HML401

Accelerate NLP training with Amazon SageMaker

Aditya Bindal

Sr. Product Manager
Amazon Web Services

Agenda

Using Hugging Face in Amazon SageMaker

Challenges in training NLP models

Main features of SageMaker's distributed training libraries

Using data parallelism in SageMaker with PyTorch

Using model parallelism in SageMaker with TensorFlow

Using Hugging Face in Amazon SageMaker

Natural language processing (NLP)

NLP serves many human-friendly functions and its usage can have huge impact on the business of a company

LANGUAGE TRANSLATION

NLP has made machine translation so fast and accurate that translation becomes accessible to many applications

SPELL CHECK & GRAMMAR CHECK

This ensures a higher data quality from the very source of data, reducing complex and tedious data cleansing operations

SMART ASSISTANTS

Offering human-friendly interactions using voice to a broad category of devices

NATURAL QUERY & SEARCH

NLP allows powerful query and search capabilities using natural language questions and human-oriented interfaces

TEXT ANALYTICS

Extracting topics or sentiments from unstructured text to serve analytics and data-driven business decisions

Customers building NLP solutions on AWS

A strong partnership to make NLP easy and accessible for all

Hugging Face

Hugging Face is the most popular open source company providing state of the art NLP technology

AWS

SageMaker offers high performance resources to train and use NLP Models

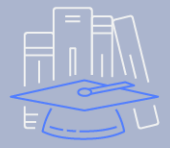
What are the Hugging Face libraries?

Open-source
`Datasets`, `Tokenizers` and `Transformers`

Popular
58k+ GitHub stars (May 2021), 1M+ downloads per month

Intuitive
NLP-specific Python frontends based on PyTorch or TensorFlow

State of the art
Transformer-based models are state of the art, enable transfer-learning and scale



Comprehensive
Model zoo with 7000+ model architectures, 160+ languages

Introducing a new Hugging Face experience in Amazon SageMaker

A new Deep Learning Container (DLC) developed with Hugging Face, includes `Datasets`, `Transformers` and `Tokenizers`, along with SageMaker integrations such as Debugger and SageMaker's distributed training libraries

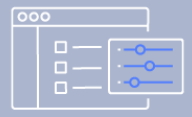
A Hugging Face Estimator in the SageMaker SDK to launch training and fine-tuning jobs with Hugging Face models on SageMaker's fully managed platform

An example gallery to find readily usable high-quality samples of Hugging Face scripts on Amazon SageMaker

Support Maintained and supported by AWS

What are the benefits of running Hugging Face in Amazon SageMaker?

Cost-effective – SageMaker optimizes performance and offers managed Spot training to reduce costs



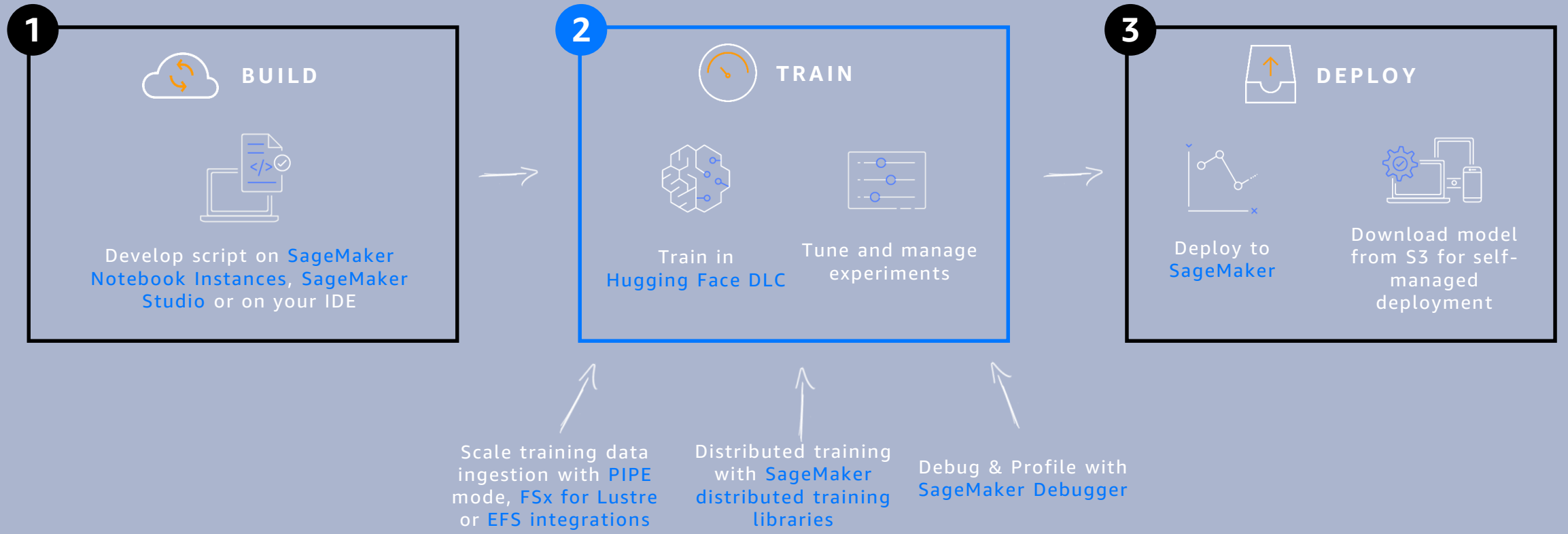
MLOps-ready – Includes automated metadata persistance & search in the SageMaker metastore, log extraction to Amazon CloudWatch, monitoring with SageMaker Debugger & Profiler and experiment management

Scalable – Able to run on clusters of GPUs, with efficient data-parallel and model-parallel distribution provided by Amazon SageMaker; ability to launch several concurrent jobs at the same time with the async mode of the API

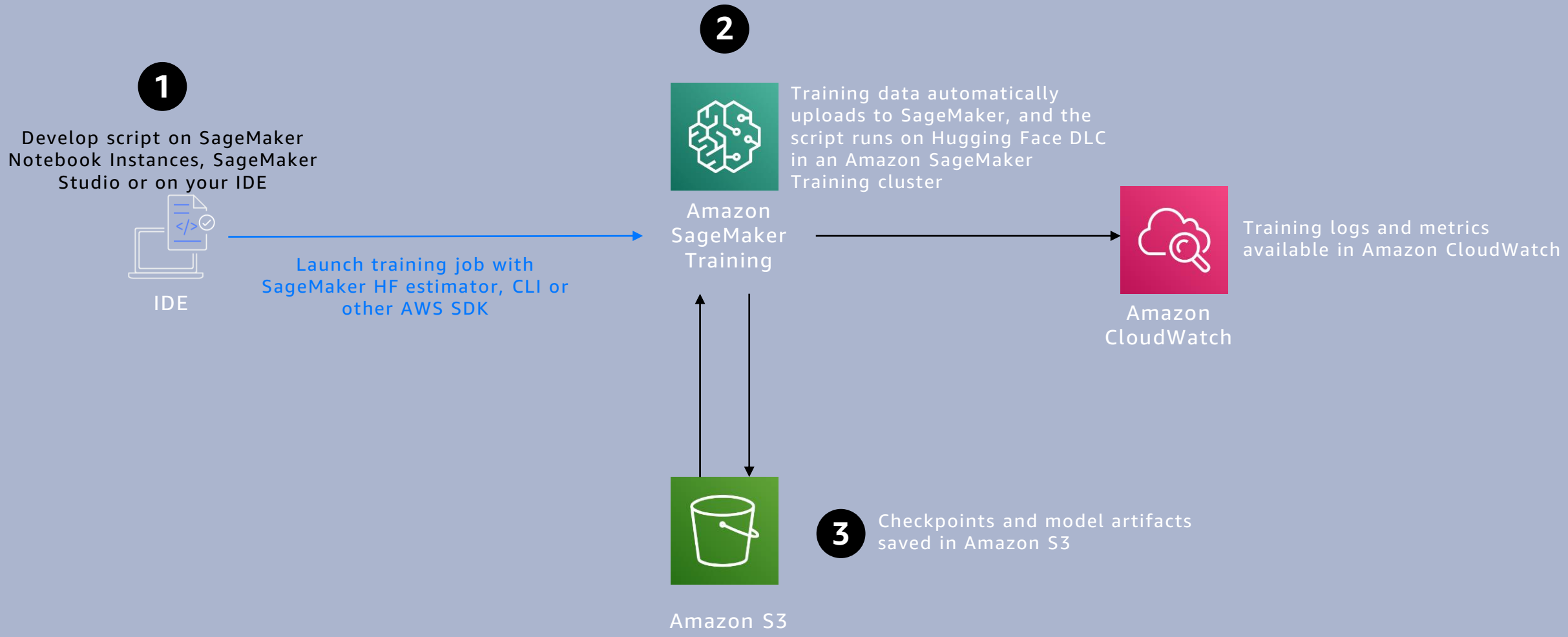
Secure – High-bar on security, with available mechanisms including encryption at rest and in transit, VPC connectivity and fine-grained IAM permissions

aws

Integrated workflow with Amazon SageMaker



1-click managed training



Example with PyTorch

```
from sagemaker.huggingface import HuggingFace

# hyperparameters, which are passed into the training job
hyperparameters={'epochs': 1,
                 'train_batch_size': 32,
                 'model_name':'distilbert-base-uncased'
                 }
```

```
huggingface_estimator = HuggingFace(entry_point='train.py',
                            source_dir='./scripts',
                            instance_type='ml.p3.2xlarge',
                            instance_count=1,
                            role=role,
                            transformers_version='4.4',
                            pytorch_version='1.6',
                            py_version='py36',
                            hyperparameters = hyperparameters)
```

```
# starting the train job with our uploaded datasets as input
huggingface_estimator.fit({'train': training_input_path, 'test': test_input_path})
```

Example with TensorFlow

```
from sagemaker.huggingface import HuggingFace

# hyperparameters, which are passed into the training job
hyperparameters={'epochs': 1,
                 'train_batch_size': 16,
                 'model_name':'distilbert-base-uncased'
                }
```

```
huggingface_estimator = HuggingFace(entry_point='train.py',
                            source_dir='./scripts',
                            instance_type='ml.p3.2xlarge',
                            instance_count=1,
                            role=role,
                            transformers_version='4.4',
                            tensorflow_version='2.4',
                            py_version='py37',
                            hyperparameters = hyperparameters)
```

```
huggingface_estimator.fit()
```

More examples

I want to train a text classification model using Hugging Face in SageMaker with PyTorch – for a sample Jupyter Notebook, see the PyTorch Getting Started Demo.

