does not need strong coordination between the machines, each node trains a portion of the model independently, this results in each machine working on a different part of the model simultaneously, leading to more efficient use of computational resources as shown in figure 6

This method in N nodes can get almost N times the speed up [24]. The asynchronous parallel method has several disadvantages compared to other parallel training methods, such as synchronous parallel training like variability in convergence the rate of

convergence can vary between machines, making it more difficult to determine when training is complete also the training process easy to falls into the local optimal solution, resulting in poor network training convergence Additionally, this method may suffer from the "stale gradients" problem, where the parameter server receives outdated gradient updates from some nodes, leading to slower convergence and less accurate results. In this algorithm, each worker node trains the model using a random subset of the training data. The gradients computed by each node are then used to update the global model parameters asynchronously. The learning rate determines the size of the update to the global model parameters. The algorithm continues for a fixed number of iterations or until the convergence criterion is met. Here is the pseudo-code for an Asynchronous update method in Algorithm 2:

Algorithm 2 Asynchronous Parallel SGD for Distributed Training

Require: Training data D, batch size B, number of workers N, learning rate γ
Initialize model parameters θ randomly
for $i = 1$ to maximum number of iterations **do**
    for $n = 1$ to N in parallel **do**
        D_n = random subset of D of size B
        g_n = compute gradients of D_n w.r.t θ
        $\theta_n = \theta - \gamma * g_n$
        update global model parameters using θ_n
    end for
end for

This algorithm follows the asynchronous parallel method and trains the model by updating the global model parameters θ using mini-batches of data computed by each worker node. Each worker computes the gradients of its mini-batch concerning the current model parameters θ, updates its local copy of the model parameters, and then updates the global model parameters by sending its updated parameters to the server. This process repeats until the maximum number of iterations is reached. Some challenges required in Implementing asynchronous parallel training that can be more complex compared to other parallel training methods it require the machines to exchange information periodically to ensure that the model parameters are consistent across all machines. This can be done either by exchanging the updated model parameters between the machines or by exchanging gradient information that can be used to update the model parameters. So, in practice, the asynchronous parallel method is not recommended to be used in model training [25]. Furthermore, achieving peak performance with big data requires building an efficient input pipeline that can deliver data to the next step before the previous step is finished, while also considering throughput, latency, ease of implementation, and maintenance. Big data has more powerful tools for reading and loading data from distributed environments, making it easier to achieve peak performance. So, when using TensorFlow or any deep learning framework with Apache Spark to be used in read the data from distributed file system frameworks like Hadoop, there is a need to do all **ETL processes** Extraction involves the process of extracting

the data from various sources, including databases, files, and web services. this data can be structured or unstructured and can come from a variety of sources. Transformation refers to data cleansing and manipulation to convert them into a proper format this may include cleaning the data, removing duplicates, and merging data from multiple sources. Loading The transformed data is loaded into a centralized repository or data warehouse, where it can be easily accessed and analyzed that would be intermediate storage then that data needs to be loaded into the deep learning cluster to do the actual training where the deep learning applications run directly on where the data are stored this makes the user maintain two different clusters one for ETL and one for distributed training of deep learning cluster. running and maintaining multiple separate clusters is tedious. But this is going to change from Apache Spark 3.0 supports working in both GPU and CPU clusters. Spark can now schedule GPU-accelerated ML and DL applications on Spark clusters with GPUs [26]. despite there being a limitation in fault tolerance because Spark 3.x implements a new execution mode called barrier execution mode which is different from the standard Map/Reduce mode. this kind of execution mode is useful for implementing distributed deep learning in Spark. In Map/Reduce, all tasks in a stage are independent of each other and they don't communicate with each other. If one of the tasks fails, only that task will be retried. But in Barrier execution mode, all tasks in a stage will be started together, and if one of the tasks fails the whole stage will be retried again all those tasks can communicate with each other [27] Where spark 2.x supports (Resilient Distributed Datasets) RDDs architecture that helps to achieve fault tolerance which is compatible to run on CPUs cluster only. While big data tasks are embarrassingly parallel and independent of each other, deep learning tasks need to coordinate with and depend on others. Therefore, it is highly inefficient to run these workloads on separate big data and deep learning systems (e.g., processing data on a Spark cluster, and then exporting the processed data to a separate deep learning cluster for training/inference) in terms of not only data extraction or transformation but also development, debugging, deployment, and operation productivity. There are efforts to address the previous challenges as the Connector approach where the data transfer between the CPUs and GPUs cluster is slow GPUs are typically connected to the CPU via a PCIe (Peripheral Component Interconnect Express)interface [28] which can limit the speed at which data can be transferred between the two. If the data transfer between the CPU and GPU is slow, then it may be slower to read data into a GPU cluster than a CPU cluster (e.g., TFX [29], CaffeOnSpark [30], TensorFlowOnSpark [31], etc.) which develops proper interfaces to connect different data processing and deep learning components using an integrated workflow (and possibly on a shared cluster). But this approach suffers from impedance mismatches [32] that arise from crossing boundaries between heterogeneous components in two clusters. BigDL can solve this challenge by providing a unified framework for distributed deep learning on both CPUs and GPUs when utilized with Apache Spark 3.x. As mentioned before, BigDL was chosen as the framework for assessing optimized deep-learning models on large-scale CPU clusters. This study presents the experimental results obtained under various conditions and discusses the challenges faced during implementation and the strategies employed to overcome them. The next section will discuss the BigDL architecture in more detail.

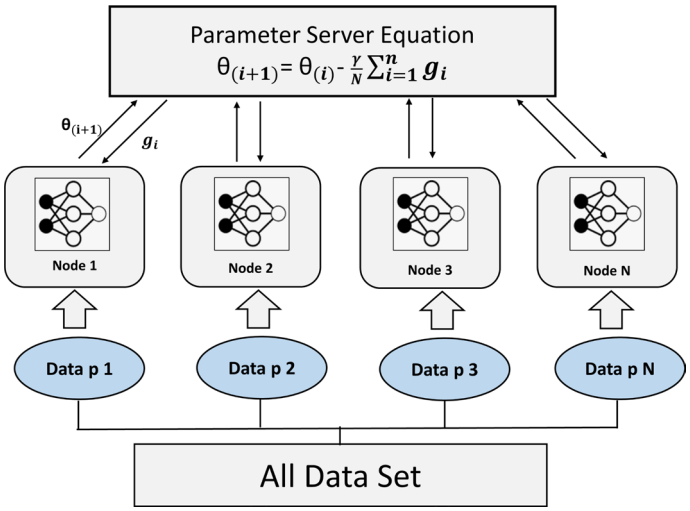

Fig. 7 Parameter servers with data parallelism is to coordinate the updates of model gradients across multiple worker nodes

Bigdl end-to-end, distributed AI on big data cluster

BigDL is a distributed deep-learning library for Apache Spark. It was developed by Intel and contributed to the open-source community. BigDL solves the challenges of distributed deep learning in Spark by using a coarse-grained synchronization technique. This means that BigDL only synchronizes the gradients between worker nodes at certain intervals, rather than after every step of the training process. This reduces communication overhead and improves the performance of distributed training jobs. BigDL uses a peer-to-peer AllReduce operation to communicate gradients between worker nodes. This operation allows all the worker nodes to exchange their gradients with each other in a single step. This is similar to how a parameter server architecture works but without the need for a central parameter server. On the other hand, traditional distributed deep learning frameworks like TensorFlow and PyTorch use a parameter server architecture. This means that there is a central server that stores the model parameters and communicates with the worker nodes that train the model. The worker nodes pull the latest parameters from the server and push their gradients back to the server as shown in Fig. 7. This approach can be efficient for certain types of workloads, but it is more complex to set up and manage. It also can become a bottleneck for very large-scale distributed systems [33].

BigDL distributes the training data to worker nodes, which process it in parallel. Each worker node loads a copy of the model and computes the gradients for its portion of the data. The gradients are then combined from all worker nodes and used to update the model. The updated model is then sent back to all worker nodes, and the process is repeated until the model is trained. BigDL uses Spark RDDs to distribute the data and model across the worker nodes in the Spark cluster. This enables scalable and distributed deep learning. RDDs containing the training data and the model are zipped together and distributed across the worker nodes. Each worker node processes its assigned partition of the RDD, computing the forward and backward passes on the data and model, respectively. The gradients computed by each worker node are then aggregated across

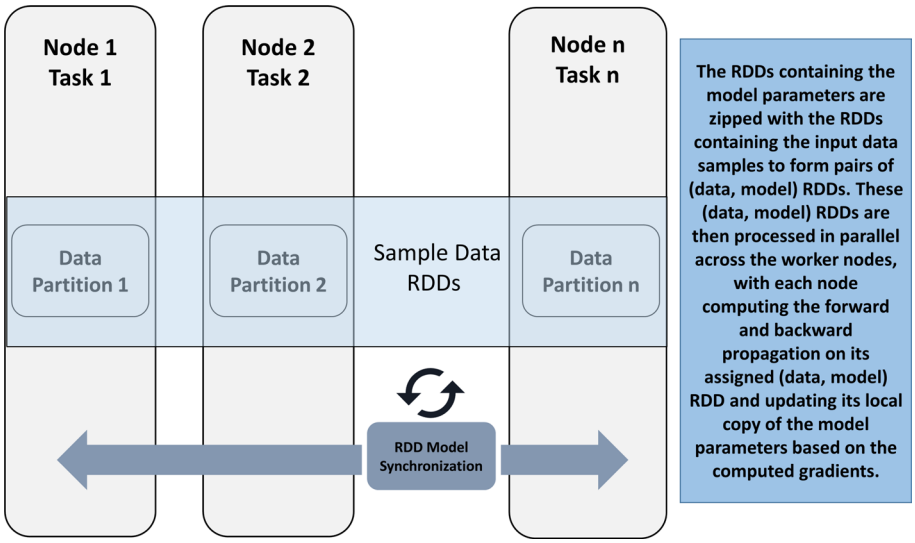

Fig. 8 Each worker node typically operates on one or more (data, model) RDDs, where each (data, model) RDD contains a subset of the input data and a replica of the model parameters. The number of (data, model) RDDs assigned to each worker node depends on the available resources in the Spark cluster and the configuration settings in BigDL

all the worker nodes, typically using an AllReduce operation. This aggregated gradient is then used to update the model's weights, and the process repeats for subsequent iterations until the model converges. During this training process, BigDL takes advantage of Spark's fault tolerance capabilities to ensure that the computation can recover from node failures. This means that if any worker node fails during the training process, the computation can be restarted from the point of failure without losing progress as shown in Fig. 8.

