
AWS
re:Invent

AIM405

Train billion-parameter models with model parallelism on Amazon SageMaker

Can Karakus

Applied Scientist

AWS

AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Agenda

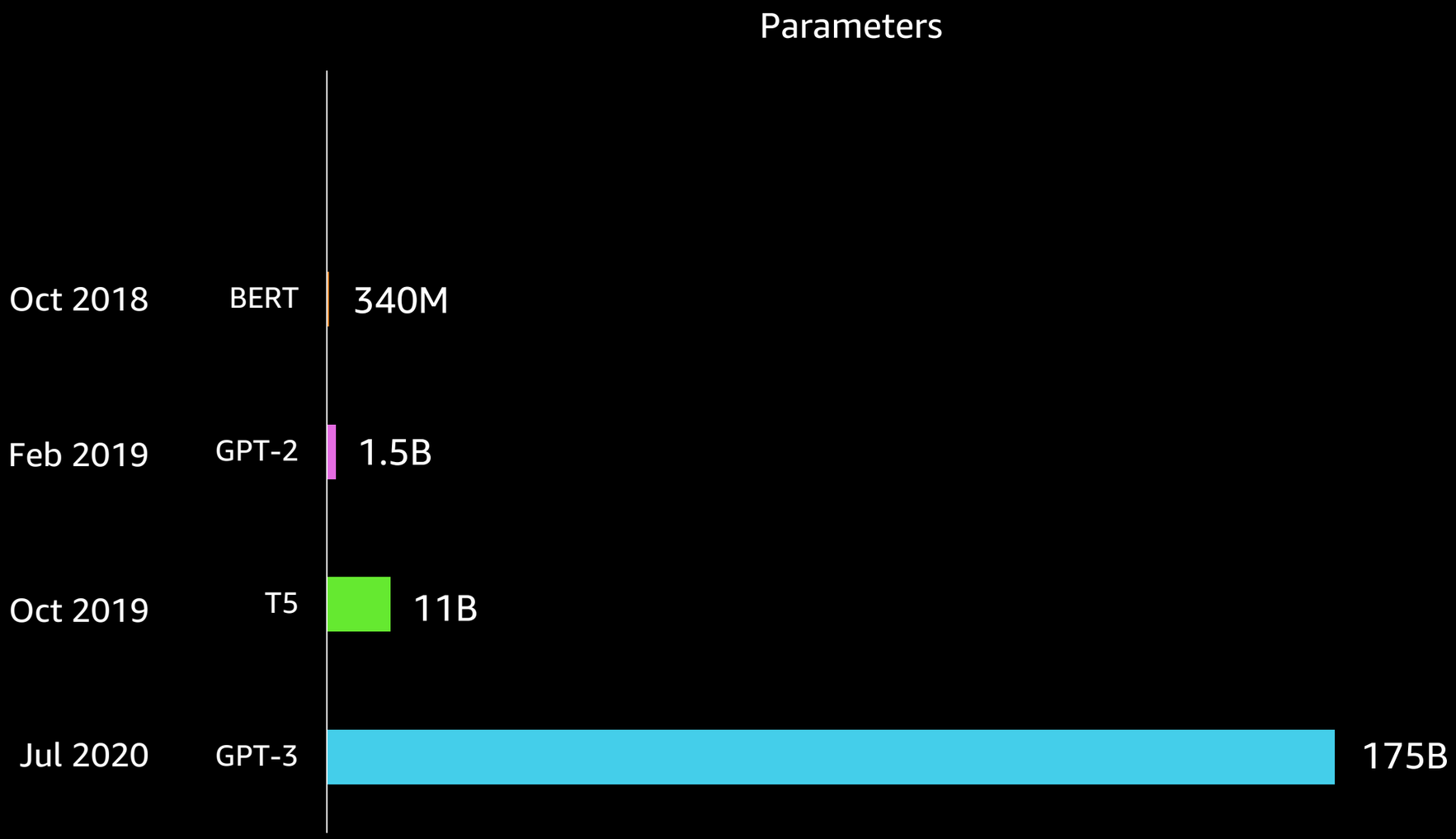
- Why use model parallelism?
- How does model parallelism work?
- Model parallelism on Amazon SageMaker: Main features
- Using model parallelism on Amazon SageMaker with TensorFlow and PyTorch

Why use model parallelism?



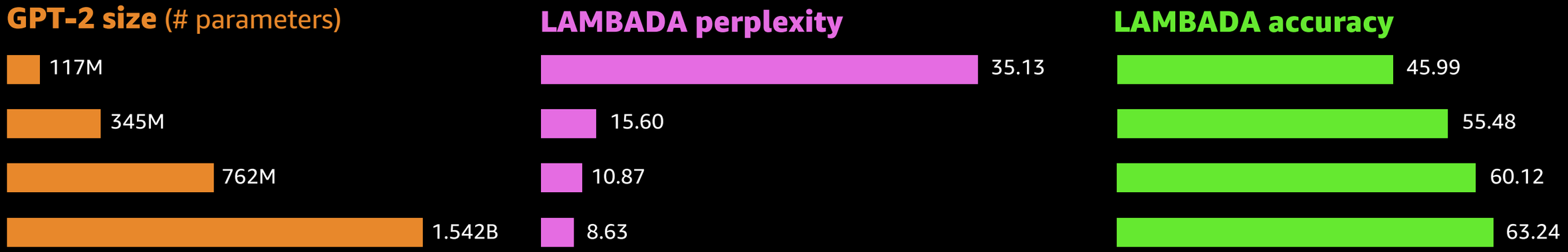
© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Deep learning models are growing