I want to train a text classification model using Hugging Face in SageMaker with TensorFlow – for a sample Jupyter Notebook, see the TensorFlow Getting Started example.

I want to run distributed training with data parallelism using Hugging Face and SageMaker – for a sample Jupyter Notebook, see the Distributed Training example.

I want to run distributed training with model parallelism using Hugging Face and SageMaker – for a sample Jupyter Notebook, see the Model Parallelism example.

I want to use a spot instance to train a model using Hugging Face in SageMaker.For a sample Jupyter Notebook, see the Spot Instances example.

I want to capture custom metrics and use SageMaker Checkpointing when training a text classification model using Hugging Face in SageMaker – for a sample Jupyter Notebook, see the Training with Custom Metrics example.

I want to train a distributed question-answering TensorFlow model using Hugging Face in SageMaker. For a sample Jupyter Notebook, see the Distributed TensorFlow Training example.

Customers using Hugging Face on SageMaker

Quantum Health

Quantum Health is on a mission to make healthcare navigation smarter, simpler, and more cost-effective for everyone. They use Hugging Face and Amazon SageMaker for use cases like text classification, text summarization, and Q&A to help agents and members.

> We are excited about the integration of Hugging Face Transformers into an Amazon Deep Learning Container to make use of SageMaker distributed training to shorten the training time for our larger datasets.
>
> Jorge Grisman, NLP Data Scientist, Quantum Health

Customers using Hugging Face on SageMaker

Kustomer

Kustomer is a customer service CRM platform for managing high support volume with ease. They use machine learning models to help customers contextualize conversations, remove time-consuming tasks, and deflect repetitive questions.

> We use Hugging Face and Amazon SageMaker extensively, and we are excited about the integration of Hugging Face Transformers into an Amazon Deep Learning Container since it will simplify the way we fine tune machine learning models for text classification and semantic search.
>
> Victor Peinado, ML Software Engineering Manager, Kustomer

Customers using Hugging Face on SageMaker

Musixmatch

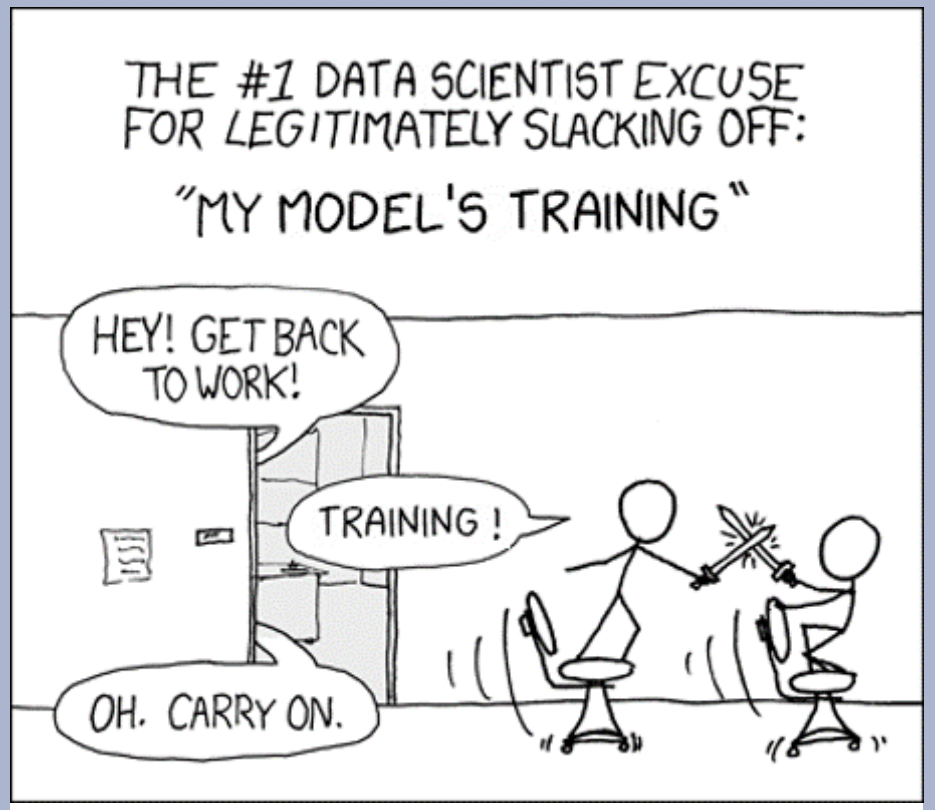
Musixmatch is an Italian music data company and platform for users to search and share song lyrics with translations. It is the largest platform of this kind in the world having 73 million users and 14 million song lyrics

> As the world's leading company in music metadata, Musixmatch has been using cutting edge technology like Hugging Face Transformers and Amazon SageMaker for several use cases, such as music and language modeling. Looking into the future, our team is super excited about the integration of Hugging Face Transformers into SageMaker, which will make training even easier.
>
> Loreto Parisi, Engineering Director, Musixmatch