Training and inference in BigDL are both distributed and scalable, thanks to the use of Apache Spark RDDs. During training, the RDDs containing the training data and the model are distributed across the worker nodes in the Spark cluster. Each worker node then performs the forward and backward passes on the data and model, respectively. The gradients computed by each worker node are then aggregated across all the worker nodes and used to update the model's weights. This process repeats for subsequent iterations until the model converges. During inference, the input data is partitioned into multiple RDDs and distributed across the worker nodes. Each worker node then applies the pre-trained model to its assigned partition of the input data to generate predictions. The predictions from each worker node are then combined to form the final output. By distributing the data and model across the worker nodes, BigDL can achieve efficient and scalable training and inference, even for large datasets and complex models.

BigDL It allows deep learning applications to run on the Apache YARN/Spark cluster to directly process the production data, and as a part of the end-to-end data analysis pipeline for deployment and management has taken a completely different approach that directly implements an efficient AllReduce operation using existing primitives in Spark (e.g., shuffle, broadcast, in-memory cache, etc.), to mimic the functionality of a parameter server architecture. In particular, existing deep learning frameworks usually implement the all-reduce operation using MPI-like primitives; as

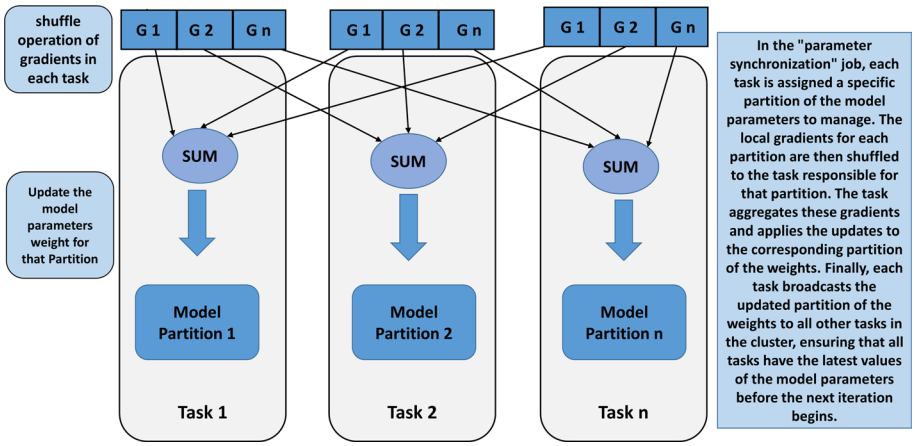

Fig. 9 Parameter synchronization is the process of aggregating the gradients computed by each worker node during the "model forward-backward" job and using them to update the model weights. This process occurs during the "parameter synchronization" job

a result, they often create long-running task replicas that coordinate among themselves with no central control. On the other hand, BigDL has adopted a logically centralized control for distributed training [34] BigDL's Parameter Manager uses a parameter server architecture with AllReduce to distribute gradients and update model weights in a distributed training setting. This architecture is more efficient than traditional parameter server architectures because it reduces communication overhead by avoiding the need for all workers to communicate with a central parameter server. To synchronize gradients, each worker first aggregates the gradients from its assigned partitions locally using all-reduce. Then, the aggregated gradients are sliced into chunks and exchanged between all the nodes in the cluster using all-reduce. Each node is responsible for a specific chunk. Next, each node retrieves gradients for the slice of the model that it is responsible for from all the other nodes and aggregates them in multiple threads using all-reduce. Finally, the updated weights are exchanged between all the nodes using all-reduce. At the end of this procedure, each node will have a copy of the updated weights. This architecture allows for efficient and scalable distributed training of deep learning models [35] as shown in Fig. 9.

BigDL 2.0 is a comprehensive AI toolkit for Apache Spark, combining the original BigDL and Analytics Zoo projects. It provides features for distributed deep learning (DLlib), Node scale-out Model and dependent packages (Orca), Ray on Spark (RayOnSpark), time series analysis (Chronos), privacy-preserving machine learning (PPML), and model serving (Serving). Some of these features will be used in our implementation process [36]. As a scalable end-to-end AI pipeline tool for distributed big data, BigDL is based on Apache Spark and supports a variety of deep learning frameworks, including TensorFlow, PyTorch, and OpenVINO. BigDL 2.0 enables users to scale their AI applications from a single laptop to large clusters, allowing them to process production-scale big data. Furthermore, BigDL 2.0 provides a high-level Spark ML pipeline for BigDL, simplifying the development and deployment of AI models on Spark clusters for users. BigDL has been utilized in a variety of real-world applications, such as object detection and image feature extraction at JD.com, product defect

detection at Midea, an NLP-based customer service chatbot for Microsoft Azure, image similarity-based house recommendation for MLS listings, LSTM-based time series anomaly detection for Baosight, and fraud detection for payment transactions for UnionPay [37].

Instance segmentation deep learning model

Instance segmentation is a complex computer vision task that assigns distinct labels to separate instances of objects belonging to the same class, providing pixel-specific object instance masks. Unlike semantic segmentation which classifies pixels based on classes, instance segmentation models classify pixels based on "instances". These algorithms can distinguish overlapping or very similar object regions based on their boundaries, regardless of the class a classified region belongs to. In semantic segmentation, smaller objects (with fewer pixels) are less significant. However, in instance segmentation, all objects, regardless of size, are equally important. This distinction makes instance segmentation applicable in various real-world scenarios. For example, in self-driving car technology, a vehicle navigating complex street scenarios such as crowded pedestrian areas or construction sites needs a detailed understanding of its surroundings. Instance segmentation plays a crucial role in this context. In the medical domain, instance segmentation has a wide range of applications. For instance, histopathologic images usually contain numerous nuclei of various shapes surrounded by cytoplasm. Recognizing and segmenting these nuclei using instance segmentation can aid in detecting severe diseases like cancer. It's also used for detecting tumors in MRI brain scans. Satellite imagery is another area where instance segmentation is highly useful. Major applications include identifying and counting cars, detecting ships for maritime security, preventing oil spills monitoring marine pollution, and segmenting buildings for geospatial analysis. Given that objects in satellite imagery are typically small and closely spaced concerning the image's resolution, pixel-wise methods are not very effective. Therefore, an instance segmentation network architecture can provide better separation between objects by understanding each object as a separate instance.

YOLACT [38] is an instance segmentation model It is used for instance segmentation in real-time and was the fastest real-time instance segmentation technique when it was introduced. it achieved a segmentation score of 29.8 Mask mAP on the COCO dataset at 33 frames per second when one Titan XP GPU was used as shown in Fig. 10.

YOLACT tackles the challenge of instance segmentation by dividing the task into two parallel subtasks: generating a dictionary of prototype masks and predicting a set of linear combination coefficients for each instance. This method implicitly learns to localize instance masks, thereby eliminating the need for the common localization step found in many instance segmentation methods. For example, other methods like Mask R-CNN have an explicit localization step such as ROIAlign.

The optimization inferences of the YOLACT model will be discussed in the following section. This will be beneficial for making the model more lightweight before deployment in a big data cluster, particularly in terms of inference or prediction time.

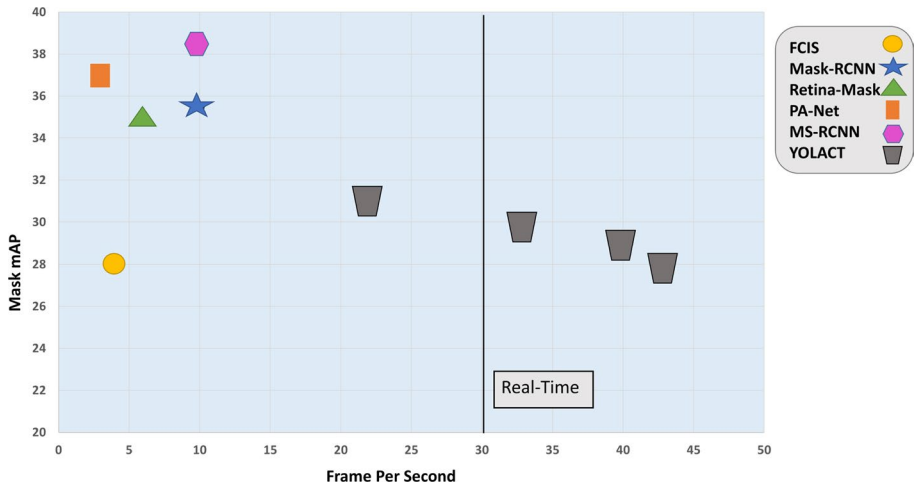

Fig. 10 Speed-performance trade-off for various instance segmentation methods on COCO [38]