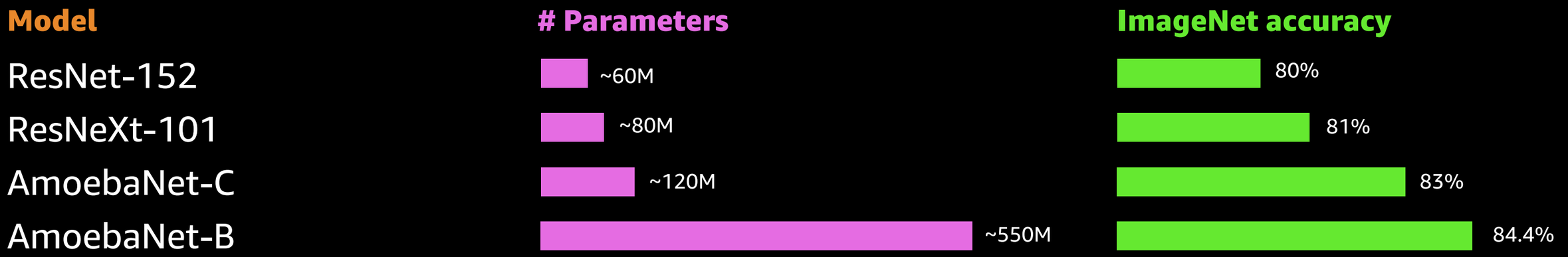


Larger models have higher prediction accuracy

Natural language processing



Computer vision



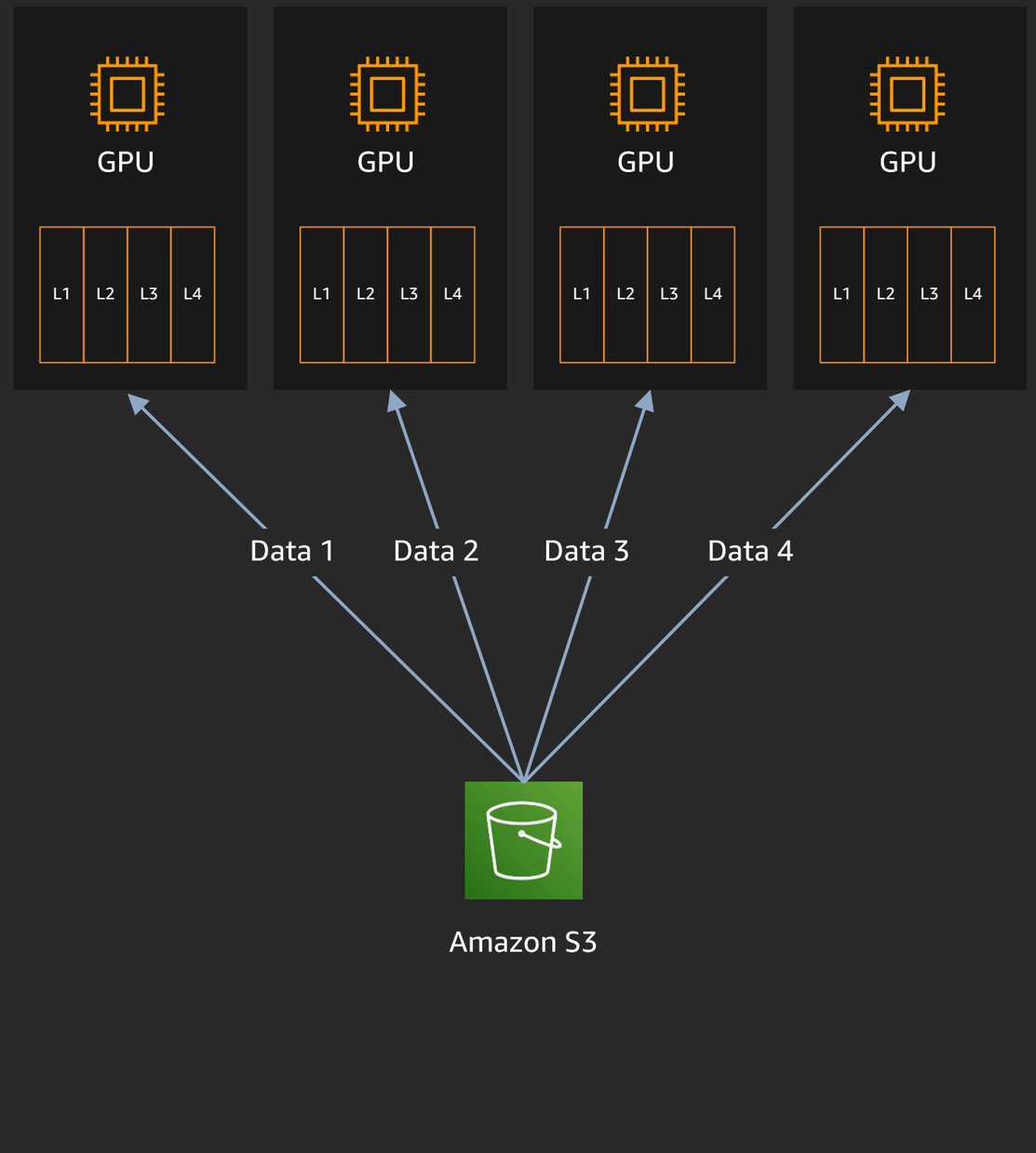
Memory bottlenecks in distributed training