aws

Challenges in training NLP models

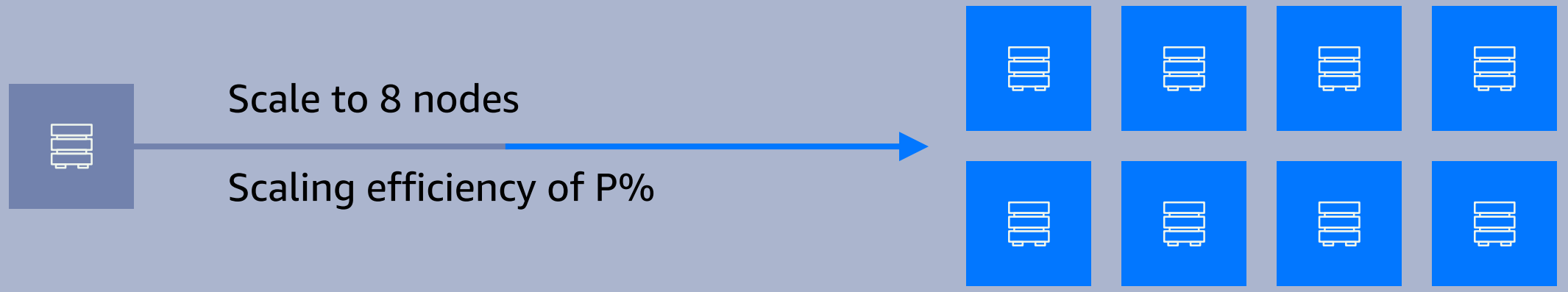
Training time can slow down development



Source: https://xkcd.com/303/

Model	RoBERTa
Dataset	300+ GB
Cluster	64 p3dn.24xl
Training time	Several days

Scaling to more GPUs can reduce training time



1 Node

Time-to-train: T hours

Cost: T • price

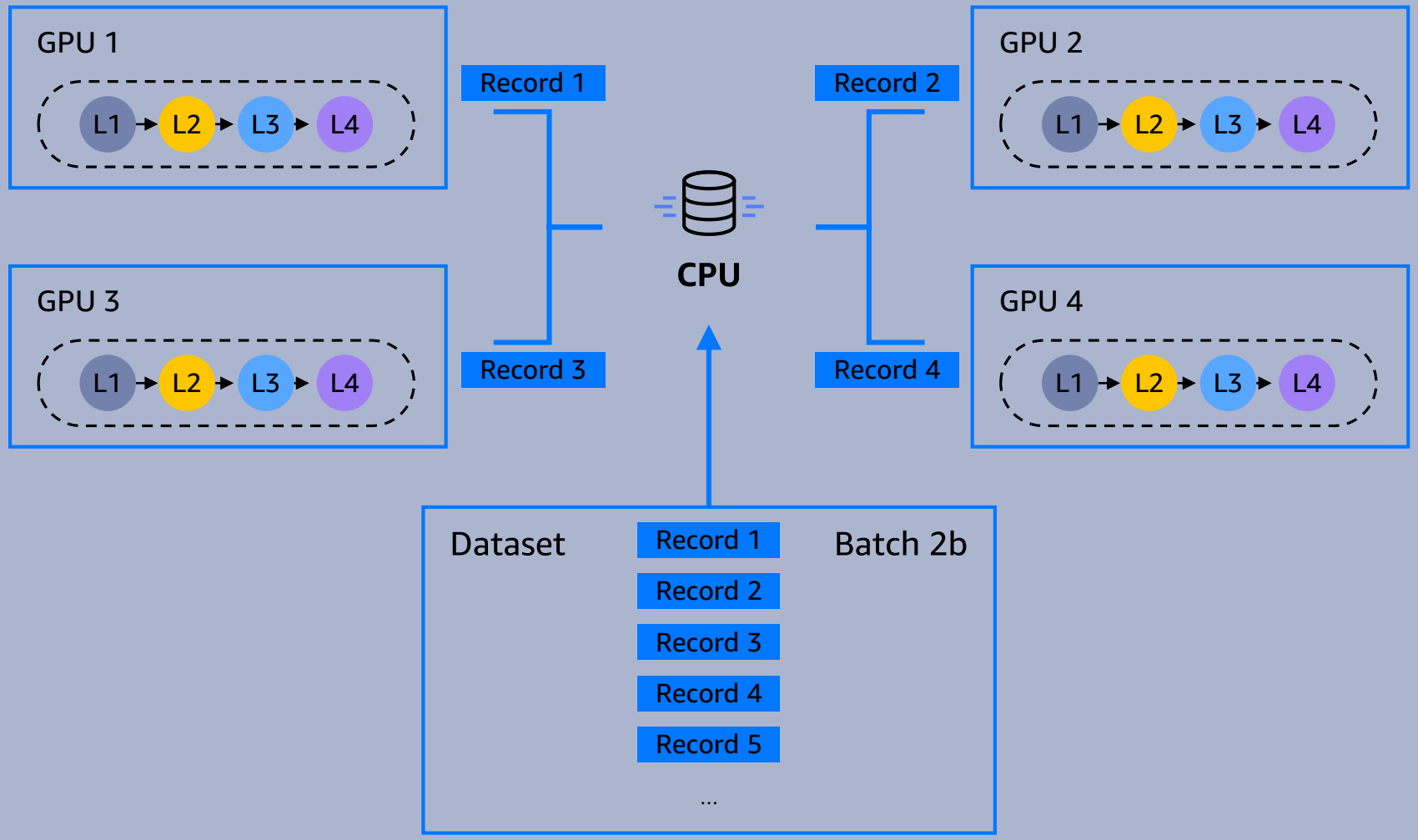
8 nodes

Time-to-train: (T/8)/P% hours

Cost: (T/8)/P% • price

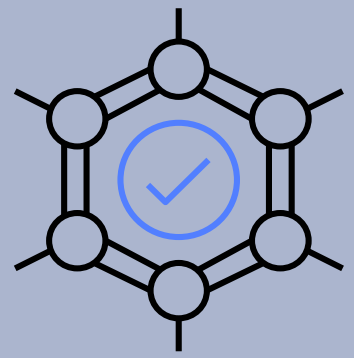
Example: with 90% scaling efficiency from 1 to 8 nodes, training time decreases by 86% and cost increases by 11%

Quick recap of data parallelism



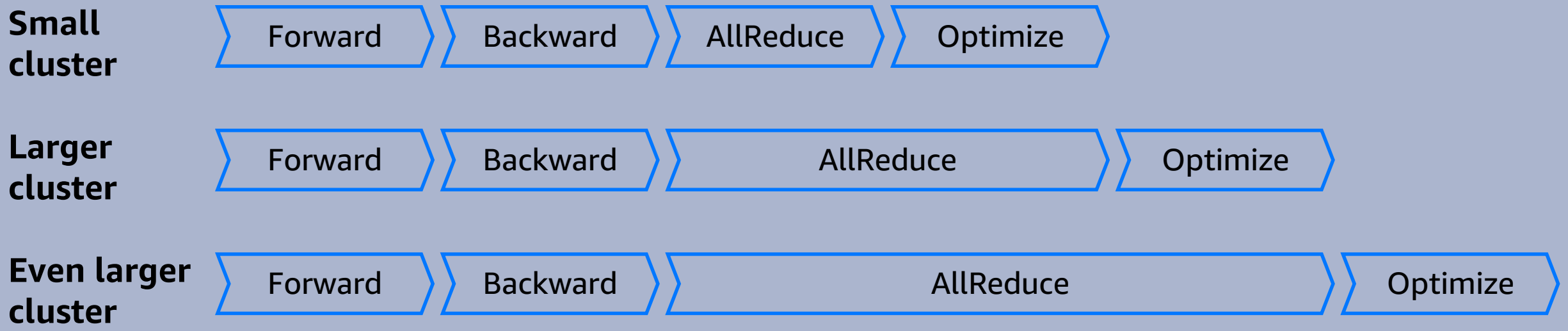
Quick recap of gradient synchronization

Parameter server
(e.g., TensorFlow ParameterServerStrategy)

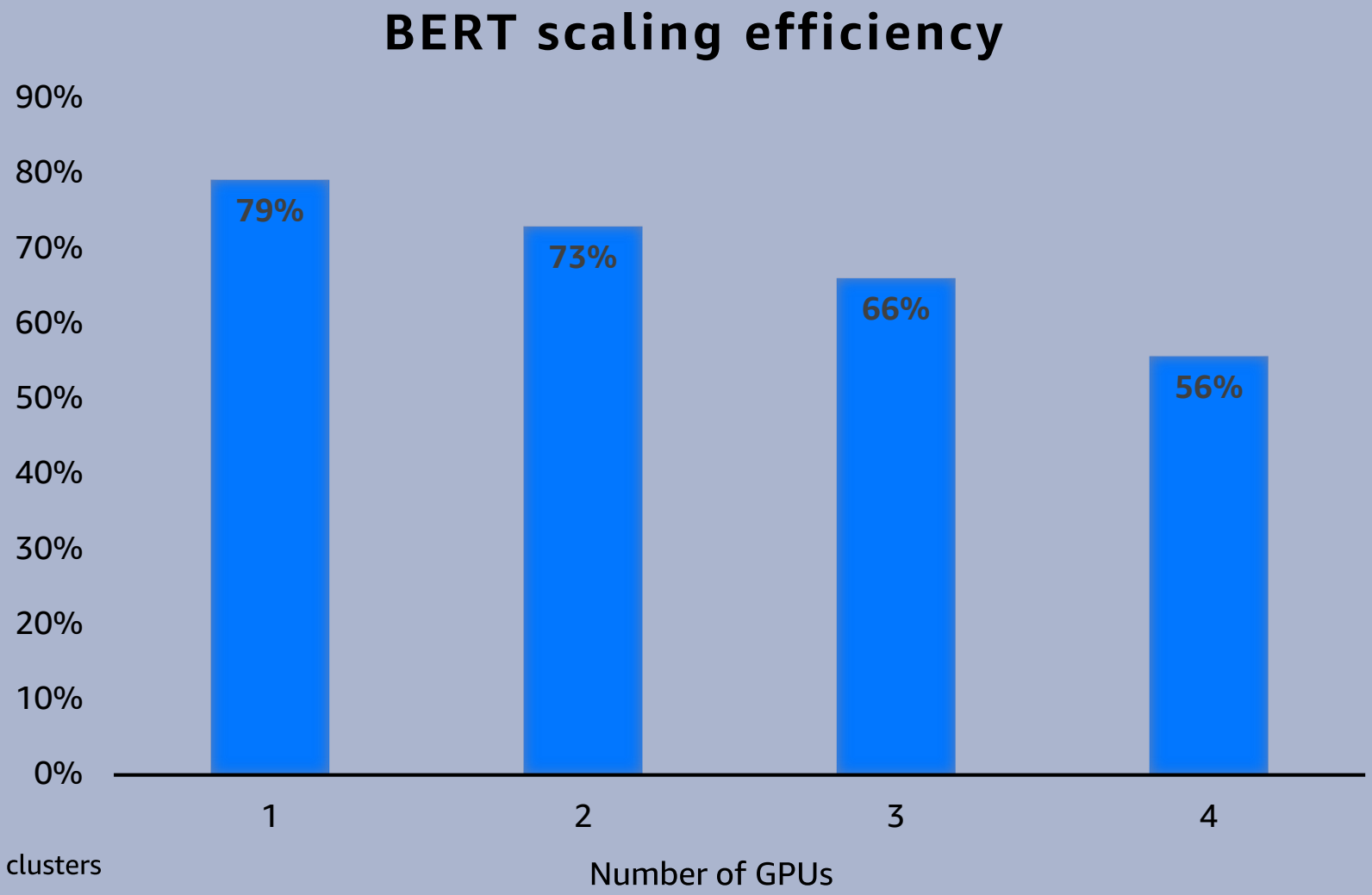


MPI AllReduce
(e.g., Horovod and PyTorch DistributedDataParallel)

Network bottlenecks in distributed training

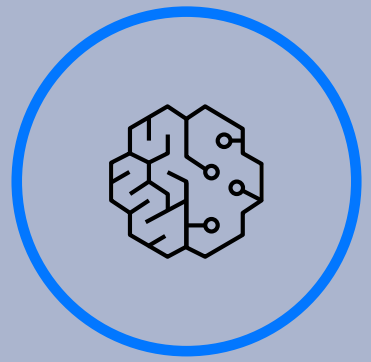


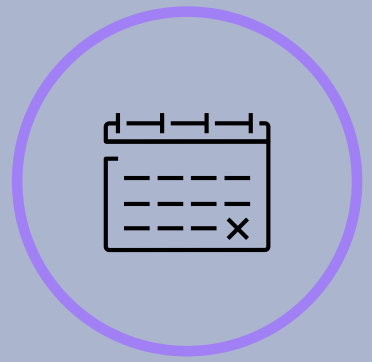
Poor scaling efficiency with prior solutions