Optimizing the YOLACT model for improved inference throughput

Optimizing model inference is vital for real-time deep learning applications like autonomous driving, focusing on throughput, memory, and energy. The process starts with converting the YOLACT PyTorch model to the ONNX format, a universal format for deep learning models that enhances interoperability between various AI frameworks. ONNX, an open-source project co-developed by Microsoft, Amazon, and Facebook, facilitates model conversion between any framework and the ONNX format. It supports faster inference using the ONNX model on the supported ONNX Runtime and is compatible with numerous machine learning frameworks like TensorFlow, PyTorch, and Scikit-learn. ONNX Runtime is designed for high performance and supports both CPU and GPU hardware. It also supports distributed training and inference across multiple devices and machines for improved performance and scalability [39]. The conversion of the YOLACT PyTorch model to ONNX format faced challenges due to unsupported layers in YOLACT, such as those in the Feature Pyramid Networks (FPN) class. The FPN class, which is optimized using torch script mode, allows for parallel multi-threading through the torch.jit.script module. This Just-In-Time (JIT) compiler improves the efficiency of PyTorch models in production. The FPN class extracts multi-level features for top-down up-sampling and fusion, creating multi-scale depth image features. However, this class inherits from the torch.jit.ScriptModule class, which often lacks tensor shape information, leading to potential size mismatch errors between ONNX and TorchScript definitions. PyTorch operates in two modes: eager mode for immediate operation execution and intuitive programming, and script mode, which compiles PyTorch code into an optimized TorchScript format for efficient execution in repeated prediction scenarios. PyTorch's JIT compilation optimizes TorchScript modules using runtime information, capturing the structure of a PyTorch model and saving it in a serialized format for use in different environments or devices. This is done without unifying the input dimensions of tensors in the FPN. When converting

a TorchScript-saved PyTorch model to ONNX, issues such as unsupported operations can occur. However, all YOLACT layers are supported in ONNX OpsetVersion 11. Another issue is related to tensor shapes - while PyTorch allows dynamic tensor shapes, ONNX requires static shapes. To address this issue in the detect transformation part used in the Backbone Network, Ma-Dan's suggestion was followed [40]. The input and output shapes were set before converting the model to ONNX in five FPN layers dimension sizes as [(69, 69), (35, 35), (18,18), (9,9), (5,5)]. This made it compatible with the ONNX format. Modifications were made to the YOLACT model to return prediction outputs directly from the forward method before post-processing. The Just-In-Time (JIT) compiler was disabled and the model was exported in ONNX format using OpsetVersion=11. The model was then visualized using the Netron App. The exported ONNX model was simplified using ONNX Simplifier [41], which removes redundant operators. The simplified model was then converted into OpenVINO format using the OpenVINO toolkit [42]. Developed by Intel, OpenVINO optimizes deep learning models for deployment on various devices, including CPUs, GPUs, and FPGAs. It supports ONNX format as an input, allowing models in ONNX format to be optimized and deployed on a variety of devices. Before conversion, the ONNX model was modified by separating the sparsity and locality prior tensor. These priors, which are learned during training and used as fixed components during inference, are used to generate object masks from the network's output feature maps. The sparsity prior thresholds the feature maps to generate binary masks, while the locality prior performs non-maximum suppression to remove overlapping masks. To make the ONNX model more lightweight, the model was decoded into a text format using the onnx.proto library. This allowed for modifications to be made, which were then encoded back into ONNX format. The prior tensor, generated based on input image size, feature map sizes, and predefined aspect ratios and scales, is used to create anchor boxes for object detection. This tensor remains constant during the prediction phase and serves as a reference for defining the location and size of the anchor boxes used in predicting the class and location of objects in an image. this prior tensor is only used during the post-processing stage, specifically in the Detect function for decoding location predictions, applying non-maximum suppression based on confidence scores, and thresholding to obtain a top k number of output predictions for both confidence scores and locations, including the predicted masks. It was used as a separate ONNX module with a size of 301 KB and integrated with the OpenVINO model prediction output result for post-processing purposes.

After experimenting with various versions of OpenVINO on different Ubuntu systems, the model optimizer dependencies were successfully installed on Ubuntu 18.04, using OpenVINO 2021.1.110. The model was converted into three files (.xml,.bin, and. mapping) for floating-point precision FP16/32. Using FP16 offers advantages such as improved speed and reduced memory usage of a neural network [43]. The Inference Engine was used to load these files, created by the model optimizer. The prediction results from the original YOLACT PyTorch model with Resnet50-FPN Backbone,

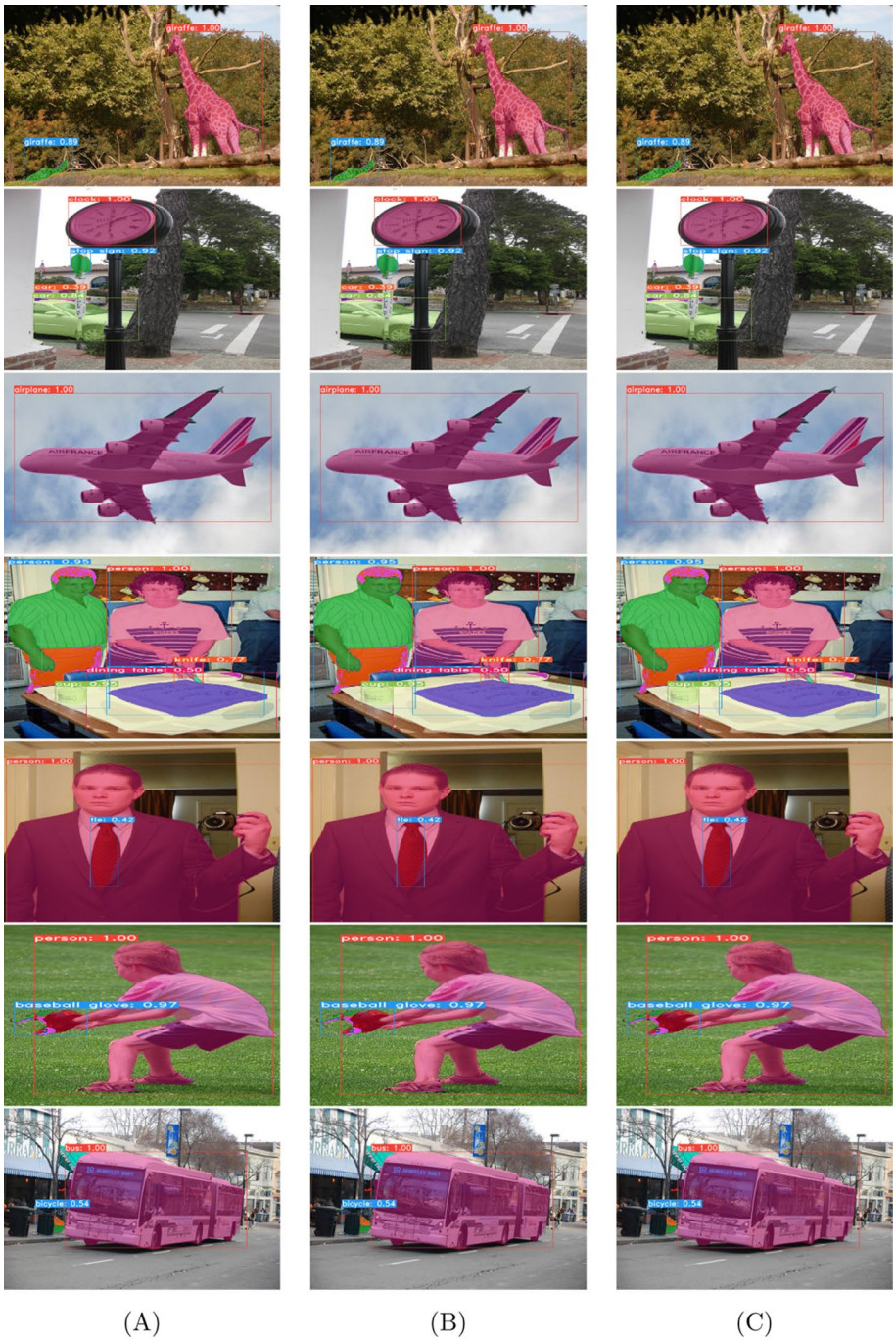

Fig. 11 Comparative Evaluation of Image Predictions: **A** Original YOLACT PyTorch Model, **B** Optimized ONNX Model, and **C** Optimized OpenVINO-FP16 Model demonstrates that the accuracy of instance masks and detection scores remained consistent across a variety of images. This consistency is crucial in ensuring the reliable performance of the models across different scenarios and datasets

ONNX, and OpenVINO FP16 models were evaluated on the same images respectively in Fig. 11. The accuracy of instance masks and detection scores remained consistent across

Fig. 12 Ground truth mask coefficients for reference to mask coefficient predictions from YOLACT PyTorch, ONNX, and OpenVINO-FP16 in Fig. 13

a variety of images, demonstrating the effectiveness of the optimized ONNX and OpenVINO models.

Another result has been obtained through the annotation mask of the ground truth image from the COCO dataset [44] making it a valuable resource for evaluating the optimized models as following Figs. 12 and 13 for further details.