Model size limited by memory of a single GPU

- Large models cause OOM errors

Replicates model across all GPUs

- Wasteful when model is large

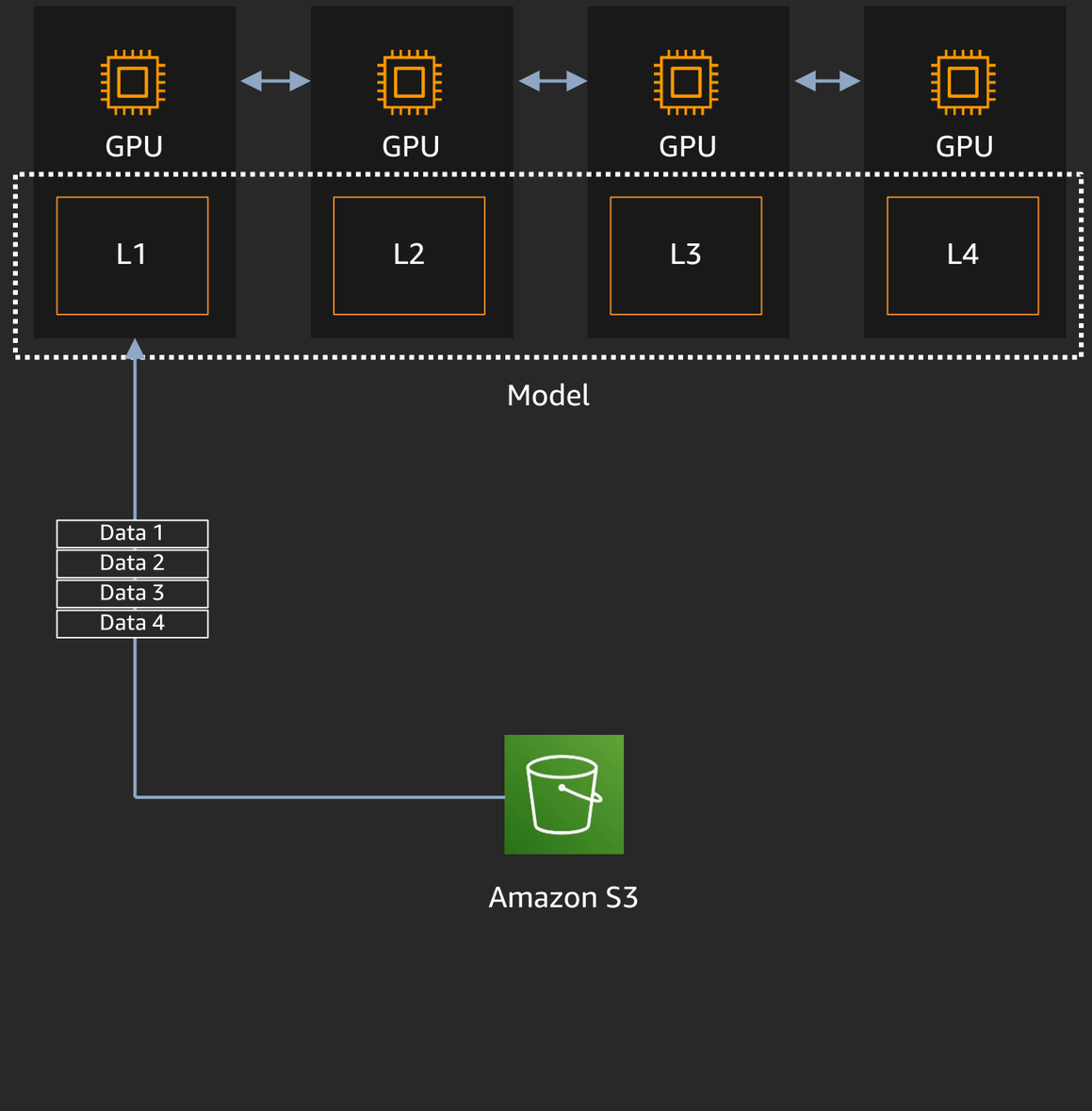


Solution

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

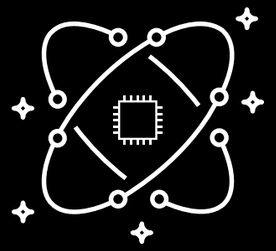


How model parallelism works

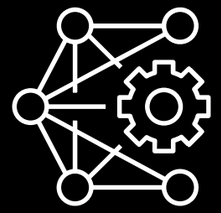


© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

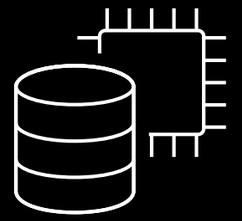
Model parallelism is hard



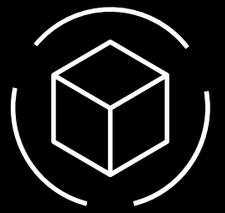