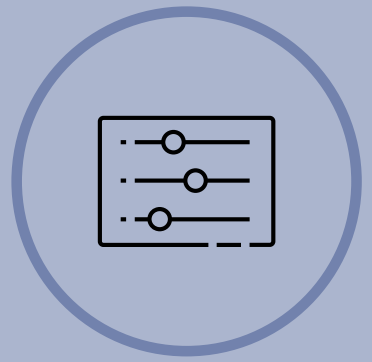


BERT large on p3dn.24xlarge clusters

Deep learning models are growing in size



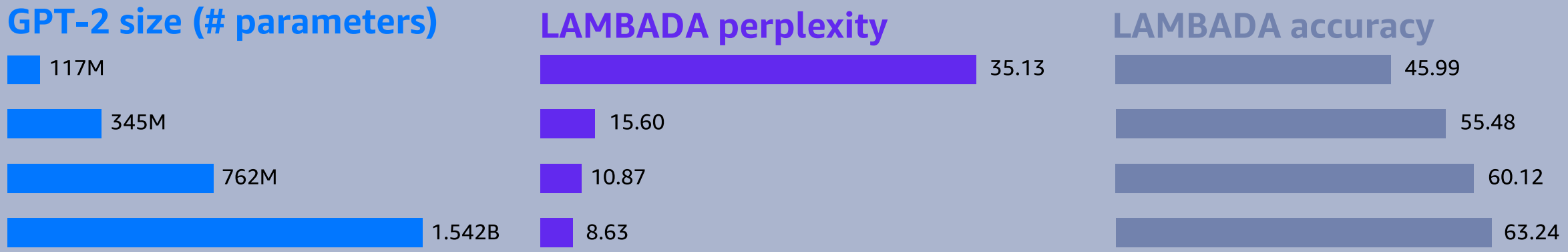




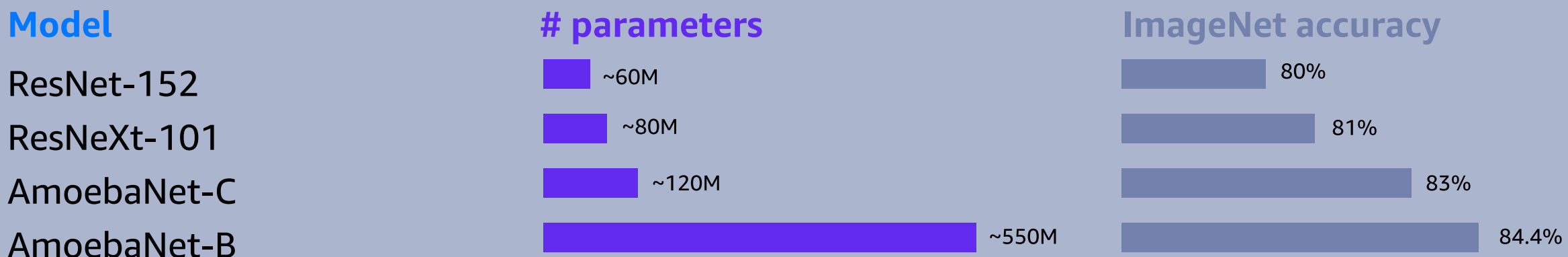
MODEL	RELEASED	# PARAMETERS
BERT	October 2018	340M
GPT-2	February 2019	1.5B
T5	October 2019	11B
GPT-3	July 2020	175B

Larger models have higher prediction accuracy

Natural language processing (NLP)

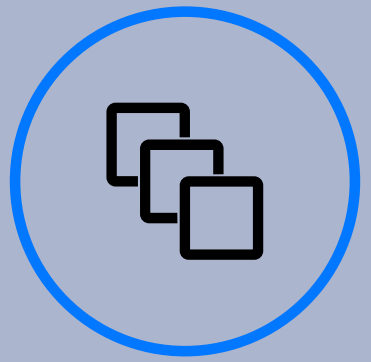
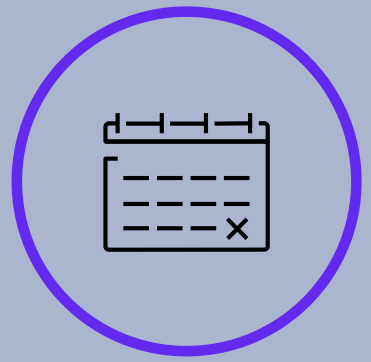
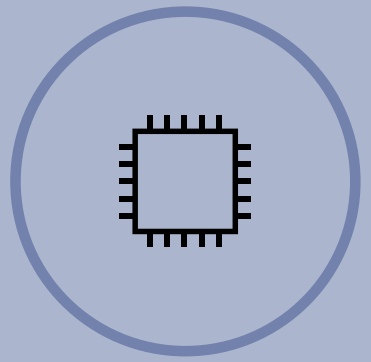


Computer vision



Hardware trends

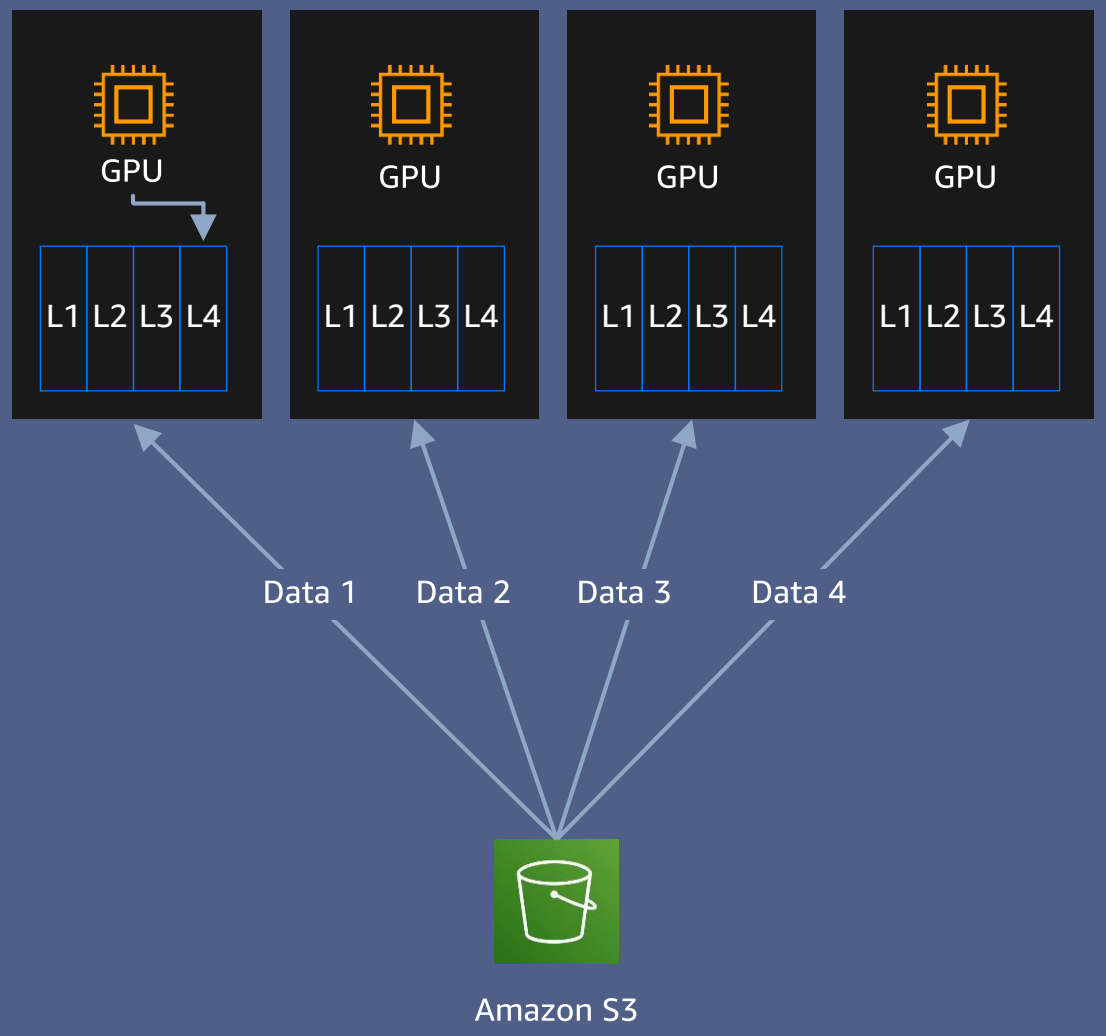
HARDWARE CAPACITY GROWS TOO, BUT NOT AS FAST


INSTANCE TYPE	AVAILABLE	GPU	GPU MEMORY
P2.16xlarge	September 2016	NVIDIA K80	12 GB
P3.16xlarge	October 2017	NVIDIA V100	16 GB
P3dn.24xlarge	December 2018	NVIDIA V100	32 GB
P4d.24xlarge	November 2020	NVIDIA A100	40 GB

Memory bottlenecks in distributed training

- Model size limited by memory of a single GPU – large models cause OOM errors
- Model replicated across all GPUs – wasteful when model is large