The binary boundary Intersection-over-Union (IoU) values are used to evaluate the accuracy of the segmentation masks where The ground truth mask coefficients, demonstrated in Fig. 12, were used as a reference to compare with the output from the three models., have shown to remain stable with only minor deviations across all three models as demonstrated in Fig. 13: the original YOLACT PyTorch model, the optimized ONNX model, and the optimized OpenVINO-FP16 model. These minor deviations can be considered negligible. This consistency holds even when various dilation ratios are applied,

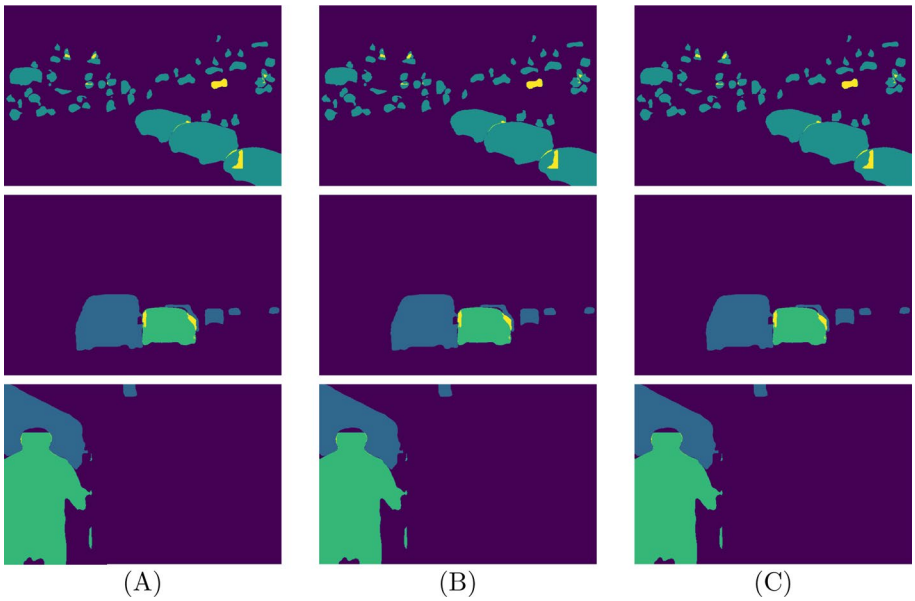

Fig. 13 The mask coefficient predictions from the YOLACT PyTorch Model (**A**), Optimized ONNX Model (**B**), and Optimized OpenVINO-FP16 Model are closely related to the ground truth images shown in the previous Fig. 12. This demonstrates the accuracy and consistency of these models in generating instance masks across a variety of images

further demonstrating the robustness of these models in generating accurate instance masks.

Comparative analysis of speed and performance across various YOLACT models

As demonstrated in the referenced Table 1, several experiments were conducted to measure the inference time taken for videos from three models (YOLACT original model with Resnet50-FPN Backbone, ONNX, OpenVINO) under different conditions using an 11th Gen Intel Core ^{TM} i7-11,370 H @3.30GHz processor.

In the **first experiment**, a 15-s video with 60 frames/second was tested. The ONNX model outperformed the others, reducing the processing time for one frame by **26.5** percent compared to the Torch Model.

The **second experiment** was conducted on a Virtual Machine with a 15-s video at 60 frames/second. Here, the OpenVINO FP16/32 models performed better than the Torch model, reducing the processing time for one frame by **28.8** percent.

In the **third experiment**, a 15-s video at 60 frames/second was used. The OpenVINO FP16 model performed better than the rest, reducing the processing time for one frame by **18** percent compared to the Torch Model.

Finally, in the **fourth experiment** with a 78-s video at 60 frames/second, the Optimized OpenVINO FP16 model outperformed the rest, reducing the processing time for one frame by **17** percent compared to the Torch Model.

As shown in Table 1, the YOLACT model with OpenVINO-FP16 optimization demonstrates superior efficiency in terms of one-frame processing time compared to other conditions.

Table 1 Shows the execution time taken for the process

Model name	Total time taken	OneFrame--Video Length time (s)	Machine configuration details
YOLACT Original	45 min 4s	3.00–15 (Video Length)	Python version 3.6
ONNX Model	26 min 9 s	1.74–15	Anaconda windows 10
OpenVINO-FP16	29 min 50 s	1.98–15	With 16 GB RAM
OpenVINO-FP32	29 min 38 s	1.97–15	Torch V 1.10.2+cpu
YOLACT Original	54 min 43 s	3.64–15 (Video Length)	Ubuntu 20.04 VM
ONNX Model	32 min 48 s	2.18–15	Torch V 1.8.1+cu102
OpenVINO-FP16	30 min 10 s	2.01–15	4 GB RAM-python 3.6
OpenVINO-FP32	30 min 9 s	2.01–15	Total cores = 4
YOLACT Original	35 min 47 s	2.30–15 (Video Length)	Ubuntu 20.04 VM
ONNX Model	32 min 48 s	2.10–15	Torch version 1.2.0
OpenVINO-FP16	24 min 22 s	1.60–15	6 GB RAM-python 3.7
OpenVINO-FP32	30 min 12 s	2.01–15	Total cores = 6
YOLACT Original	1 h 33 min 2s	2.30–78 (Video Length)	Ubuntu 20.04 VM
ONNX Model	1 h 21 min 34 s	2.09–78	Torch version 1.2.0
OpenVINO-FP16	1 h 3 min 35 s	1.63–78	6 GB RAM-python 3.7
OpenVINO-FP32	1 h 8 min 31 s	1.76–78	Total cores = 6

Table 2 Table shows the execution time taken on spark stand-alone local mode

Experiment (1) Machine Configuration: Ubuntu 20.04 VM with 7 GB RAM, Number processors = 2, Number cores each processor = 3, Python Version=3.7 Anaconda, Java Version = 1.8.0–201, PySpark Version = 2.4.6, spark.driver.memory = 4 GB, The number of cores = 2	Local mode spark stand-up time Total execution time for 18 images as one patch task job	17 s The total time was 43 s, with 2.3 s allocated to process a single image (one frame).
Experiment (2) Machine Configuration: Ubuntu 20.04 VM with 7 GB RAM, Number processors = 2, Number cores each processor = 3, Python Version=3.7 Anaconda, Java Version=1.8.0-201, PySpark Version=2.4.6, spark.driver.memory = 4 GB, **The number of cores = 4**	Local mode spark stand-up time Total execution time for 26 images as one patch task job	14 s The total time was 48 s, with 1.8 s dedicated to processing a single image (one frame).

This suggests that this optimized model could be a promising candidate for inference tasks on a big data cluster. Further investigations could include performance evaluation of this optimized model on larger datasets, scalability assessment across multiple nodes in the cluster, and analysis of its resource utilization and cost-effectiveness. The insights gained from these analyses could provide valuable guidance for future deployment of the optimized YOLACT model in a distributed deep learning environment.

Table 3 Execution time for the first experiment on a standalone spark cluster with 2-VM workers

Cluster Configuration: - Master VM in specs (7 GB RAM, Total processor cores = 6, HD = 130 GB)—Slave1 VM in specs (4.5 GB RAM, Total processor cores = 4, HD = 130 GB) And Spark cluster configured by the driver- memory = 2 GB, num-executors = 2, executor-cores = 2, driver-cores = 2, executor-memory = 3 GB	Spark cluster stand-alone stand-up time with 2 Executors	14 s
	Total sum execution time for 21 images as one patch task job in 2 Executors	1.28 min
	Work Executors in Master VM processing time	42 s
	Work Executors in slave1 VM processing time	35 s
	One Frame/image Processing time	42/21 = 2 s it calculated by the long executor time
	WIFI speed on the local network	130 Mbps

Table 4 Execution time for the second experiment on a standalone spark cluster with 2-VM workers

Cluster Configuration: - Master VM in specs (7 GB RAM, Total processor cores = 8, HD = 130 GB)—Slave1 VM in specs (4.5 GB RAM, Total processor cores = 6, HD = 130 GB) And Spark cluster configured by driver-memory =3 GB, num-executors = 2, executor-cores = 4, driver-cores = 2, executor-memory = 3 GB	Spark cluster stand-alone stand-up time with 2 Executors	14 s
	Total sum execution time for 26 images as one patch task job in 2 Executors	1.28 min
	Work Executors in Master VM processing time	35.7 s
	Work Executors in slave1 VM processing time	36.7 s
	One Frame/image Processing time	36.7/26 =1.4 s it calculated by the long executor time
	WIFI speed on the local network	130 Mbps

YOLACT OpenVINO distributed inference on big data clusters using virtual machines

BigDL on apache spark in standalone local mode (single node)

The inference times from the two experiments are presented in Table 2. These experiments were conducted on a Ubuntu virtual machine, hosted on a machine equipped with an 11th Gen Intel Core ^{TM} i7-11370H processor.

The two experiments presented in Table 2 varied the number of cores used for the executor. In the first experiment, 2 cores were used, while in the second experiment, the number of cores was increased to 4. This increase led to a reduction in processing time for one frame, resulting in a 12 percent decrease in inference time. These results suggest that increasing the number of executor cores can positively impact the performance and efficiency of the distributed deep-learning tasks, leading to faster processing times.

BigDL on apache spark in standalone cluster mode

In a Spark cluster, the Spark driver is responsible for running the application's code and storing its logic. Data is distributed across all the worker nodes in the cluster. When the application is running, the driver sends the code to the worker nodes, where it is executed and the results are returned to the driver. In practice, the authors have implemented a process that involves installing all the necessary dependencies, including the BigDL library, on all worker machines.

This is required because Spark standalone mode requires each application to manually run an executor worker on every node in the cluster. Once the dependencies are installed, the master node runs a container as an Executor on each worker node based on the input configuration. The results obtained under various conditions will be discussed in the context of the following cluster architecture.

First experiment with a 2-VM spark cluster (configuration details in Table 3)

Second experiment with a 2-VM spark cluster (configuration details in Table 4)

In the two experiments detailed in Tables 3 and 4, the number of Executor Cores was adjusted. The first experiment used 2 cores per worker, while the second experiment increased this to 4 cores per worker. This adjustment resulted in a decrease in processing time for one frame by 18 percent, indicating that increasing the number of Executor Cores can enhance performance and reduce processing time in distributed deep learning systems. However, to achieve optimal results, it's crucial to fine-tune the number of Executor Cores based on the specific hardware and workload characteristics.