Maximizing
GPU utilization



Finding efficient
partitions



Infrastructure
management

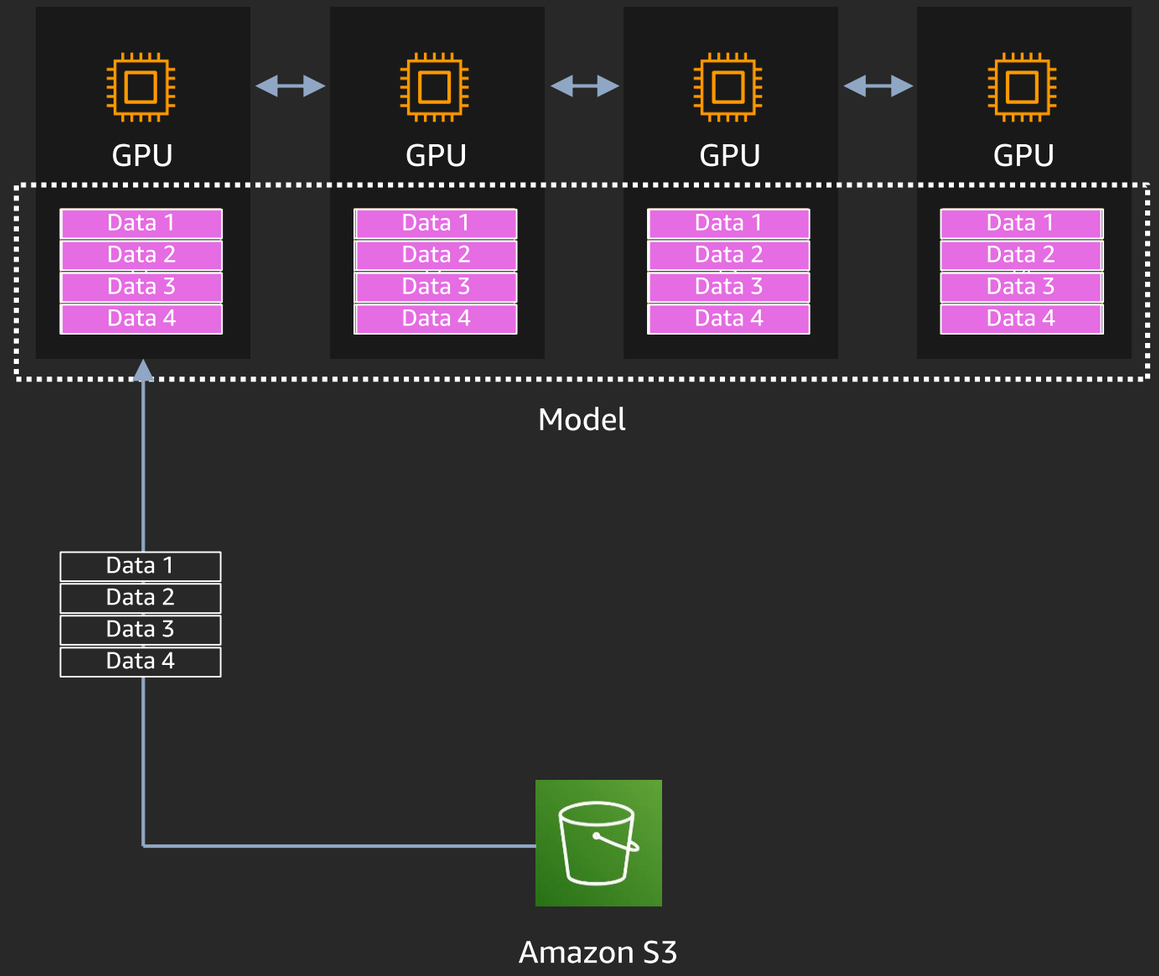


Customized
training code

What's so hard about model parallelism?

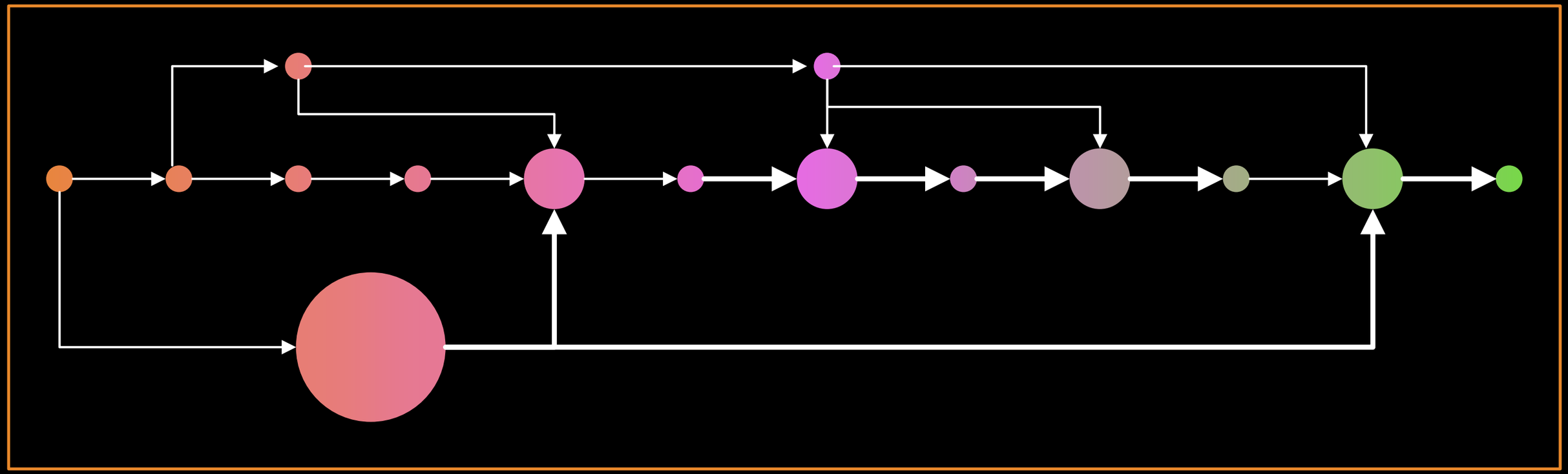
Very low GPU utilization

- Only 1 GPU active at a time



What's so hard about model parallelism?

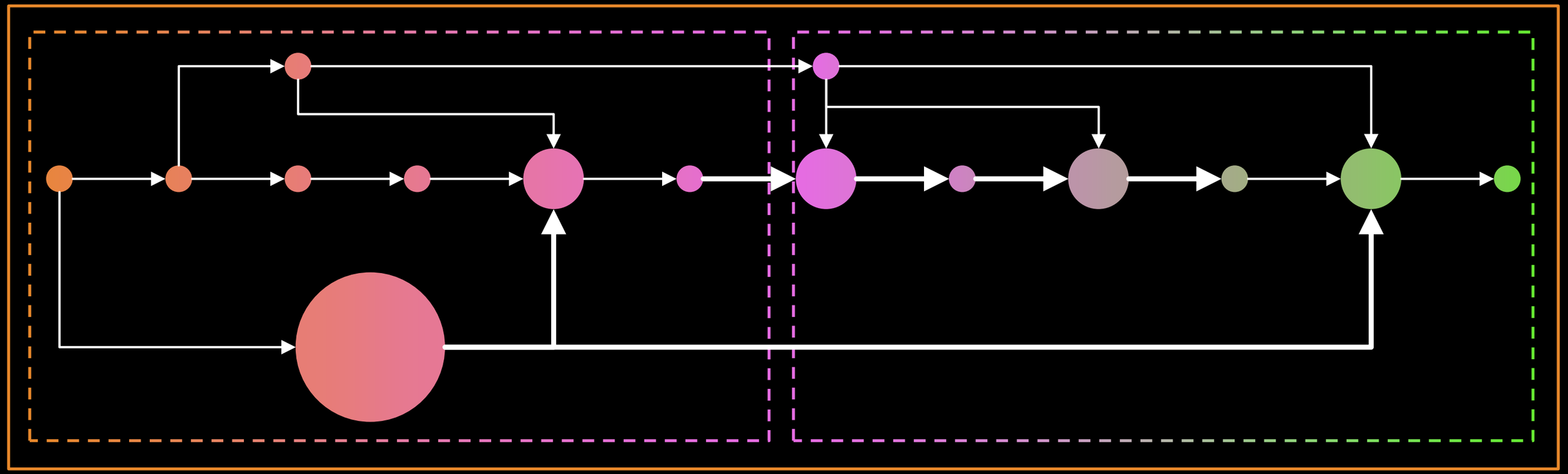
PARTITIONING DECISION



- Where to split the model?
- Large node = expensive computation, thick edge = expensive communication

