M8-2

Scalable Deep Learning Algorithms

Topics

- Role of Scalability in DL's Evolution
- Distributed training with TensorFlow
- Distributed training with PyTorch
- Challenges and Opportunities
- Summary

The Role of Scalability in Deep Learning's Evolution

Role of Scalability in DL's Evolution

- **The Evolution of Deep Learning**
  - Deep learning has witnessed exponential growth, driven by advances in computational power and the availability of large datasets.
  - This rapid evolution has led to the development of increasingly complex model architectures designed to solve a broader range of problems more accurately.
- **Challenges Posed by Growth**
  - **Data Volume**: The explosion in data volume demands algorithms capable of learning from vast datasets without prohibitive increases in training time or resources.
  - **Model Complexity**: As models grow in complexity, they require more computational power and memory, posing significant challenges for training and inference.

Role of Scalability in DL's Evolution

- **The Importance of Scalable Algorithms**
  - Scalable deep learning algorithms are essential for harnessing the full potential of large datasets and complex models.
  - These algorithms enable the efficient utilization of computational resources, ensuring that increases in data volume and model complexity do not lead to intractable increases in computational cost.
- **Objectives of Scalable Algorithms**
  - **Improve Training Speed**: By optimizing computational resource use, scalable algorithms significantly reduce the time required to train complex models on large datasets.
  - **Enhance Model Accuracy**: Scalable algorithms allow for the training of more sophisticated models, leading to improvements in accuracy and performance across various tasks.
  - **Boost Efficiency**: These algorithms ensure that deep learning models can be trained and deployed effectively, even as the scale of data and model complexity grows.

Role of Scalability in DL's Evolution

Breakthroughs Enabled by Scalable Algorithms

Scalable algorithms have been pivotal in driving progress across various domains of artificial intelligence, especially in natural language processing (NLP), computer vision, and other areas

Natural Language Processing (NLP):

- **Efficient Parallel Processing**
  - The Transformer architecture (**Attention Is All You Need**: https://arxiv.org/abs/1706.03762, 2017)introduced efficient parallel processing of data, enabling faster training times and improved handling of complex language data compared to traditional sequential models like RNNs and LSTMs.
- **Bidirectional Context Understanding**
  - BERT (https://arxiv.org/abs/1810.04805, 2018), built on the Transformer model, advanced NLP by learning from both directions of text simultaneously. This bidirectionality allowed for a nuanced understanding of language context, enhancing performance on tasks requiring deep linguistic comprehension.
- **Leveraging Massive Datasets**
  - Scalability enabled BERT to learn from enormous datasets, including the entirety of Wikipedia and the BooksCorpus.

“Distributed training with TensorFlow

Distributed training with TensorFlow

MirroredStrategy

Purpose:

To perform synchronous training across multiple GPUs on a single machine.

Function:

Each GPU maintains an identical copy of the model. Computes gradients in parallel. **Synchronization:**

Gradients are averaged across all devices before updating the model, ensuring consistency.

Best used for:

Systems with multiple GPUs seeking to reduce training time without compromising on data integrity.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How MirroredStrategy Works:

Model Replication:
The strategy replicates the model's variables (weights) across all the GPUs.

Data Distribution:
The input data is sliced and distributed evenly across the GPUs.

Gradient Calculation:
Each GPU calculates the gradients for its chunk of the data independently.

Gradient Aggregation:
The gradients from all GPUs are then gathered and averaged.

Update Models:
The averaged gradients are applied to the models on all GPUs simultaneously.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How MirroredStrategy Works:

```
import tensorflow as tf

# Assuming you have a model defined as `model`

# Create a MirroredStrategy
strategy = tf.distribute.MirroredStrategy()

# Wrap your model training step in the strategy scope
with strategy.scope():
    # Compile, fit, or train your model using TensorFlow APIs
    model.compile(...)
    model.fit(...)  # Or custom training loop
```

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

TPUStrategy

- **Purpose:**
  - Optimizes training processes for TensorFlow computations on Google's Tensor Processing Units (TPUs).
- **Function:**
  - Abstracts the complexity of distributed training on TPUs, handles data partitioning to leverage TPU cores.
- **Synchronization:**
  - Coordinates gradient updates across TPU cores effectively.
- **Best used for:**
  - Projects requiring extreme computation power and rapid execution, such as large-scale neural networks.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How TPUStrategy Works:

- **Model Placement**
  - Unlike MirroredStrategy that replicates on GPUs, TPUStrategy places the entire model (or a partitioned version) onto the TPU device.
- **Data Parallelism**
  - The strategy splits the input data across the TPU cores.
  - Each core processes a portion of the data.
- **Gradient Aggregation**
  - All TPU cores calculate gradients for their data shard and then the gradients are synchronized across all cores.
  - This means that all cores update their local model weights in a coordinated fashion after each global batch is processed.
- **Performance Optimization**
  - TPUStrategy includes optimizations specific to TPUs, such as utilizing TPU-specific operations for faster execution.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How TPUStrategy Works:

```
import tensorflow as tf

# Assuming you have a model defined as `model`

# Create a TPUStrategy
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  # Identify TPUs
strategy = tf.distribute.TPUStrategy(resolver)

# Wrap your model training step in the strategy scope
with strategy.scope():
    # Compile, fit, or train your model using TensorFlow APIs
    model.compile(...)
    model.fit(...)  # Or custom training loop
```

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

MultiWorkerMirroredStrategy: Scaling Training Across Multiple Machines

- **Purpose:**
  - Extends MirroredStrategy for synchronous training across multiple workers, each possibly with multiple GPUs.
- **Function:**
  - Similar to MirroredStrategy but operates over the network.
- **Synchronization:**
  - Uses collective communication methods like All-Reduce to update model copies.
- **Best used for:**
  - Large-scale distributed training tasks over multiple machines.

400 data pts / N. nodes

1-200

node 1

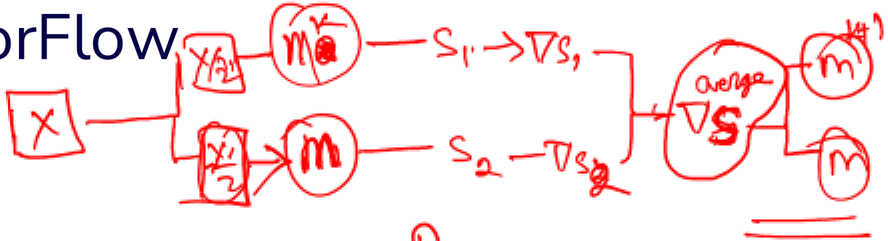
MPI

node 2

201 — 400

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow



How MultiWorkerMirroredStrategy Works:

- **Cluster Setup:**
  - You have a cluster of machines (nodes), each node potentially equipped with multiple GPUs.
- **MirroredStrategy on Each Worker:**
  - MultiWorkerMirroredStrategy creates a separate MirroredStrategy instance on each node.
  - This means each node replicates the model across its own available GPUs.

1 - 500

501 - 10,000

- **Data Sharding:**
  - The training dataset is divided (sharded) into smaller chunks and distributed to each node.
  - This ensures parallelism across nodes.
- **Synchronized Training:**
  - Each node's MirroredStrategy processes its assigned data shard independently.
  - Gradients are calculated locally on each node.
- **Gradient Exchange:**
  - After processing, gradients are exchanged and aggregated efficiently between all nodes using a technique called "all-reduce."
- **Model Update:**
  - The aggregated gradients are applied to update the model replicas on all GPUs of

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How MultiWorkerMirroredStrategy Works:

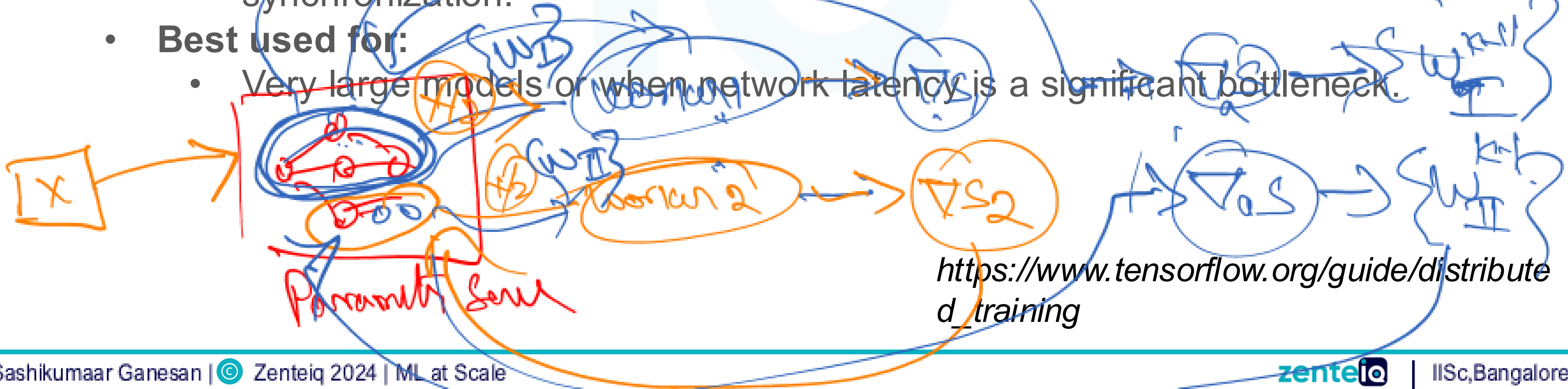
```
import tensorflow as tf

# Assuming you have a model defined as `model`

# Define cluster configuration (replace with actual cluster details)
cluster_resolver = tf.distribute.cluster_resolver.SlurmClusterResolver(...)

# Create a MultiWorkerMirroredStrategy
strategy = tf.distribute.MultiWorkerMirroredStrategy(cluster_resolver)

# Wrap your model training step in the strategy scope
with strategy.scope():
    # Compile, fit, or train your model using TensorFlow APIs
    model.compile(...)
    model.fit(...)  # Or custom training loop
```