Third experiment with a heterogeneous 3-VM spark cluster across two machines (configuration details in Table 5)

In this experiment, as shown in Table 5, the executor in the Slave 2 VM workers took the most time to execute compared to the other workers. This is because the Slave 2 VM workers had an older version of the Intel processor (3rd Generation), while the other workers had newer processors (11th Generation). As a result, this experiment took longer to generate the outputs because the faster nodes had to wait for the slower node (Slave 2) to finish its task. This is an example of Synchronous Parallel Execution, a concept discussed earlier, where all nodes in a distributed system must wait for each other to finish before moving forward. This highlights the importance of hardware specifications, such as processor generation, in distributed deep learning systems.

Table 5 Execution time for the third experiment on a heterogeneous standalone spark cluster with 3-VM workers across two machines

Cluster Configuration: - Master VM in specs (7 GB RAM, Total processor cores = 8, HD = 130 GB)—Slave1 VM in specs (4.5 GB RAM, Total processor cores = 6, HD = 130 GB)—Master and Slave1 Processor: - 11th Gen Intel (R) Core (TM) i7-11370H @ 3.30 GHz—Slave2 VM in specs (6 GB RAM, Total processor cores=6, HD=130 GB)—Slave2 Processor: Intel (R) Core (TM) i7-3630QM CPU @ 2.40 GHz. And Spark cluster configured by driver-memory = 3 GB, num-executors = 3, executor-cores = 3, driver-cores = 3, executor-memory = 3GB	Spark cluster stand-alone stand-up time with 3 Executors	63 s
	Total sum execution time for 35 images as one patch task job in 3 Executors	4.43 min
	Work executors in master VM processing time	27 s
	Work executors in slave2 VM processing time	**3.37 min**
	One Frame/image Processing time	3.37*60 /35 = 5.77 s it calculated by the long executor time
	WIFI speed on the local network	130 Mbps

Table 6 YARN client mode cluster configuration with 3 executors

Machine type	Specifications
Master VM	- 6 GB RAM - 6 processor cores - 130 GB HD
Slave1 VM	- 5 GB RAM - 4 processor cores - 130 GB HD
Slave2 VM	- 5 GB RAM - 4 processor cores - 130 GB HD
YARN node manager memory	- Master node: 5 GB - Slave1 and Slave2 nodes: 4 GB
Executor/Driver	- Number of nodes: 3 - Worker cores: 2 - Worker memory: 3 GB - Driver memory: 2 GB - Driver cores: 2

Table 7 YARN client mode cluster configuration with 2 executors

Machine type	Specifications
Master VM	- 6 GB RAM - 6 processor cores - 130 GB HD
Slave1 VM	- 5 GB RAM - 4 processor cores - 130 GB HD
YARN node manager memory	- Master node: 5 GB - Slave1 node: 5 GB
Executor/Driver	- Number of nodes: 2 - Worker cores: 2 - Worker memory: 3 GB - Driver memory: 2 GB - Driver cores: 2

BigDL on apache spark with YARN

The YARN cluster experiment used the conda-pack feature from BigDL Orca to scale out the master node by distributing Python dependencies across the cluster worker nodes. The packed environment size was about 2 GB, and YARN started containers with configured values without the need for installation on other worker nodes, unlike the standalone Spark cluster as mentioned before. However, the results of the execution time did not show better performance compared to the standalone Spark cluster. This could be due to various factors such as the small cluster size and limited resources in the experiment setup. Additionally, it was observed that the startup time to initiate the configured containers and make them ready for executing the task job was about 1 h in this experiment. However, each time YARN initiates the containers on all machines, this time decreases to about 10 min after clearing the cache on all VM machines. This measurement was done on VMs with Intel (R) Core (TM) i7-3630QM CPU @ 2.40 GHz processor. The results obtained under various conditions will be discussed in the context of the following YARN cluster architecture.

First experiment with a 3-VM spark on YARN cluster in client mode (configuration details in Table 6)

The master VM serves as the driver and data node, while the Slave1 VM functions as an executor node. Slave2 VM operates similarly to Slave1 VM. The detailed YARN cluster configuration is available in Table 6. Under these conditions, One Frame/Image Execution time was observed to be 4.3 s at this condition, YARN initiates 1 container executor on the slave1 and slave2 VM data node and also one container for the driver and one for the executor on the master VM which has the both Name Node and Data Node.

Second experiment with a 2-VM spark on YARN cluster in client mode (configuration details in Table 7)

The master VM serves as the driver and data node, the Slave1 VM functions as an executor node. The detailed YARN cluster configuration is available in Table 7. Under these conditions, One Frame/Image Execution time was observed to be 2.8 s. at this condition, YARN initiates 1 container executor on the slave1 VM data node and also one container for the driver and one for the executor on the master VM which has the Name Node and Data Node.

In the comparison of two tests on a Spark on YARN Cluster in Client Mode, an interesting observation was made. Reducing the number of Virtual Machines (VMs) from three to two actually decreased the execution time for one frame/image from 4.3 s to 2.8 s. This suggests that sometimes, using fewer resources can lead to better performance and efficiency. This could be due to factors like lower network latency and more effective workload distribution. However, further investigation is needed as future work to fully understand why this is happening and how to best optimize the configuration of the Spark on the YARN Cluster.

Table 8 YARN client mode cluster configuration with 2 executors in heterogeneous system with 2-VM workers across two machines

Machine type	Specifications
Master VM	- 6 GB RAM - Total processor cores: 8 - 130 GB HD - Processor: 11th Gen Intel (R) Core (TM) i7-11370H @ 3.30GHz
Slave VM	- 6 GB RAM - Total processor cores: 4 - 130 GB HD - Processor: Intel (R) Core (TM) i7-3630QM CPU @ 2.40GHz
Network	- Connection: WIFI Local Network - Network Speed: 26 Mbps
Executor/Driver	- Number of nodes: 2 - Worker cores: 2 - Worker memory: 3 GB - Driver memory: 2 GB - Driver cores: 2

Third experiment with a 2-VM spark on YARN cluster in client mode on a heterogeneous system (configuration details in Table 8

The master VM operates as both the driver and data node, equipped with a processor from the 11th Gen Intel (R) Core (TM) i7-11370H family, running at 3.30 GHz. The slave 1 VM, connected to the master VM on a separate machine, utilizes a processor from the Intel (R) Core (TM) i7-3630QM CPU family, with a clock speed of 2.40 GHz. Under these conditions, the time taken to process one frame/image was recorded as **2.2 min.**

In this setup, YARN initiated one container executor on the slave1 VM's data node, which was on a separate machine. Additionally, it created one container for the driver and another for the executor on the master VM, which also hosted the Name Node and Data Node. These two VMs were connected via a WiFi local network with a bandwidth of 26 Mbps. The detailed YARN cluster configuration is available in Table 8.

Under these conditions, the time taken to process one frame/image was recorded as 2.2 min. YARN initiated one container executor on the slave VM's data node and created two containers on the master VM for the driver and executor. These two VMs were connected via a WiFi local network with a bandwidth of 26 Mbps.

This experiment highlights the impact of hardware specifications on execution time in a distributed system. Despite having fewer resources, the execution time was significantly longer compared to previous experiments. This could be attributed to the older generation processor in the slave VM, which could not perform tasks as quickly as the newer processor in the master VM.

This is an example of synchronous parallel execution, where all nodes in a distributed system must wait for each other to finish before moving forward. In this case, the faster node (master VM) had to wait for the slower node (slave VM) to complete its task, resulting in a longer overall execution time. This highlights the crucial role of hardware characteristics, such as the generation of the processor and its clock speed, in the performance of distributed deep learning systems.

YOLACT OpenVINO distributed inference on big data clusters on azure databricks: experimental results

Azure Databricks is a cloud-based platform that merges the capabilities of Apache Spark and the robust cloud infrastructure of Microsoft Azure. It offers an interactive workspace that enables data scientists, data engineers, and business analysts to collaborate on big data and machine learning projects.

To manage the storage and access of image data and models, Azure Cloud computing was utilized. A new resource was created, which included a Container Data Lake storage class and resources specifically for Databricks. The integration of Azure Storage with Azure Databricks was achieved through the use of Azure Key Vault. This allowed for efficient loading of models and images, and the storage of resulting output in designated directories within the container. This method facilitated streamlined and effective management of data throughout the deep learning pipeline. BigDl provides a user guide for running a BigDL program on a Databricks cluster [45]. New resources were allocated for Databricks and Container Data Lake storage, which allowed for the creation of separate directories for storing image data and models. Azure Storage was then integrated with

Azure Databricks using Azure Key Vault, streamlining the process of loading the model and images, and writing the results to the designated container.

Azure Databricks offers the latest versions of Apache Spark and facilitates seamless integration with open-source libraries. The configured cluster Databricks Runtime Version is 9.1 LTS, which includes Apache Spark 3.1.2 and Scala 2.12. The pipeline was constructed in three stages: reading the images as a Spark data frame, applying a chain of preprocessing, and finally loading the model and applying inference on preprocessed feature images. In the first stage, the NNImageReader class from BigDL was utilized. This class provides a variety of methods to read image data from different sources, including local file systems, Hadoop Distributed File System (HDFS), and Amazon S3. It supports a wide range of image file formats, including JPEG, PNG, BMP, and GIF. Despite attempts to read the images as a Spark DataFrame from Azure Data Lake Storage, the job was not distributed across all nodes, achieving full parallelism or utilizing all cluster cores. Instead, only one task was active, utilizing just a single core and resulting in increased processing time. Various configurations were experimented with, including increasing the number of tasks per stage, in an effort to resolve this challenge [46]. However, the issue was only resolved when using the original Spark library to read the images as a Spark DataFrame. This highlights the importance of library selection in achieving optimal performance and resource utilization in distributed computing environments.