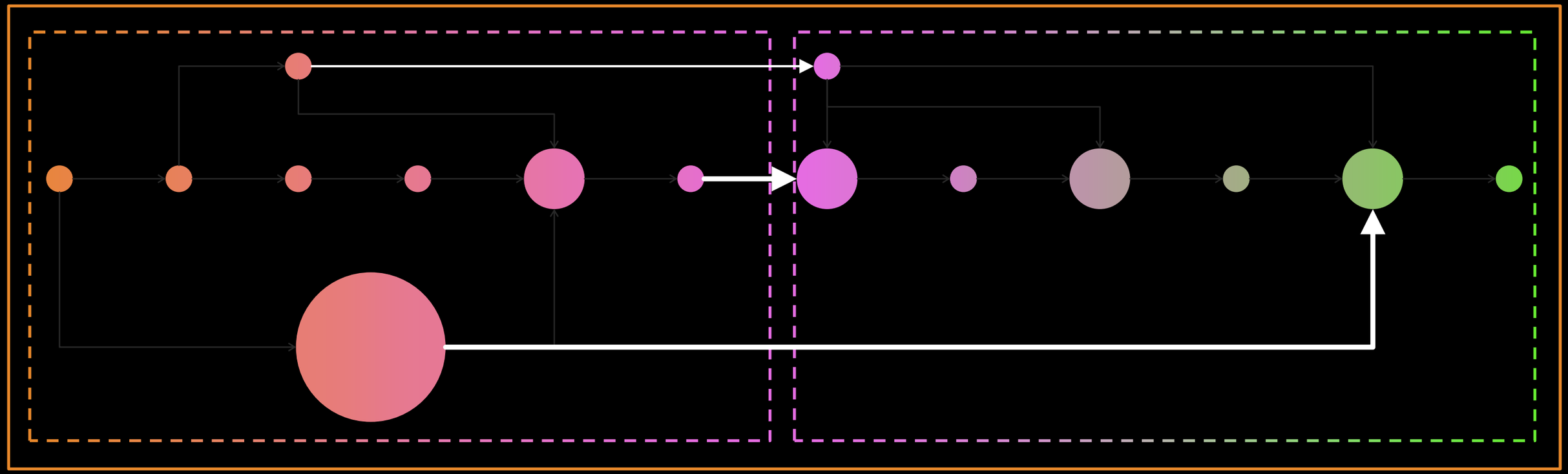
What's so hard about model parallelism?

DECISION 1



What's so hard about model parallelism?