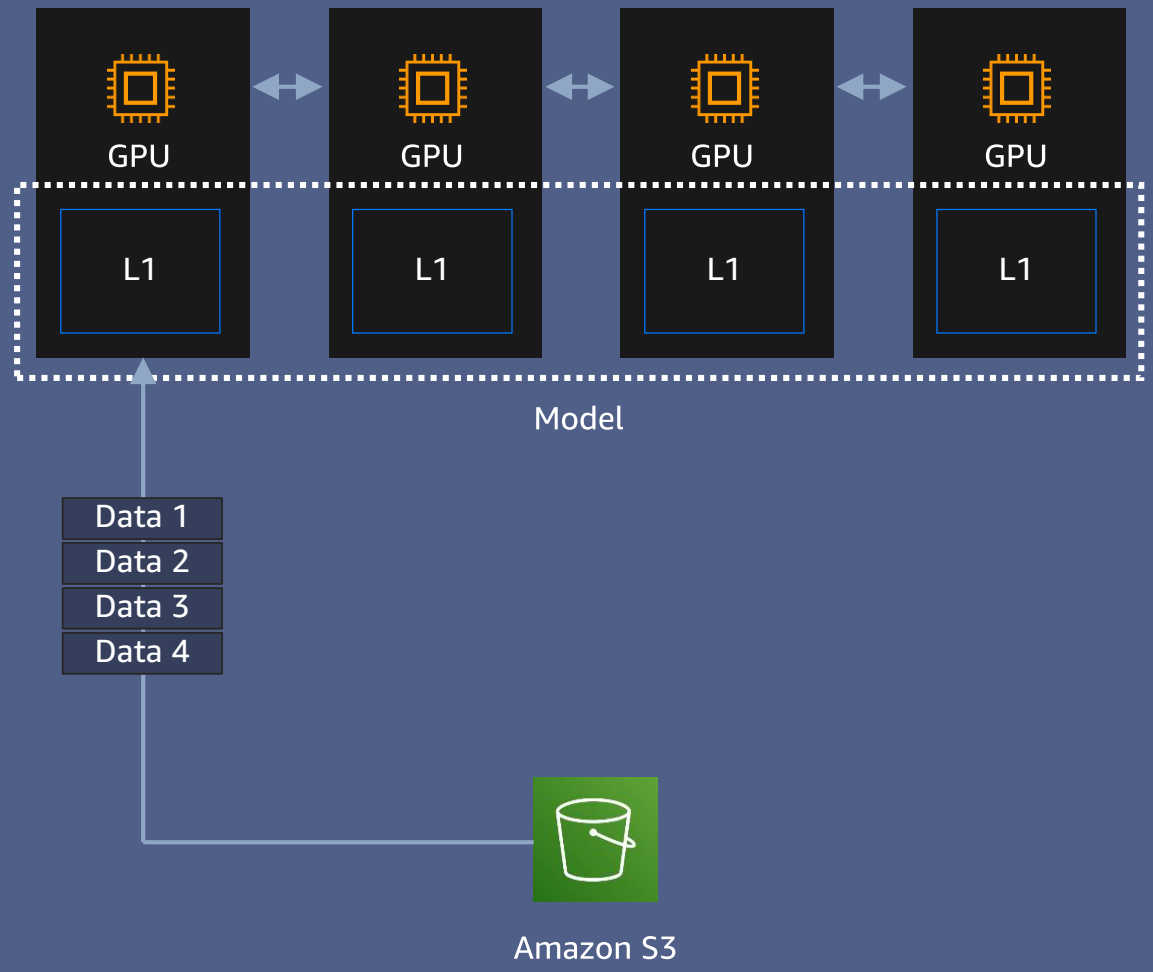


Solution

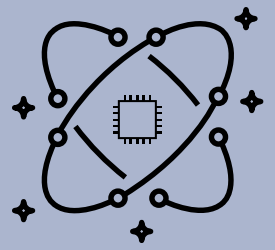
MODEL PARALLELISM

Single model replica partitioned across multiple GPUs

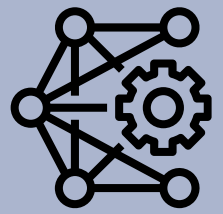
- Combine memory of all GPUs
- No model replication – save additional memory
- Devices communicate during forward and backward pass



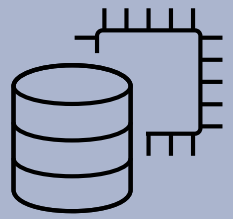
But model parallelism is hard



Maximizing GPU utilization



Finding efficient partitions

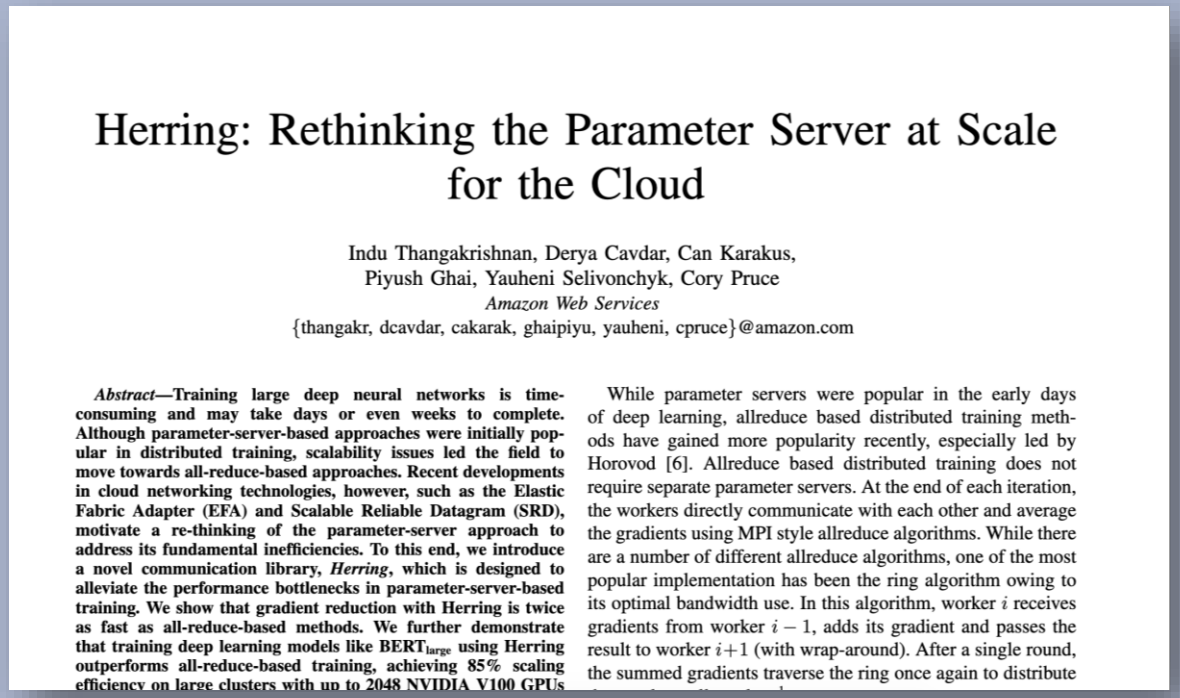


Infrastructure management

Customized training code

Main features of SageMaker's distributed training libraries

Record training times on AWS with NVIDIA GPUs



Herring: Rethinking the Parameter Server at Scale for the Cloud

Indu Thangakrishnan, Derya Cavdar, Can Karakus,
Piyush Ghai, Yauheni Selivonchyk, Cory Pruce
Amazon Web Services
{thangakr, dcavdar, cakarak, ghaipiyu, yauheni, cpruce}@amazon.com

***Abstract*—Training large deep neural networks is time-consuming and may take days or even weeks to complete. Although parameter-server-based approaches were initially popular in distributed training, scalability issues led the field to move towards all-reduce-based approaches. Recent developments in cloud networking technologies, however, such as the Elastic Fabric Adapter (EFA) and Scalable Reliable Datagram (SRD), motivate a re-thinking of the parameter-server approach to address its fundamental inefficiencies. To this end, we introduce a novel communication library, *Herring*, which is designed to alleviate the performance bottlenecks in parameter-server-based training. We show that gradient reduction with Herring is twice as fast as all-reduce-based methods. We further demonstrate that training deep learning models like $\text{BERT}_{\text{large}}$ using Herring outperforms all-reduce-based training, achieving 85% scaling efficiency on large clusters with up to 2048 NVIDIA V100 GPUs**

While parameter servers were popular in the early days of deep learning, allreduce based distributed training methods have gained more popularity recently, especially led by Horovod [6]. Allreduce based distributed training does not require separate parameter servers. At the end of each iteration, the workers directly communicate with each other and average the gradients using MPI style allreduce algorithms. While there are a number of different allreduce algorithms, one of the most popular implementation has been the ring algorithm owing to its optimal bandwidth use. In this algorithm, worker i receives gradients from worker $i-1$, adds its gradient and passes the result to worker $i+1$ (with wrap-around). After a single round, the summed gradients traverse the ring once again to distribute

2018	Fastest Resnet
2019	Fastest BERT and Mask-RCNN
2020	Fastest T5-3B and Mask-RCNN

Presented at SC 20

https://www.amazon.science/publications/herring-rethinking-the-parameter-server-at-scale-for-the-cloud