A pipeline job was developed to perform instance segmentation using the OpenVINO model on top of BigDL on Azure Databricks. The pipeline consists of four stages: **Stage-1** The first stage of the pipeline job is to read the images as a Spark DataFrame. This typically takes around 7 s in all experiments, but this value is excluded from the subsequent Table 10 of results for clarity; **Stage-2**, This stage is dedicated to preprocessing the DataFrame on the cluster. All necessary transformations are implemented in a distributed manner, following the approach of the original YOLACT model that uses PyTorch transformation functions. A BigDL identity model is constructed using the ChainedPreprocessing class to execute these transformations. The transformations involve converting Row to ImageFeature, applying Image Resize, implementing specific normalization and standardization, and finally converting ImageFeature to Tensor. This stage is vital for preparing the data for further processing and analysis. In this preprocessing stage, the NNModel class from BigDL-DLlib is used to incorporate a simple BigDL model or an identity model. An identity model is a neural network architecture that passes input directly to the output without any nonlinear transformations. This allows for the creation of transformation prediction columns as preprocessed images. To facilitate this process, a PySpark UDF (User Defined Function) is defined that converts the preprocessed one-dimensional array into a three-dimensional array. This conversion is necessary as the model expects input in this format for prediction purposes; **Stage-3**, In this stage, the Estimator class from BigDL-Orca is used to load the OpenVINO model on the cluster. BigDL-Orca is responsible for broadcasting the model and caching it on each worker in the cluster, enabling distributed prediction. This allows for performing instance segmentation using the OpenVINO pipeline on top of BigDL on Azure Databricks; Finally in **Stage-4**, In this final stage, the pre-configured post-processing method available in YOLACT Original is applied to the results obtained from the prediction stage. The

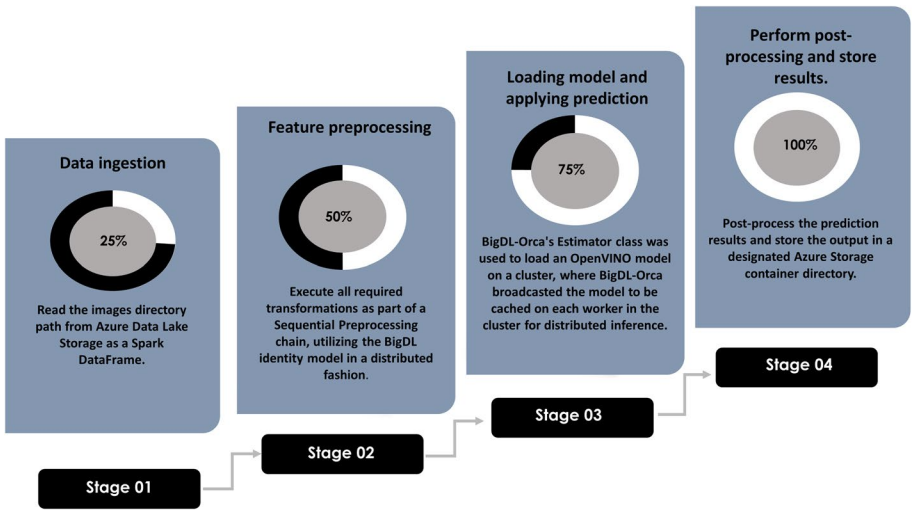

Fig. 14 Distributed instance segmentation pipeline using OpenVINO-FP16 and OpenVINO Estimator on BigDL in Azure Databricks

Table 9 Pseudocode for image processing pipeline using OpenVINO estimator on top of BigDL on Azure Databricks

Pipeline Pseudo-code

```
Data Ingestion:
Read the images directory path from Azure Data Lake Storage as a Spark DataFrame

Feature Preprocessing:
Define preprocessing transformations using the BigDL identity model
Apply chained preprocessing transformations to the distributed image dataset

Loading Model and Applying Inference:
Import necessary libraries from BigDL and Orca
Load the OpenVINO model using the Estimator class
Broadcast the model to be cached on each worker in the cluster using BigDL-Orca
Apply inference on the distributed dataset using the loaded model

Post-processing and Output Saving:
use the  pre-configured post-processing method in YOLACT original
Apply post-processing on the results from the prediction stage
store the resulting output in a designated directory  within the container in Azure
     Storage
```

resulting output is stored in a designated directory within the Azure Storage container. This entire process is illustrated in Fig. 14.

Here is the pseudocode for implementing an instance segmentation pipeline using Optimized OpenVINO-FP16 and OpenVINO Estimator on BigDL within Azure Databricks, as shown in Table 9.

In the third stage of the pipeline, as detailed in Table 10, the prediction duration was recorded. In Experiment 23, a cluster of six workers was utilized with 288 partitions. During the process of reading images as a Spark data frame, the data frame, referred to as "R", was repartitioned from its original 170 partitions to 288. This was done to reduce the number of images per partition and optimize resource utilization.

In the third stage of the pipeline, as detailed in Table 10, the prediction duration was recorded. In Experiment 23, a cluster of six workers was utilized with 288 partitions. During the process of reading images as a Spark data frame, the data frame, referred to as "R", was repartitioned from its original 170 partitions to 288. This was done to reduce

Table 10 This table shows the inference time for processing different numbers of images across different cluster architecture configurations

Num	Num workers	Cores per driver	Cores per worker	Memory driver	Memory worker	Number images	Num partitions	Stage-2 time sec	Stage-3 time	Patch size images	Failed tasks
1	1	4	4	14	14	70	8(R)	8	6.2 min	8/11	–
2	2	4	4	14	14	70	8	0.5	2.1 min	7/9	–
3	2	4	4	28	14	128	16(R)	8	8.5min	8	4failed tasks
4	2	4	4	14	28	128	8	0.5	3.7 min	17	–
5	2	4	4	14	28	128	16(R)	9	7.6 min	8	–
6	2	4	16	14	64	320	31	0.5	3.8 min	10	–
7	3	8	16	28	64	500	46	0.5	3.3 min	11	–
8	3	8	32	56	128	500	84	0.5	2.3 min	6	–
9	3	8	32	56	128	1000	91	0.5	5.4 min	11	–
10	4	16	32	56	128	1355	124	0.5	4.8 min	11	–
11	4	16	32	56	128	500	125	0.5	2.0 min	4	–
12	4	16	32	56	128	500	256(R)	11	5.3 min	4/5	–
13	4	16	32	56	128	70	70	0.5	47 sec	1	–
14	3	8	32	28	128	128	64	0.5	1.4 min	2	–
15	3	8	32	28	128	320	80	0.5	1.6 min	4	–
16	3	8	32	28	128	320	96(R)	9	1.3 min	8	–
17	5	16	32	64	128	1700	155	0.5	9.3 min	10/11	95failed tasks
18	5	16	32	64	128	1700	288(R)	15	5.3 min	7/6	32failed tasks
19	6	16	32	56	128	1700	288(R)	15	4.7 min	5/6	32failed tasks
20	6	16	32	56	128	1700	320(R)	15	4.5 min	5/6	–
21	6	16	32	112	128	1700	352(R)	15	5.8 min	4/5	63failed tasks
22	6	16	32	112	128	2000	352(R)	15	5.7 min	5/6	–
23	6	16	32	112	128	1355	288(R)	8	4.3 min	5/6	–
24	6	16	32	112	128	1355	170	0.5	3.3 min	7/8	–

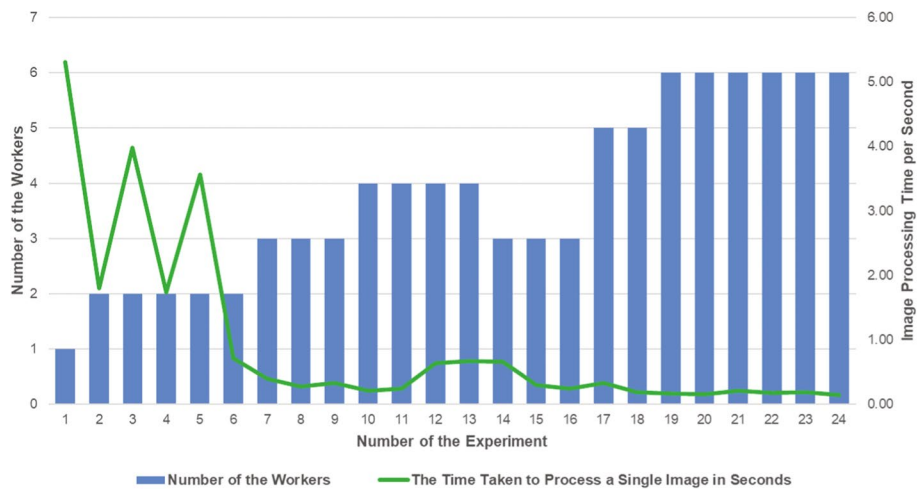

Fig. 15 Trade-offs between Number of Workers and Processing Time for a Single Image