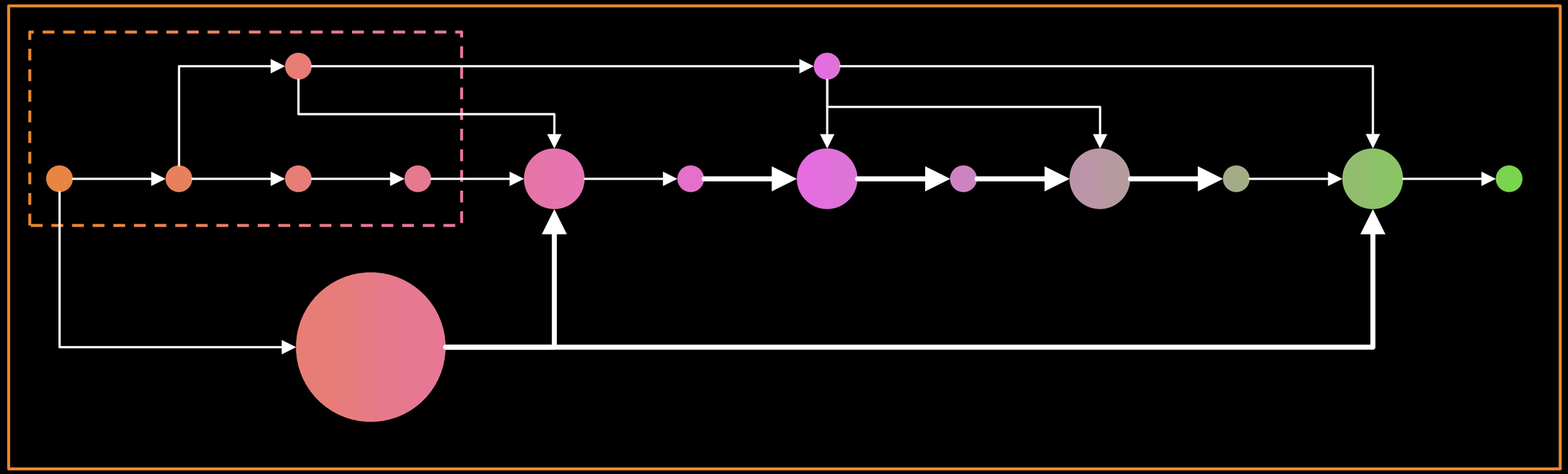
DECISION 1



- Balanced nodes
- Expensive communication

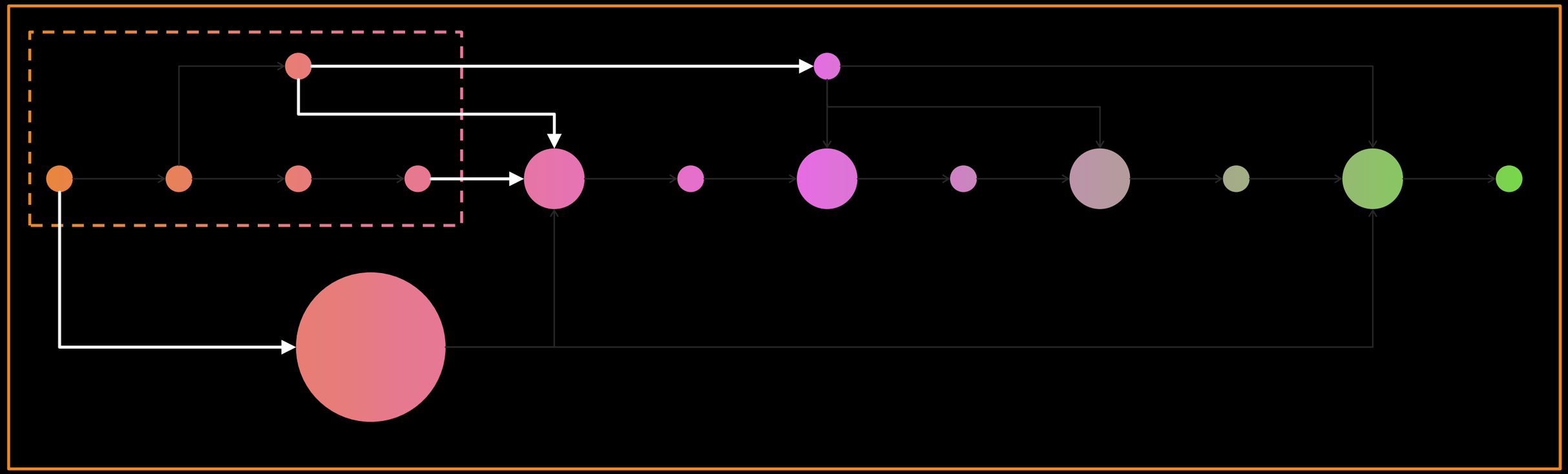
What's so hard about model parallelism?

DECISION 2



What's so hard about model parallelism?

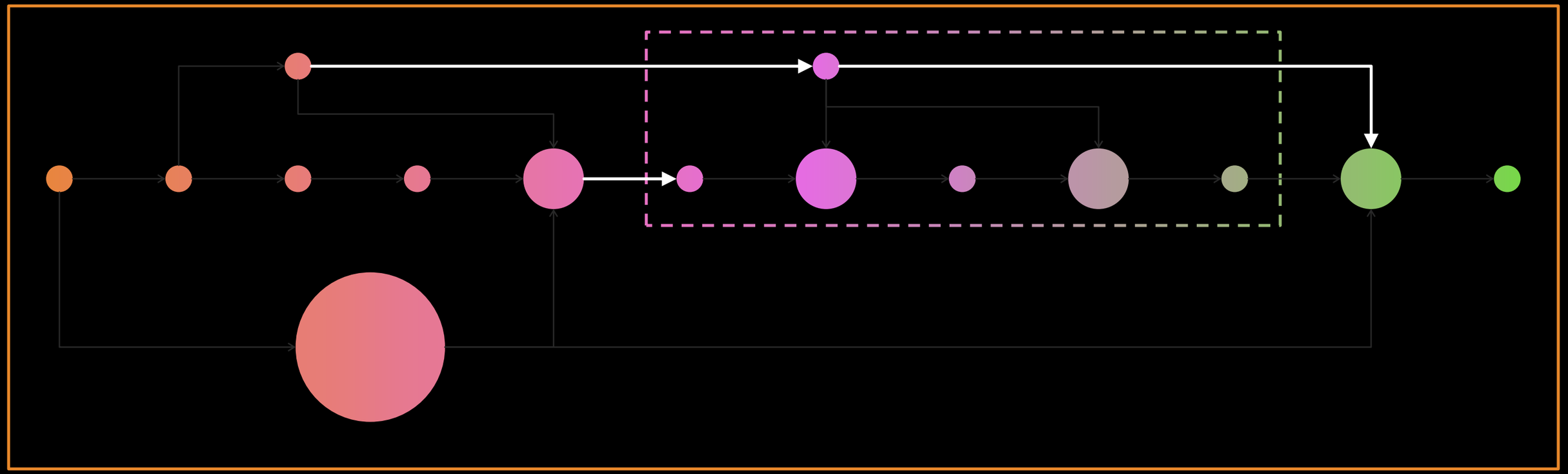
DECISION 2



- Low communication
- Imbalanced nodes

What's so hard about model parallelism?

DECISION 3



- Balanced nodes
- Low communication

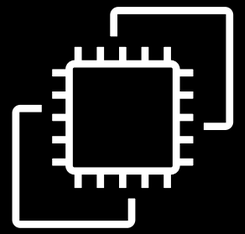
Model parallelism on Amazon SageMaker: Main features



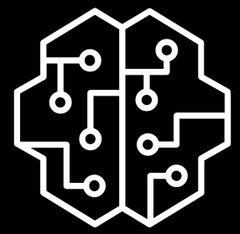
AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

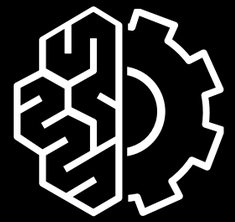
Model parallelism on Amazon SageMaker



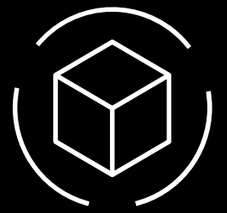
Efficient pipelined training



Automated model partitioning



Managed SageMaker training



Tight framework integration

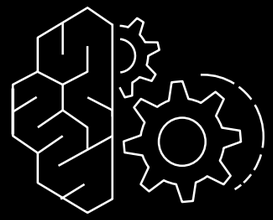
Automated partitioning

Analyze model



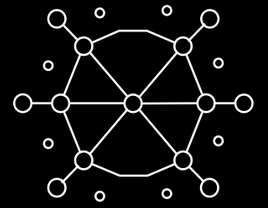
- Graph structure
- Sizes of trainable weights
- Sizes of exchanged tensors