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

ParameterServerStrategy (experimental strategy)

- **Purpose:**
  - For asynchronous training with parameter servers.
- **Function:**
  - Model parameters are stored on parameter servers, workers update these asynchronously.
- **Synchronization:**
  - Minimizes network bottleneck, as workers do not need to wait for synchronization.
- **Best used for:**
  - Very large models or when network latency is a significant bottleneck.



https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How ParameterServerStrategy Works:

- **Cluster Roles**: The training cluster consists of three types of nodes:
    - **Worker Nodes**:
        - These nodes handle the actual computations for training the model.
        - Each worker can have CPUs or GPUs.
    - **Parameter Servers**:
        - These dedicated servers store and manage the model variables (weights and biases).
        - They don't perform computations directly.
    - **Coordinator: (Optional)**
        - A central coordinator node can be used to manage cluster resources and dispatch tasks to workers.
- **Variable Distribution:**
    - Model variables are sharded (divided) and distributed across multiple parameter servers.
    - This allows for efficient storage and handling of large models.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How ParameterServerStrategy Works:

- **Worker-Server Interaction:**
  - During training, each worker node:
    - Downloads a copy of the relevant variable shards from the parameter servers.
    - Performs computations on its assigned data batch using the downloaded variables.
    - Calculates gradients based on the computation results.
    - Sends the calculated gradients back to the parameter servers.
- **Parameter Update:**
  - Parameter servers receive gradients from all workers, aggregate them (e.g., averaging), and update their corresponding variable shards.
  - This aggregation step is typically **synchronous**. This ensures that all workers contribute to the same update, and the model state remains consistent across the cluster.
- **Potential Alternatives for Asynchronous Parameter Updates**:
  - Stale gradients:
    - Workers might use slightly outdated parameter values when calculating gradients.
    - This can improve efficiency but might affect convergence or introduce instability.

Distributed training with TensorFlow

How ParameterServerStrategy Works:

```python
import tensorflow as tf

# Assuming you have a model defined as `model`

# Define cluster configuration (replace with actual cluster details)
cluster_resolver = tf.distribute.cluster_resolver.SlurmClusterResolver(...)

# Create a ParameterServerStrategy
strategy = tf.distribute.experimental.ParameterServerStrategy(cluster_resolver)

# Wrap your model training step in the strategy scope
with strategy.scope():
    # Compile, fit, or train your model using TensorFlow APIs
    model.compile(...)
    model.fit(...)  # Or custom training loop
```

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

CentralStorageStrategy (experimental strategy)

- **Purpose:**
  - To allow computation to be offloaded from one device to others, while centralizing the model's parameters.
- **Function:**
  - One device holds the parameters, and others assist in computation.
- **Synchronization:**
  - Occurs on the central device, reducing the need for complex communication protocols.
- **Best used for:**
  - Moderate-sized models where device-to-device communication is efficient..

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

How CentralStorageStrategy Works:

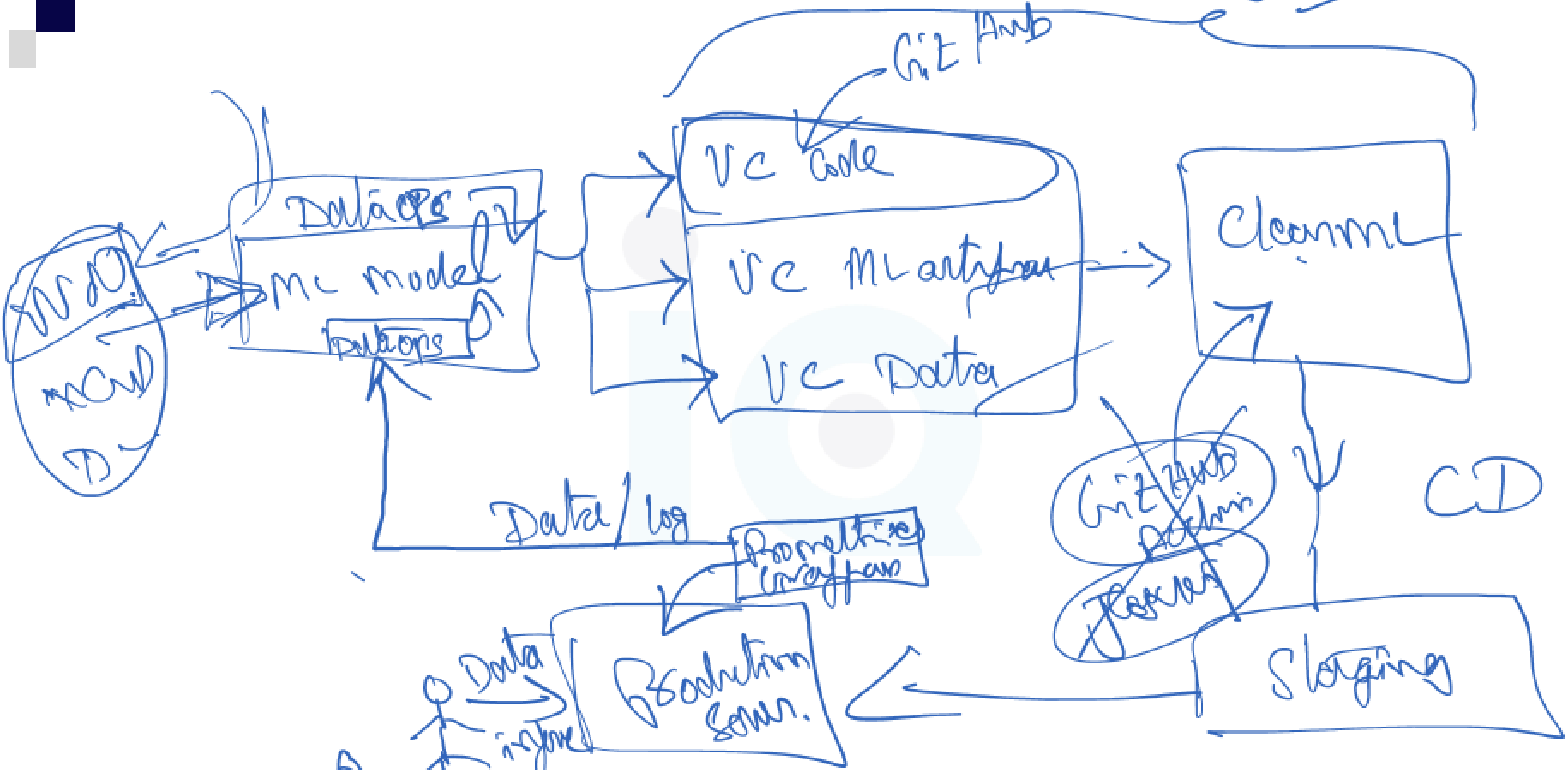
- **Central Storage:**
  - Similar to ParameterServerStrategy, model variables are likely stored centrally on a designated node (CPU or GPU).
- **Worker Computations:**
  - Worker nodes perform computations on their assigned data batches, presumably downloading or accessing the variables from the central storage as needed.
- **Gradient Communication:**
  - Workers calculate gradients and send them back to the central storage for aggregation and model updates.

https://www.tensorflow.org/guide/distributed_training

Distributed training with TensorFlow

Feature	MultiWorkerMirroredStrategy	ParameterServerStrategy (PS)	CentralStorageStrategy (CS)
Model Replication	Each worker replicates model (uses MirroredStrategy)	Variables distributed across parameter servers	Variables stored centrally (potentially)
Worker Roles	All workers perform computations	Workers compute, parameter servers store variables	Workers compute, central storage for variables
Communication Pattern	All-reduce gradients between workers	Workers communicate with parameter servers	Workers communicate with central storage
Scalability	Good for large datasets, decent for large models	Potentially better for very large models	Potential for scalability, details unclear
Flexibility	Workers and GPUs can be scaled together	Workers and parameter servers can be scaled independently	Limited information on flexibility
Complexity	Relatively simpler to set up	More complex setup and configuration	Experimental, complexity unclear
Synchronous/Asynchronous	Synchronous updates	Typically synchronous parameter updates, potential for asynchronous exploration	Information limited, might be synchronous
Fault Tolerance	Limited fault tolerance	Potentially better due to separate storage	Information limited
Status	Stable	Experimental	Experimental

Summary of this Module



https://www.tensorflow.org/guide/distributed_training

Summary of this Module

https://www.tensorflow.org/guide/distributed_training