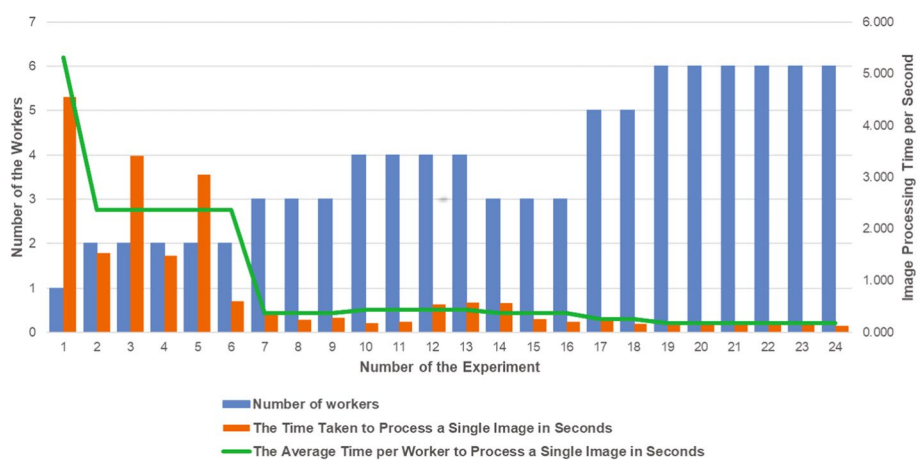

Fig. 16 Scalability of image processing: impact of number of workers on average processing time

the number of images per partition and optimize resource utilization. Interestingly, in Experiment 24, where the data frame was not repartitioned and the number of partitions remained at 170, a decrease in prediction time was observed compared to Experiment 23. This implies that each partition of the data frame contained a patch of images, as indicated in the "Patch size images" column in Table 10, for example, in Experiment 2, the term "7 / 9" for patch size images indicates that some of the 8 partitions contain 7 images, while others contain 9 images. The total number of images should be equal to the number of partitions multiplied by the patch or partition size of images. Data partitioning has a significant impact on prediction time in distributed systems. Therefore, it is crucial to choose the optimal number of partitions when configuring distributed systems. The successful completion of Experiment 17, despite having 96 failed tasks, demonstrates the fault tolerance of distributed Big Data clusters. This resilience is a crucial feature of Big Data systems, allowing them to recover from task failures without restarting the entire job. In addition, experiments 3, 18, 19, and 21, executed on the Databricks

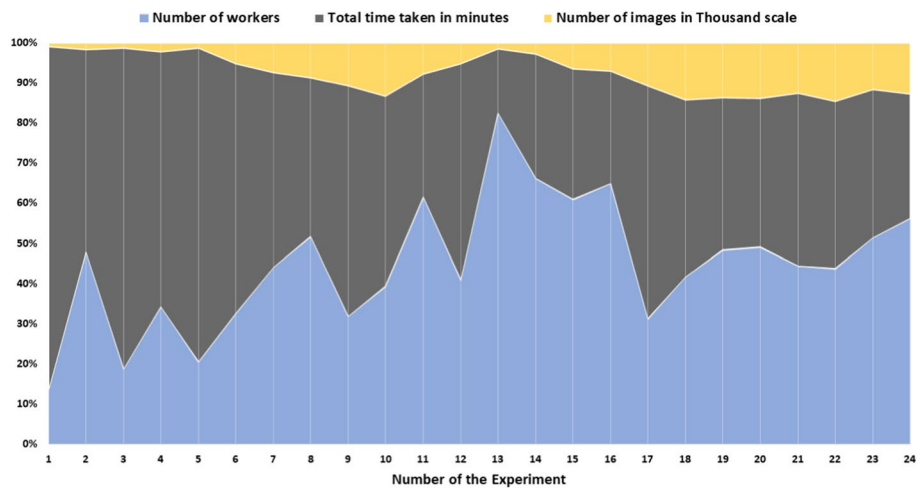

Fig. 17 Exploring the relationship between worker count, image count, and total processing time with 100% stacked area charts

cluster with varying architecture configurations, provide additional illustration of this concept through the automated restart of certain failed tasks.The results of these experiments are presented in Table 10. These experiments highlight the importance of fault tolerance in big data systems. Even when tasks fail, the system is designed to recover and continue processing. This ensures that the overall performance of the system is not significantly affected by individual task failures. In conclusion, Distributed systems are designed to be robust and reliable, despite failures. This is important because complex computational tasks can often be divided into smaller tasks that can be executed in parallel on different nodes of the system. If one node fails, the other nodes can continue executing the failed tasks, and the overall system can still complete the task. Some distributed systems, such as Apache Spark, support a fault tolerance feature that only restarts failed tasks, rather than the entire job. This can be a more efficient approach, as it can reduce the amount of time and resources required to recover from a failure.

The visual representations of the insights obtained from the experiments, as outlined in Table 10, can be found in Figs. 15, 16, and 17. Figure 15 specifically demonstrates the correlation between the processing time for one image per second and the number of workers. It reveals that processing time decreases significantly as the number of workers increases. For example, processing one image on a single worker takes 5 s, but this is reduced to about 0.15 s when using 6 workers. This shows that increasing the worker count significantly improves processing speed.

Moreover, Fig. 16 also shows that the average time per worker in seconds decreased as the worker count increased, suggesting that a higher number of workers improves efficiency. Figure 17 further highlights that reducing the input image size and increasing the number of workers can lead to faster image processing times. This suggests an inverse relationship between image size and processing time, where smaller image sizes and a higher number of workers can result in significantly faster processing times.These results can be used as helpful tips for making image processing tasks quicker and more efficient.

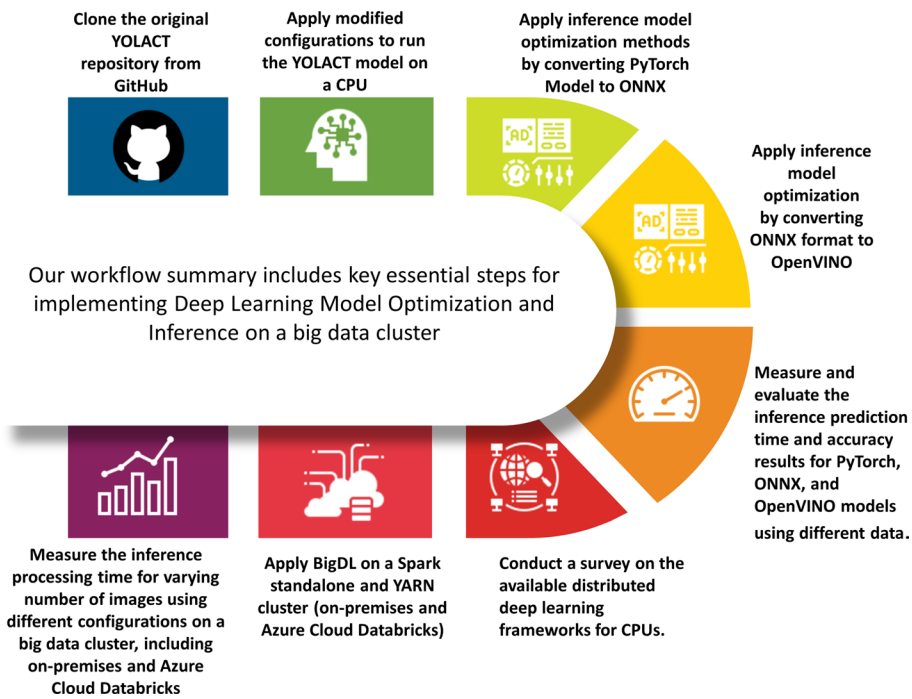

Fig. 18 This is our workflow summary outlines the key steps involved in completing a task

Figure 18 illustrates a comprehensive workflow for implementing Deep Learning Model Optimization and Inference on a big data cluster, The workflow begins by cloning the original model repository from GitHub and modifying the model to run on a CPU. The model is then optimized for inference by converting it from PyTorch to ONNX and OpenVINO formats.

The inference prediction time and accuracy results for PyTorch, ONNX, and OpenVINO models are measured and evaluated. Next, a survey of the available distributed deep learning frameworks was performed. BigDL was then applied to a Spark standalone and YARN cluster, both on-premises and on Azure Cloud Databricks. Finally, the inference processing time for varying numbers of images using different configurations on a big data cluster, both on-premises and on Azure Cloud Databricks, was measured.

Conclusions and future work

This paper proposes a distributed system to run the optimized YOLACT instance segmentation model, a complex and large deep learning model, on-premises and in the cloud. The paper describes the end-to-end data loading and preprocessing pipeline and evaluates the inference time on different frameworks, including PyTorch, ONNX, and OpenVINO. The experimental results showed that increasing the number of executors across the cluster significantly sped up inference time, which could lead to cost savings for server resources. However, the authors observed some excessive memory usage issues that need to be addressed in future work. Additionally,

exploring the possibility of running BigDL on a distributed GPU cluster using Spark 3.x is a potential direction for further investigation.

Overall, this study demonstrates the feasibility of DDL for deploying and scaling sophisticated deep learning models, such as YOLACT, on big data clusters for instance segmentation tasks. It also addresses the challenges of deploying optimized complex deep learning models on DDL big data clusters and explains the choice of BigDL as the DDL framework for this study.

Acknowledgements
Not applicable.

Author contributions
ME designed the approach, performed the software, and wrote the methodology, experimental results, and literature review. HEA reviewed the methodology and experimental results and provided valuable ideas for a better article framework. IE and HEA double-checked the manuscript. All authors read and approved the final manuscript.

Funding
Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB). This research was self-funded and conducted independently by the author. The author received supervision and guidance from HEA and IE at the Faculty of Engineering, Ain Shams University.