Run graph partitioning algorithm



- Balance stored weights and activations
- Minimize communication

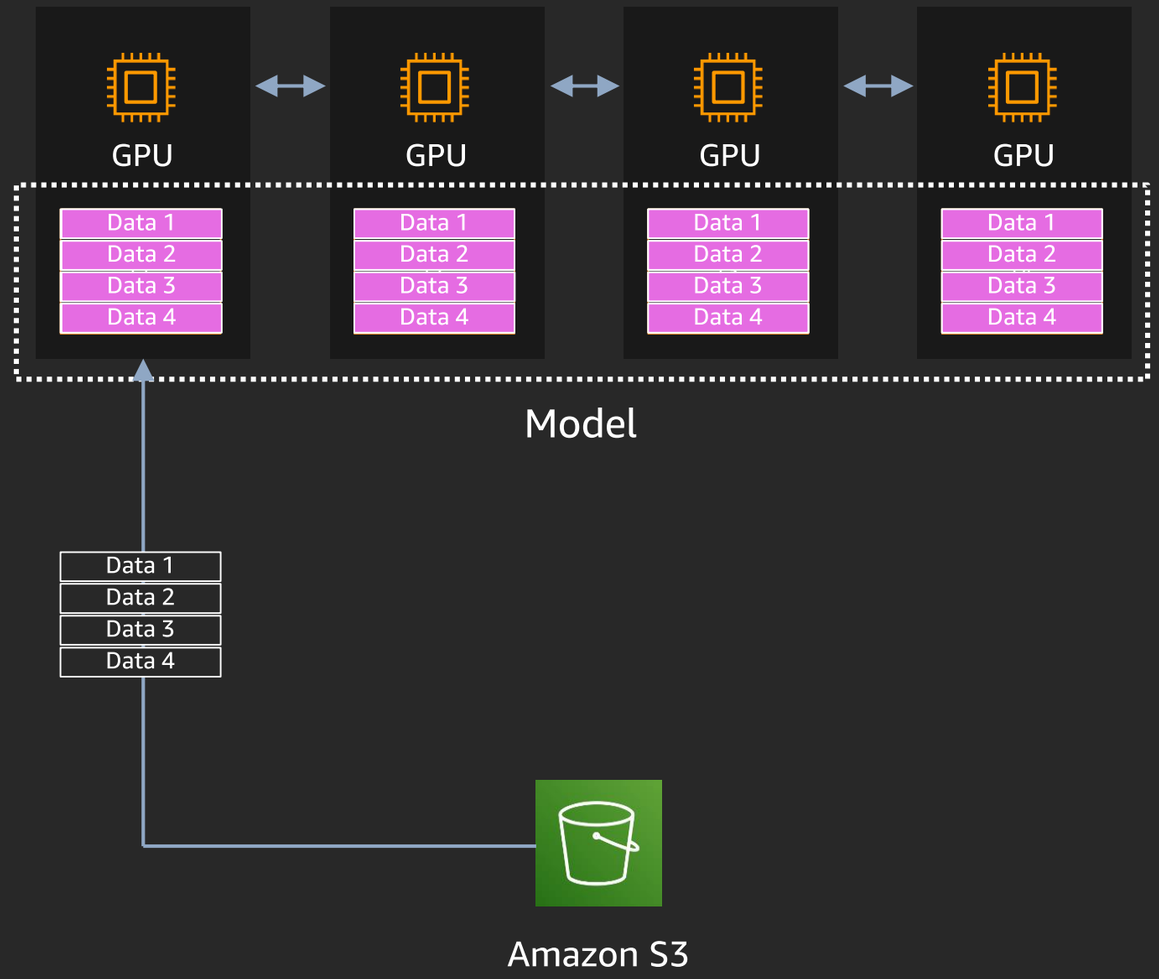
Place partitions on devices



To be executed in a pipelined manner

Pipelined training

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



Using model parallelism on Amazon SageMaker with TensorFlow



AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Using model parallelism with TensorFlow

STEP 1: ADD modelparallel ARGUMENT IN SAGEMAKER ESTIMATOR API

```
sagemaker_session = sagemaker.session.Session(boto_session=session)

mpi_options = {
                "enabled": True,
                "processes_per_host": 4,
                "custom_mpi_options": "--mca btl_vader_single_copy_mechanism none "
               }

smdistributed_options = {
                          "modelparallel": {
                               "enabled": True,
                               "parameters": {
                                   "partitions": 2,
                                   "microbatches": 4,
                                   "optimize": "memory",
                                   }
                               }
                         }

tf_estimator = TensorFlow(
          entry_point="my_script.py",
          role=role,
          instance_type="ml.p3.8xlarge",
          sagemaker_session=sagemaker_session,
          instance_count=1,
          framework_version="2.3",
          py_version="py37",
          base_job_name="modelparallel_job",
          distributions={"mpi": mpi_options, "smdistributed": smdistributed_options},
      )

tf_estimator.fit()
```

Using model parallelism with TensorFlow

STEP 1: ADD modelparallel ARGUMENT IN SAGEMAKER ESTIMATOR API

```
sagemaker_session = sagemaker.session.Session(boto_session=session)

mpi_options = {
                "enabled": True,
                "processes_per_host": 4,
                "custom_mpi_options": "--mca btl_vader_single_copy_mechanism none "
               }

smdistributed_options = {
                         "modelparallel": {
                              "enabled": True,
                              "parameters": {
                                  "partitions": 2,
                                  "microbatches": 4,
                                  "optimize": "memory",
                                  }
                              }
                        }

tf_estimator = TensorFlow(
          entry_point="my_script.py",
          role=role,
          instance_type="ml.p3.8xlarge",
          sagemaker_session=sagemaker_session,
          instance_count=1,
          framework_version="2.3",
          py_version="py37",
          base_job_name="modelparallel_job", 1 2
          distributions={"mpi": mpi_options, "smdistributed": smdistributed_options},
      )

tf_estimator.fit()
```