aws

Model parallelism

Automated and efficient model partitioning

Data parallelism

Reduced training time

Minimal code change

Distributed training on Amazon SageMaker

https://aws.amazon.com/sagemaker/distributed-training/

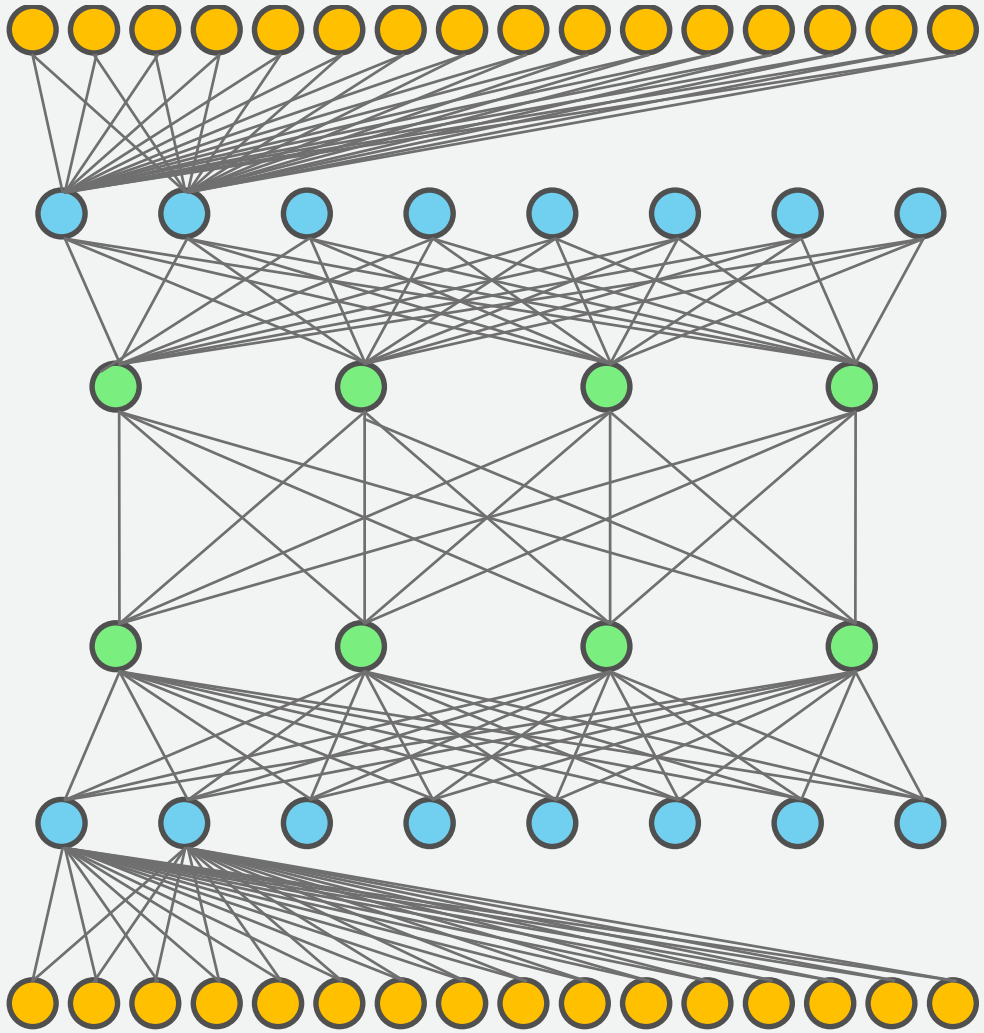
Optimized for AWS

Efficient pipelining

Support for popular ML framework APIs

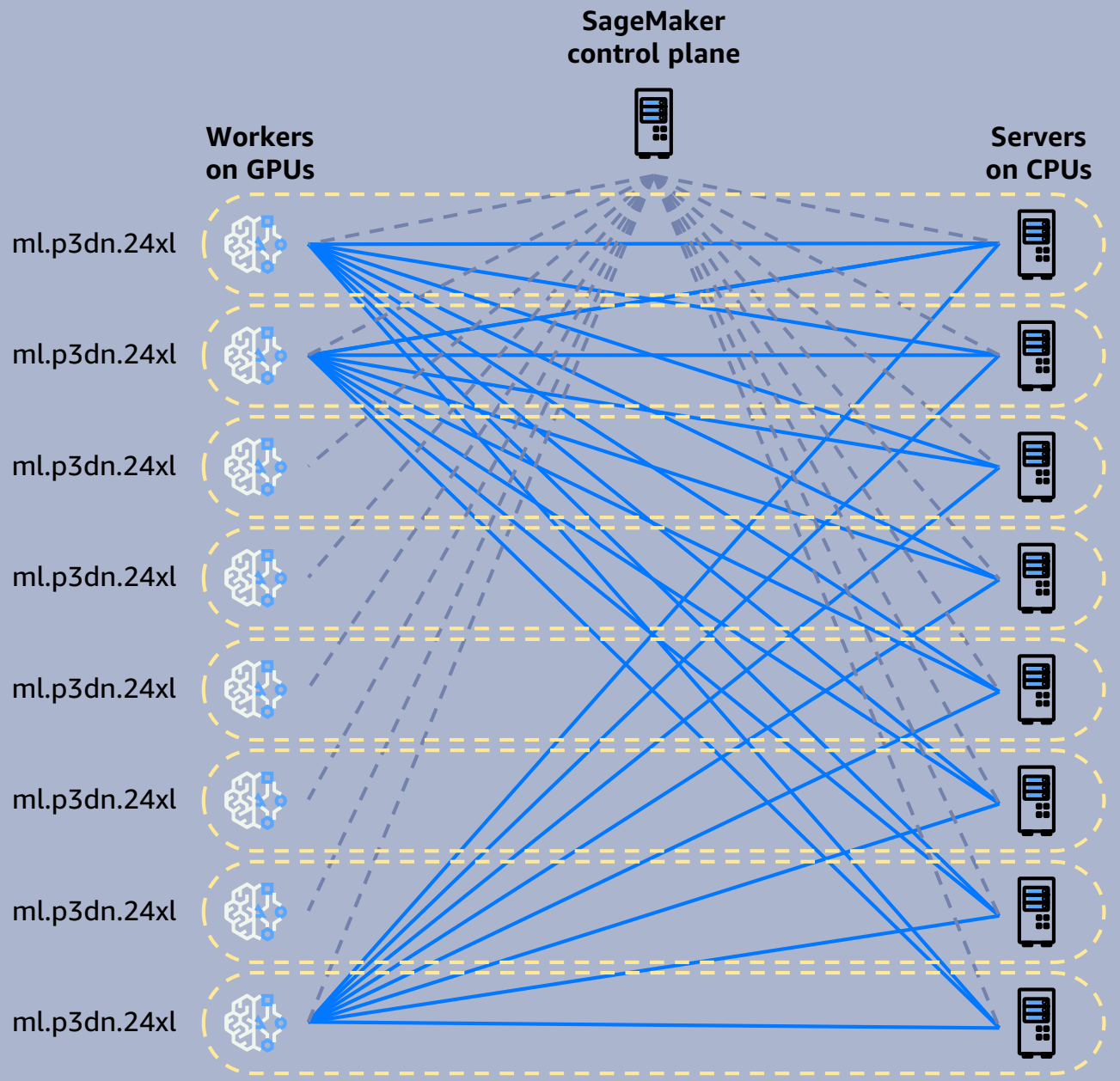
DataParallel in SageMaker

- Library for distributed training of deep learning models in TensorFlow and PyTorch
- Accelerates training for network-bound workloads
- Built and optimized for AWS network topology and hardware
- 20%–40% faster and cheaper; best performance on AWS

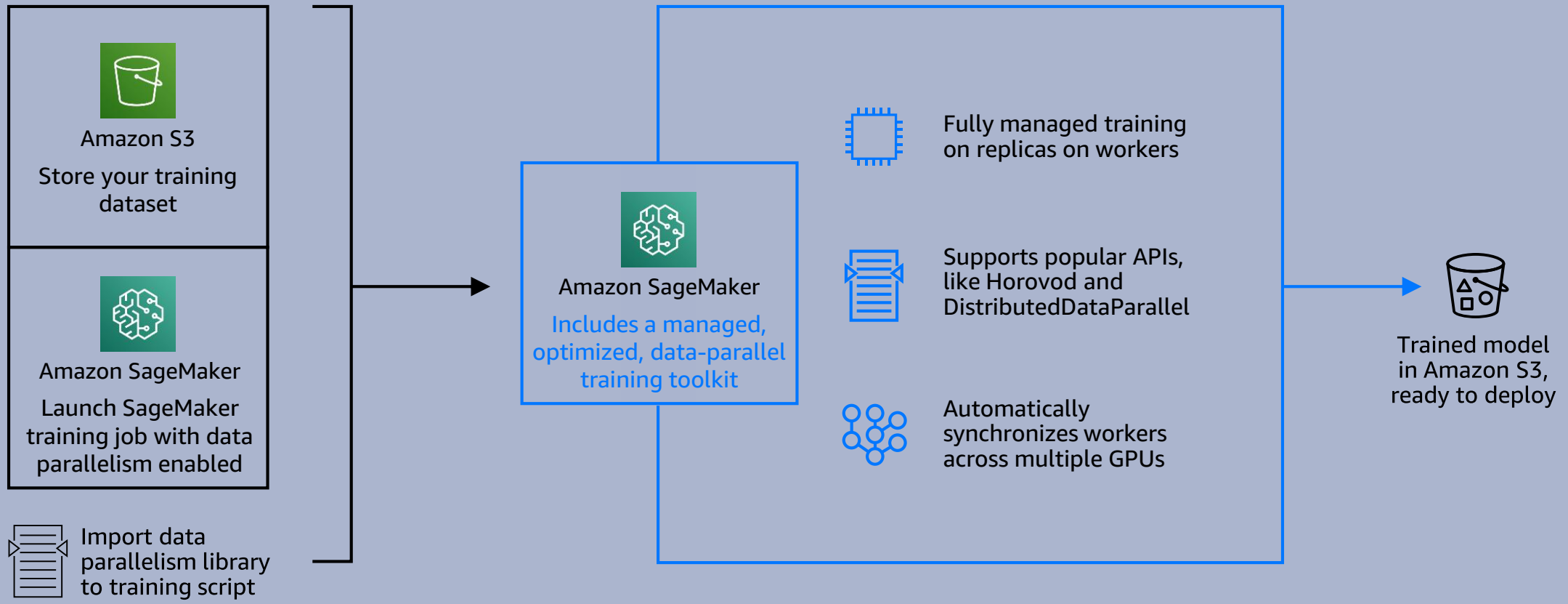


aws

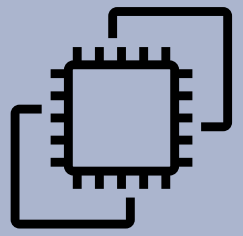
DataParallel under the hood



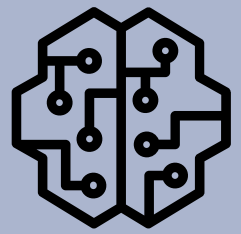
DataParallel architecture



Model parallelism on Amazon SageMaker



Efficient pipelined training



Automated model partitioning

Managed SageMaker training

Tight framework integration

Automated partitioning

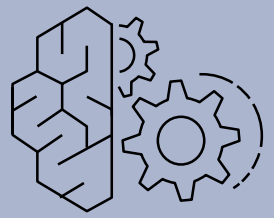
ANALYZED MODEL

Graph structure

Sizes of trainable weights

Sizes of exchanged tensors (using SageMaker Debugger)