Availability of data and materials
The data that support the findings of this study are publicly available. COCO val2017 dataset, for the Instance Segmentation The dataset, can be accessed at the official COCO dataset website: https://cocodataset.org/#download. Alternatively, you can download the dataset directly from the following link: http://images.cocodataset.org/zips/val2017.zip. Annotation for COCO val2017 dataset The annotation file can be downloaded directly from the following link: https://huggingface.co/datasets/merve/coco/blob/main/annotations/instances_val2017.json.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 19 May 2023 Accepted: 16 December 2023
Published online: 02 January 2024

References

1. Ng, Andrew, Machine learning yearning. deeplearning. ai. URL: https://www.deeplearning.ai, https://github.com/ajaymache/machine-learning-yearning 2018.
2. Teerapittayanon S, McDanel B, Kung HT. Distributed deep neural networks over the cloud, the edge, and end devices. 2017 IEEE 37th international conference on distributed computing systems (ICDCS). IEEE, 2017.
3. Krichevsky N, St Louis R, Guo T. Quantifying and improving performance of distributed deep learning with cloud storage. 2021 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2021.
4. *10 Best Machine Learning Software*, unite.ai. Alex McFarland, 2022. [Online]. Available: https://www.unite.ai/10-best-machine-learning-software/ Accessed on: 2022.
5. Dai JJ, Wang Y, Qiu X, Ding D, Zhang Y, Wang Y, Jia X, Zhang CL, Wan Y, Li Z, Wang J et al. Bigdl: a distributed deep learning framework for big data. Proceedings of the ACM Symposium on Cloud Computing, 2019;50–60.
6. Sergeev A, Del Balso M. Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 2018.
7. MPI Forum. Message Passing Interface (MPI) Forum, Home Page. [Online]. Available: http://www.mpi-forum.org Accessed on: 2022.
8. Andrew Gibiansky. Bringing HPC Techniques to Deep Learning. [Online]. Available: https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/ Accessed on: 2022.
9. Horovod on Spark,Horovod on Spark user guide. [Online]. Available: https://horovod.readthedocs.io/en/stable/spark_include.html Accessed on: 2023.
10. End-to-End Deep Learning with Horovod on Apache Spark, Databricks. [Online]. Available: https://www.databricks.com/session_na20/end-to-end-deep-learning-with-horovod-on-apache-spark Accessed on: 2023.
11. Distributed Deep Learning with Ray Train User Guide, Ray. [Online]. Available: https://docs.ray.io/en/latest/train/dl_guide.html Accessed on: 2023.

12. Distributed training with TensorFlow user guide, TensorFlow. Google, 2015. [Online]. Available: https://www.tensorflow.org/guide /distributed_training Accessed on: 2022.
13. PYTORCH DISTRIBUTED OVERVIEW, pytorch. Facebook. [Online]. Available: https://pytorch.org/tutorials/beginner/dist_overview.html#pytorch-distributed-overview Accessed on: 2023.
14. Shen L, et al. Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint arXiv:2006.15704 (2020).
15. Nilesh Barla. Distributed Training: Frameworks and Tools, neptune.ai blog. [Online]. Available: https://neptune.ai/blog/distributed-training-frameworks-and-tools Accessed on: 2023.
16. Najafabadi MM, Villanustre F, Khoshgoftaar TM, Seliya N, Wald R, Muharemagic E. Deep learning applications and challenges in big data analytics. J Big Data. 2015;2(1):1–21.
17. Berloco F, Bevilacqua V, Colucci S. A systematic review of distributed deep learning frameworks for big data. International Conference on Intelligent Computing, 2022;242–256, Springer.
18. Hwang C, Kim T, Kim S, Shin J, Park K. Elastic resource sharing for distributed deep learning, 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 2021;721–739.
19. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 2014.
20. Szegedy C, et al. Inception-v4, inception-resnet and the impact of residual connections on learning. Proceedings of the AAAI conference on artificial intelligence. 2017;31(1).
21. Liu S, Zhang H, Jin Y. A survey on surrogate-assisted efficient neural architecture search. arXiv preprint arXiv:2206.01520 2022.
22. Hegde V, Usmani S. Parallel and distributed deep learning. 2016;31:1-8.
23. Chen CC, Yang CL, Cheng HY. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. arXiv preprint arXiv:1809.02839 2018.
24. Jostins L, Jaeger J. Reverse engineering a gene network using an asynchronous parallel evolution strategy. BMC Syst Biol. 2010;4:1–16.
25. Zhang J, Xiao J, Wan J, Yang J, Ren Y, Si H, Zhou L, Tu H. A parallel strategy for convolutional neural network based on heterogeneous cluster for mobile information system. Mobile Inf Syst. 2017; (2017).
26. McDonald C, Robert Evans Jason Lowe. Accelerating Apache Spark 3.0 with GPUs and RAPIDS , nvidia Blog, 2020. [Online]. Available: https://developer.nvidia.com/blog/accelerating-apache-spark-3-0-with-gpus-and-rapids/ Accessed on: 2021.
27. Distributed TensorFlow on Apache Spark 3.0,Madhukar. Madhukar's Blog, 2020. [Online]. Available: https://blog.madhukaraphatak.com/tensorflow-on-spark-3.0/ Accessed on: 2021.
28. Shi S, Wang Q, Chu X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech). IEEE, 2018.
29. Baylor Denis, et al. Tfx: A tensorflow-based production-scale machine learning platform. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2017.
30. Caffe On Spark github repository, yahoo. Github, 2017. [Online]. Available: https://github.com/yahoo/CaffeOnSpark Accessed on: 2022.
31. TensorFlow On Spark github repository,yahoo. Github, 2018 [Online]. Available: https://github.com/yahoo/TensorFlowOnSpark Accessed on: 2022.
32. Lin J, Ryaboy D. Scaling big data mining infrastructure: the twitter experience. ACM SIGKDD Explorations Newsl. 2013;14(2):6–19.
33. Li M, et al. Scaling distributed machine learning with the parameter server. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014.
34. Liang E, et al. RLlib: Abstractions for distributed reinforcement learning. International Conference on Machine Learning. PMLR, 2018.
35. Accelerating Deep Learning Training with BigDL and Drizzle on Apache Spark, RISELab at UC Berkeley,2017. [Online]. Available: https://rise.cs.berkeley.edu/blog/accelerating-deep-learning-training-with-bigdl-and-drizzle-on-apache-spark/ Accessed on:2022.
36. BigDL: fast, distributed, secure AI for Big Data.intel-analytics, 2020. [Online]. Available: https://bigdl.readthedocs.io/en/latest/ Accessed on:2022.
37. Analytics-Xoo Application Powered By.intel-analytics, 2020. [Online]. Available: https://analytics-zoo.readthedocs.io/en/latest/doc/Application/powered-by.html Accessed on:2022.
38. Bolya D, et al. Yolact: Real-time instance segmentation. Proceedings of the IEEE/CVF international conference on computer vision. 2019.
39. onnxruntime Get Started and Resources,MicrosoftOpenSourceCodes, 2019. [Online]. Available: https://github.com/microsoft/onnxruntime Accessed on:2022.
40. FPN Adapt,Ma-Dan, 2019. [Online]. Available: https://github.com/dbolya/yolact/issues/74#issuecomment-510713725. Accessed on:2022.
41. onnx-simplifier,daquexian, 2019. [Online]. Available: https://github.com/daquexian/onnx-simplifier Accessed on:2022.
42. OpenVINO get startedt,develper, OpenVINO, 2020. [Online]. Available: https://docs.OpenVINO.ai/latest/index.html Accessed on:2022.
43. OpenVINO fp16-or-fp32,develper, OpenVINO, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/developer /articles/technical/should-i-choose-fp16-or-fp32-for-my-deep-learning-model.html Accessed on:2022.
44. Lin T-Y, et al. Microsoft coco: Common objects in context. Computer Vision-ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13. Springer International Publishing, 2014.

45. Databricks User Guide, BigDL Authors. 2022. [Online]. Available: https://bigdl.readthedocs.io/en/latest/doc/UserGuide/databricks.html#create-a-databricks-cluster Accessed on:2022.
46. Increase the number of tasks per stage,Databricks, 2022. [Online]. Available: https://kb.databricks.com/en_US/execution/increase-tasks-per-stage Accessed on:2022.

Publisher's Note

Mohammed Elhmadany received a B.Sc. degree from the Electrical Engineering Department, at Aswan University, Aswan, Egypt, in 2013. His research interests include Distributed Machine/Deep Learning on Big Data clusters. He is currently an Artificial Intelligence Engineer, where his job duties involve designing and implementing machine learning and deep learning models, including generative models to solve complex problems, such as image recognition, natural language processing, or predictive analytics by training AI models using appropriate algorithms and techniques. Optimizing model performance by fine-tuning hyperparameters, applying regularization techniques, or implementing advanced optimization methods and deploying AI models into production environments, or integrating them into existing systems. Ensuring scalability, reliability, and compatibility with the target infrastructure.

Islam Elmadah received a Ph.D. degree from the University of London, U.K., in 2004. He worked as a Professor Assistant at Taibah University, Saudi Arabia, from 2009 to 2011. He is currently working with the Faculty of Engineering, Ain Shams University, Egypt, as an Assistant Professor. He is interested in software engineering development and research, especially requirements analysis using goal models as well as software visualization. He is developing courses related to software engineering, including software formal specification, advanced software engineering, software testing, verification and validation, program analysis, and visualization.

Hossam E. Abdelmunim received the BSc and MS degrees in electrical engineering from Ain Shams University, Egypt, in 1995 and 2000, respectively. He received his Ph.D. degree in Electrical and Computer Engineering from the University of Louisville, Louisville, Kentucky. He joined the Computer Vision and Image Processing Laboratory (CVIP Lab) at the University of Louisville in June 2002, where he has been involved in the applications of image processing and computer vision for medical image analysis. He is now a Full Professor at the Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University in Cairo, Egypt. His current research interests include image modeling, image segmentation, 2D and 3D registration, visualization, and surgical simulation, about which he has authored or co-authored more than 60 technical articles including ICIP, MICCAI, ICCV, CVPR, BMVC, and TIP, TPAMI. He received an excellence in visualization award from Silicon Graphics in 2004, a US National Science Foundation travel grant to attend the International Conference on Computer Vision (ICCV) in 2005, and first place in the University of Louisville Engineering Competition (E-Expo) in 2006. Also, he was awarded the Outstanding ECE Graduate Student Award at the University of Louisville, in 2007.