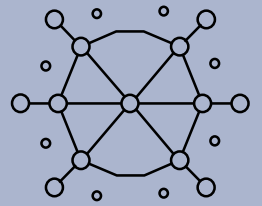
RUN GRAPH PARTITIONING ALGORTIHM



Balance stored weights and activations

Minimize communication

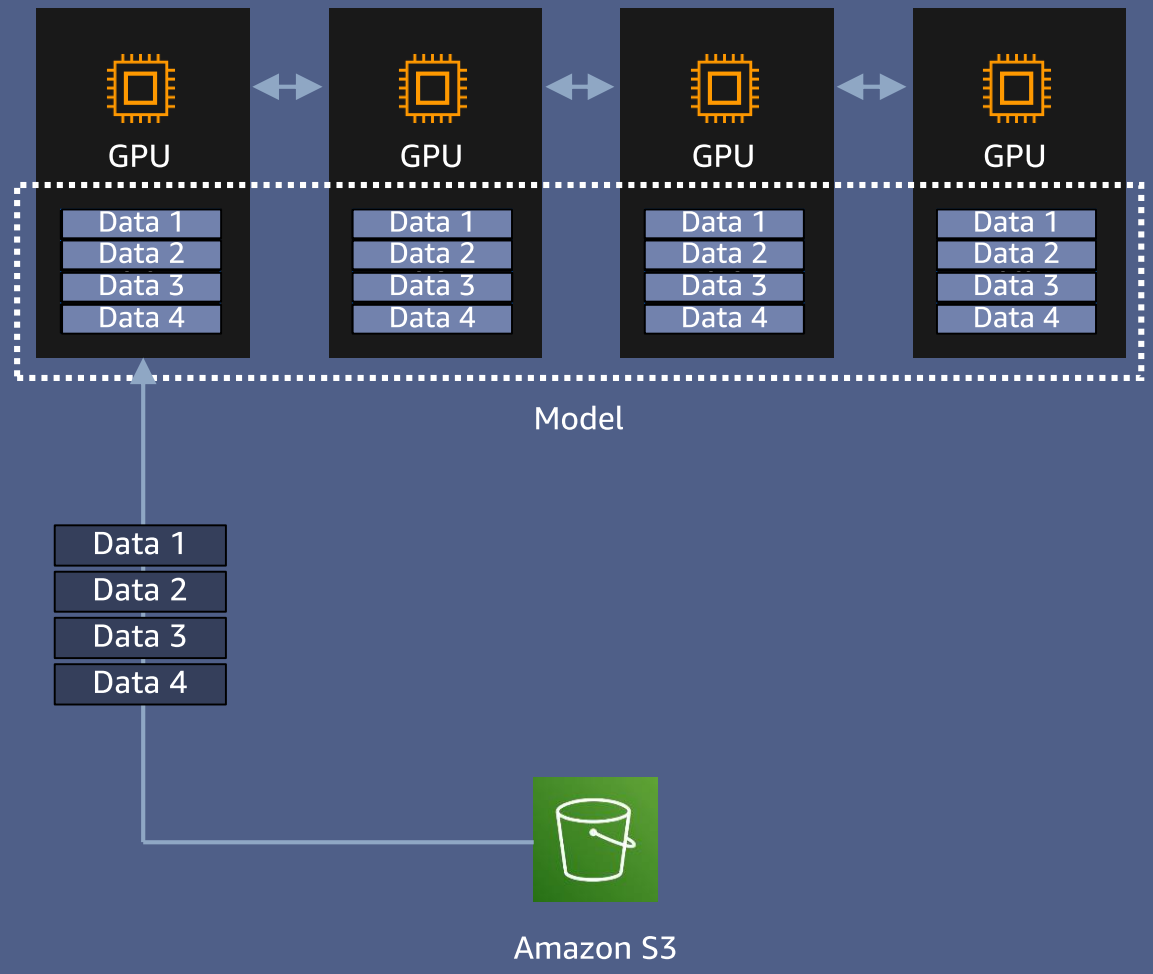
PLACE PARTITIONS ON DEVICES



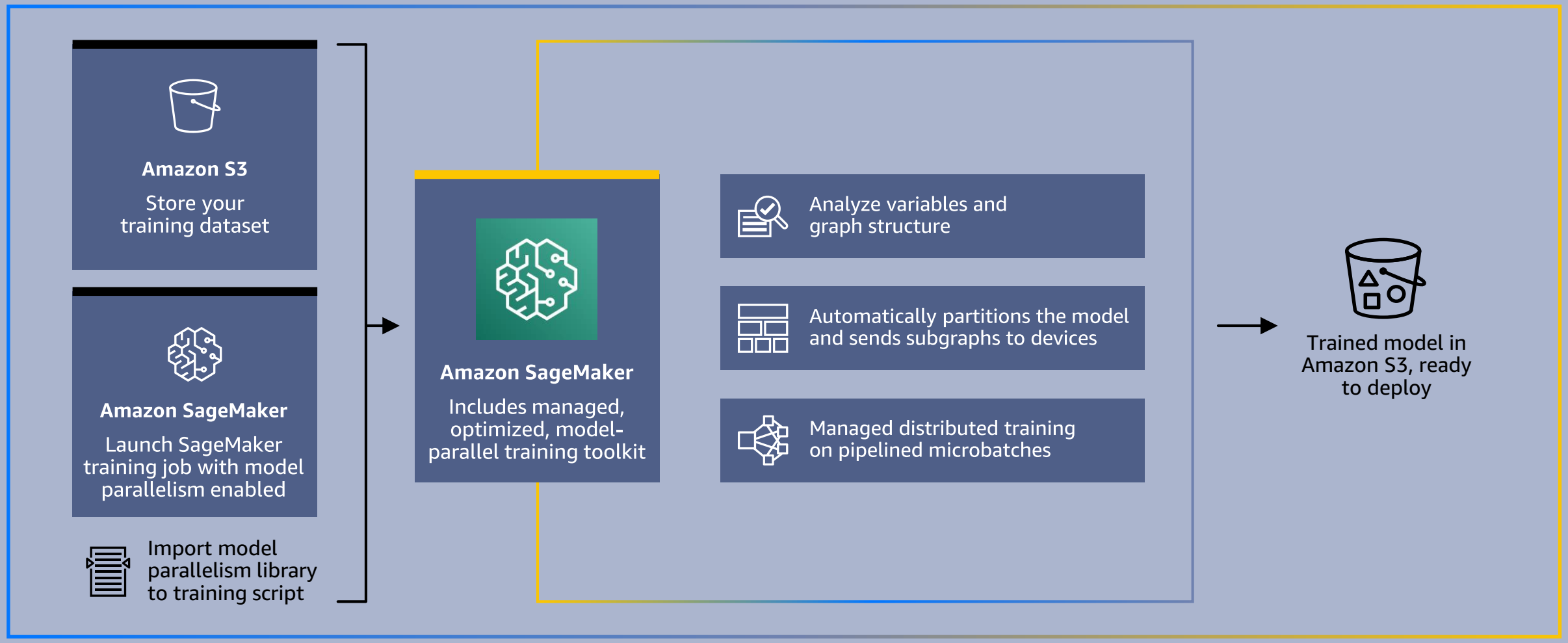
To be executed in a pipelined manner

Pipelined training

- Split batch into N microbatches
- Feed microbatches sequentially
- Minimize idle time on GPUs



Model parallelism on Amazon SageMaker



Benchmarks

ModelParallel training with T5-3B

Model	Instances	Performance with `modelparallel`	Performance without `modelparallel`
T5-3B	8 P4d.24xlarge	299 seq/s	OOM
T5-3B	8 P4d.24xlarge	263 seq/s	OOM
T5-3B	256 P4d.24xlarge	4.68 days	OOM

DataParallel training

Model	Instances	Performance with `dataparallel`	Speed up
RoBERTa (1.3B)	30 P4d.24xlarge	1.85 iter/s	32.4%
RoBERTa (1.3B)	16 P4d.24xlarge	2.00 iter/s	33.1%
Mask-RCNN	64 P3dn.24xlarge	6:12 minutes	25%

Using data parallelism in SageMaker with PyTorch

Using data parallelism with PyTorch

IMPORT AND INITIALIZE SAGEMAKER DATAPARALLEL LIBRARY

```
# SDP: Import SDP PyTorch API
import smdistributed.dataparallel.torch.distributed as dist

# SDP: Import SDP PyTorch DDP
from smdistributed.dataparallel.torch.parallel.distributed import DistributedDataParallel as DDP

# SDP: Initialize SDP
dist.init_process_group()
```

Using data parallelism with PyTorch

NO CHANGE TO CORE TRAINING LOOP

```
class Net(nn.Module):
    ...
    # Define model

def train(...):
    ...
    # Model training

def test(...):
    ...
    # Model evaluation
```

No changes to model definition, core training loop, and evaluation

Using data parallelism with PyTorch

REPLACE PYTORCH DDP WITH SAGEMAKER'S DATAPARALLEL LIBRARY

1. Scale batch size by number of processes
2. Set `number_replicas` = number of GPUs in the cluster; set the node rank in the distributed sampler
3. Wrap the model with `DDP(model)`
4. Pin each GPU to a single SageMaker DDP process
5. Save checkpoints on the leader node

```
def main():

    # SDP: Scale batch size by world size
1   batch_size //= dist.get_world_size() // 8
    batch_size = max(batch_size, 1)

    # Prepare dataset
    train_dataset = torchvision.datasets.MNIST(...)

    # SDP: Set num_replicas and rank in DistributedSampler
    train_sampler = torch.utils.data.distributed.DistributedSampler(
        train_dataset,
2       num_replicas=dist.get_world_size(),
        rank=dist.get_rank())

    train_loader = torch.utils.data.DataLoader(..)

    # SDP: Wrap the PyTorch model with SDP's DDP
3   model = DDP(Net().to(device))

    # SDP: Pin each GPU to a single SDP process.
    torch.cuda.set_device(local_rank)
4   model.cuda(local_rank)

    # Train
    optimizer = optim.Adadelta(...)
    scheduler = StepLR(...)
    for epoch in range(1, args.epochs + 1):
        train(...)
        if rank == 0:
            test(...)
        scheduler.step()

    # SDP: Save model on master node.
5   if dist.get_rank() == 0:
        torch.save(...)

if __name__ == '__main__':
    main()
```

Using model parallelism in SageMaker with TensorFlow

Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```
import tensorflow as tf

class Model(tf.keras.models.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)


train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@tf.function
def train_step(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss


for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()

class Model(tf.keras.models.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)


train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@tf.function
def train_step(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss


for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Import and initialize `modelparallel` library

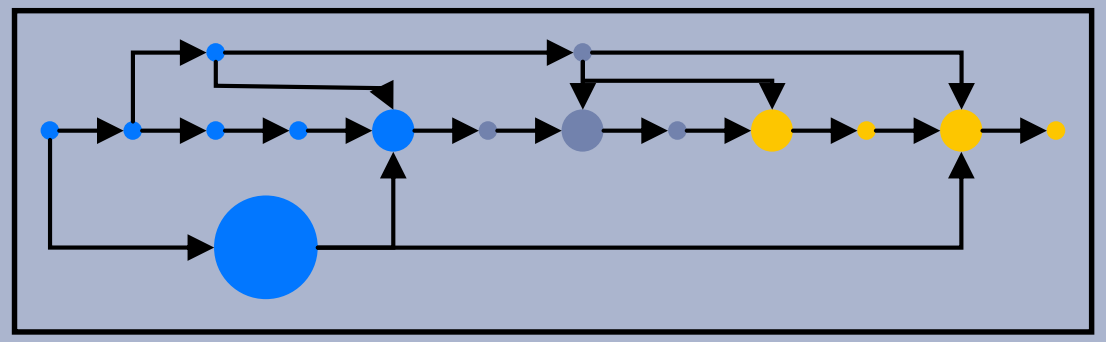
Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()

class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)


train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@tf.function
def train_step(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss


for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Switch Keras Model with `smp.DistributedModel`

TensorFlow operations defined inside `smp.DistributedModel` are subject to automated partition



Any operation outside `smp.DistributedModel` is placed and executed in all devices

Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()

class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)


train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@tf.function
def train_step(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss


for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Extract forward and backward passes into `@smp.step`-decorated function

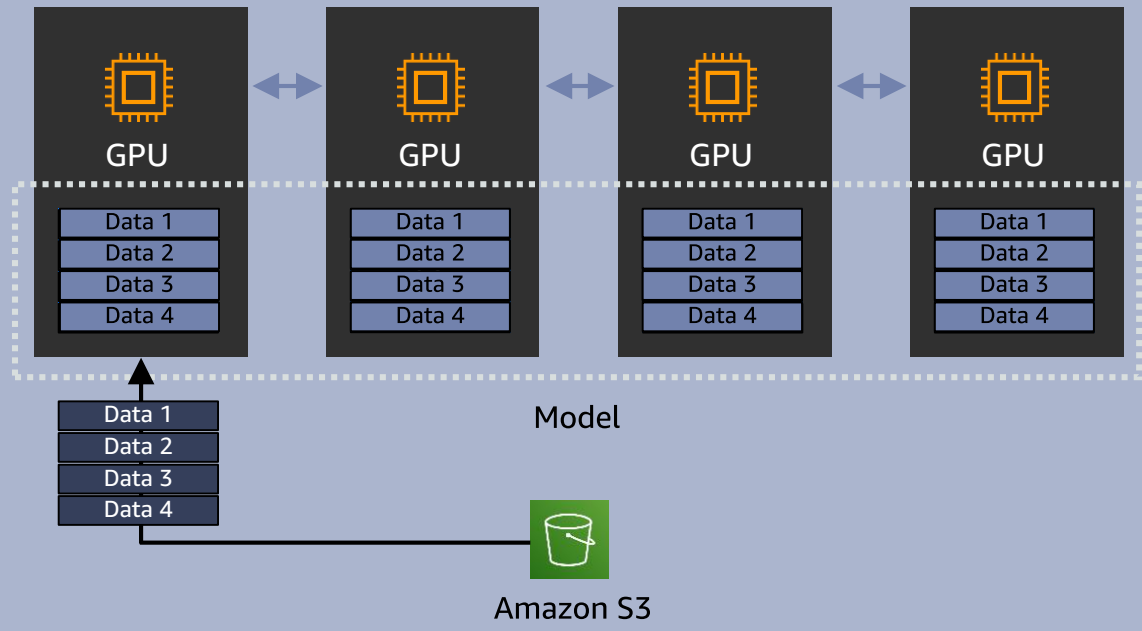
Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```python
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()

class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)

train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    return gradients, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)
    gradients = [g.reduce_mean() for g in gradients]
    loss = loss.reduce_mean()
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Extract forward and backward passes into `@smp.step`-decorated function

`@smp.step` produces `StepOutput` objects containing results from all microbatches – average across microbatches to get single tensor



Return tensors that are needed for taking the training step; typically gradients, but may also include loss, predictions, etc.

Using model parallelism with TensorFlow

MODIFY TRAINING SCRIPT

```
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()

class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)

train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    return gradients, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)
    gradients = [g.reduce_mean() for g in gradients]
    loss = loss.reduce_mean()
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

Extract forward and backward passes into `@smp.step`-decorated function

`@smp.step` produces `StepOutput` objects containing results from all microbatches – average across microbatches to get single tensor

Using model parallelism with TensorFlow

MANUAL PARTITION

```python
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp

smp.init()
class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        with smp.partition(0):
            x = self.conv(x)
            x = self.flatten(x)
        with smp.partition(1):
            x = self.dense1(x)
            return self.dense2(x)

train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    return gradients, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)
    gradients = [g.reduce_mean() for g in gradients]
    loss = loss.reduce_mean()
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

```
"smdistributed": {
    "modelparallel": {
        "enabled": True,
        "parameters": {
            "partitions": 4,
            "microbatches": 4,
            "optimize": "memory",
            "auto_partition": False,
            "default_partition": 0
        }
    }
}
```

1. Define model partitions
2. Disable auto-partition; define default partition

Using model parallelism with TensorFlow

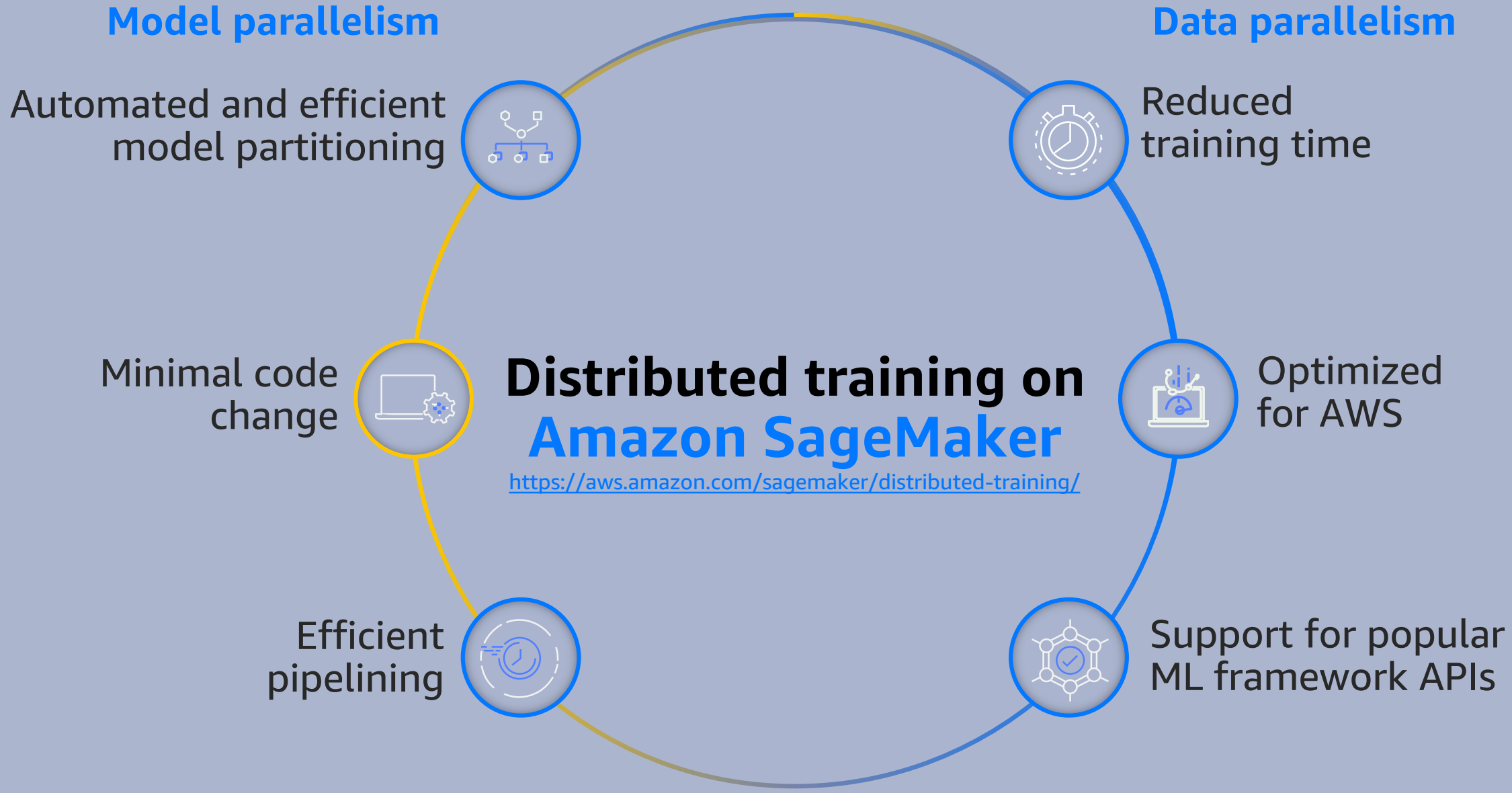
COMBINING DATA PARALLELISM

```
import tensorflow as tf
import smdistributed.modelparallel.tensorflow as smp
import horovod.tensorflow as hvd

smp.init()
class Model(smp.DistributedModel):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv = Conv2D(32, 3, activation="relu")
        self.flatten = Flatten()
        self.dense1 = Dense(128)
        self.dense2 = Dense(10)

    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)

train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@smp.step
def forward_backward(images, labels):
    predictions = model(images, training=True)
    loss = ce_loss(labels, predictions)
    gradients = optimizer.get_gradients(loss, model.trainable_variables)
    return gradients, loss

@tf.function
def train_step(images, labels):
    gradients, loss = forward_backward(images, labels)
    gradients = [hvd.allreduce(g.reduce_mean()) for g in gradients]
    loss = loss.reduce_mean()
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    return loss

for images, labels in train_ds:
    loss = train_step(images, labels)
    print(loss)
```

`hvd.allreduce` can be directly called – no need to initialize Horovod

Just set `"horovod": True` in Python SDK parameters

aws



Learn ML with AWS Training and Certification

LEARN TO APPLY MACHINE LEARNING TO YOUR BUSINESS, UNLOCKING NEW INSIGHTS AND VALUE

Courses appropriate for your role, including developers, data scientists, data platform engineers, and business decision-makers

65+ free digital machine learning courses from AWS experts let you learn from real-world challenges solved by Amazon

Build credibility and confidence with AWS Certification, including *AWS Certified Machine Learning – Specialty*

Go deeper with labs, white papers, tech talks, and more by accessing the AWS Ramp-Up Guide

Visit aws.training/MachineLearning

aws

Thank you!