1. Enable and set MPI options
2. Enable and set `smdistributed.modelparallel` options

Using model parallelism with TensorFlow

STEP 1: ADD modelparallel ARGUMENT IN SAGEMAKER ESTIMATOR API

```
sagemaker_session = sagemaker.session.Session(boto_session=session)

mpi_options = {
              1 "enabled": True,
              2 "processes_per_host": 4,
              3 "custom_mpi_options": "--mca btl_vader_single_copy_mechanism none "
            }

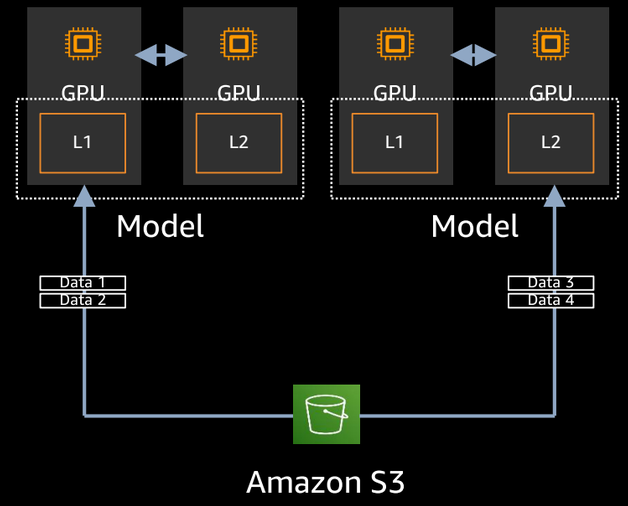
smdistributed_options = {
                         "modelparallel": {
                              "enabled": True,
                              "parameters": {
                                  "partitions": 2,
                                  "microbatches": 4,
                                  "optimize": "memory",
                                  }
                              }
                        }

tf_estimator = TensorFlow(
          entry_point="my_script.py",
          role=role,
          instance_type="ml.p3.8xlarge",
          sagemaker_session=sagemaker_session,
          instance_count=1,
          framework_version="2.3",
          py_version="py37",
          base_job_name="modelparallel_job",
          distributions={"mpi": mpi_options, "smdistributed": smdistributed_options},
      )

tf_estimator.fit()
```

1. Enable MPI
2. Set `processes_per_host` = number of GPUs per instance

   Total processes = (data-parallel degree) x (model-parallel degree)



3. Set custom MPI options

Using model parallelism with TensorFlow

STEP 1: ADD modelparallel ARGUMENT IN SAGEMAKER ESTIMATOR API

```
sagemaker_session = sagemaker.session.Session(boto_session=session)

mpi_options = {
                "enabled": True,
                "processes_per_host": 4,
                "custom_mpi_options": "--mca btl_vader_single_copy_mechanism none "
              }

smdistributed_options = {
                         "modelparallel": {
                            1 "enabled": True,
                              "parameters": {
                                  2 "partitions": 2,
                                    "microbatches": 4,
                                    "optimize": "memory",
                                    }
                              }
                         }

tf_estimator = TensorFlow(
          entry_point="my_script.py",
          role=role,
          instance_type="ml.p3.8xlarge",
          sagemaker_session=sagemaker_session,
          instance_count=1,
          framework_version="2.3",
          py_version="py37",
          base_job_name="modelparallel_job",
          distributions={"mpi": mpi_options, "smdistributed": smdistributed_options},
      )

tf_estimator.fit()
```

1. Enable `modelparallel`
2. Set `modelparallel` parameters

Using model parallelism with TensorFlow

STEP 2: 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

STEP 2: MODIFY TRAINING SCRIPT

Step 2.1: Import and initialize `modelparallel` library

```
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)
```

Using model parallelism with TensorFlow

STEP 2: MODIFY TRAINING SCRIPT

Step 2.2: Switch Keras Model with `smp.DistributedModel`

```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)

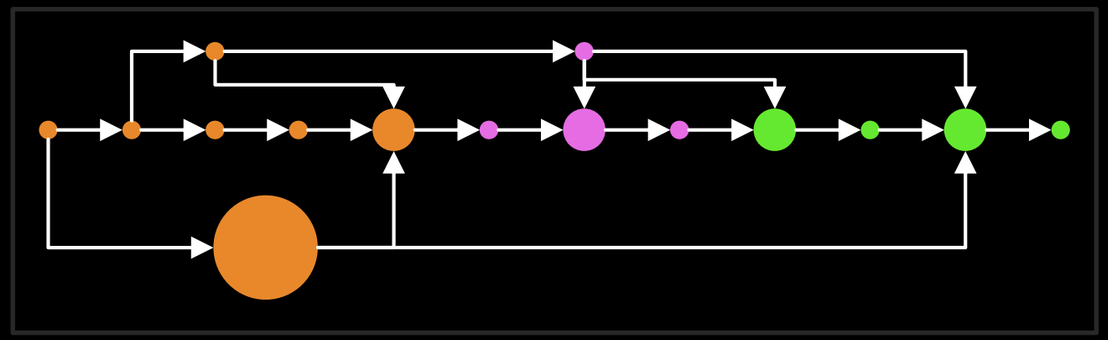
    def call(self, x, training=None):
        x = self.conv(x)
        x = self.flatten(x)
        x = self.dense1(x)
        return self.dense2(x)


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()

@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)
```

TF 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

STEP 2: MODIFY TRAINING SCRIPT

Step 2.3: Extract forward and backward passes into a `@smp.step`-decorated function

```
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)
```

Using model parallelism with TensorFlow

STEP 2: MODIFY TRAINING SCRIPT

Step 2.3: Extract forward and backward passes into a `@smp.step`-decorated function

```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)

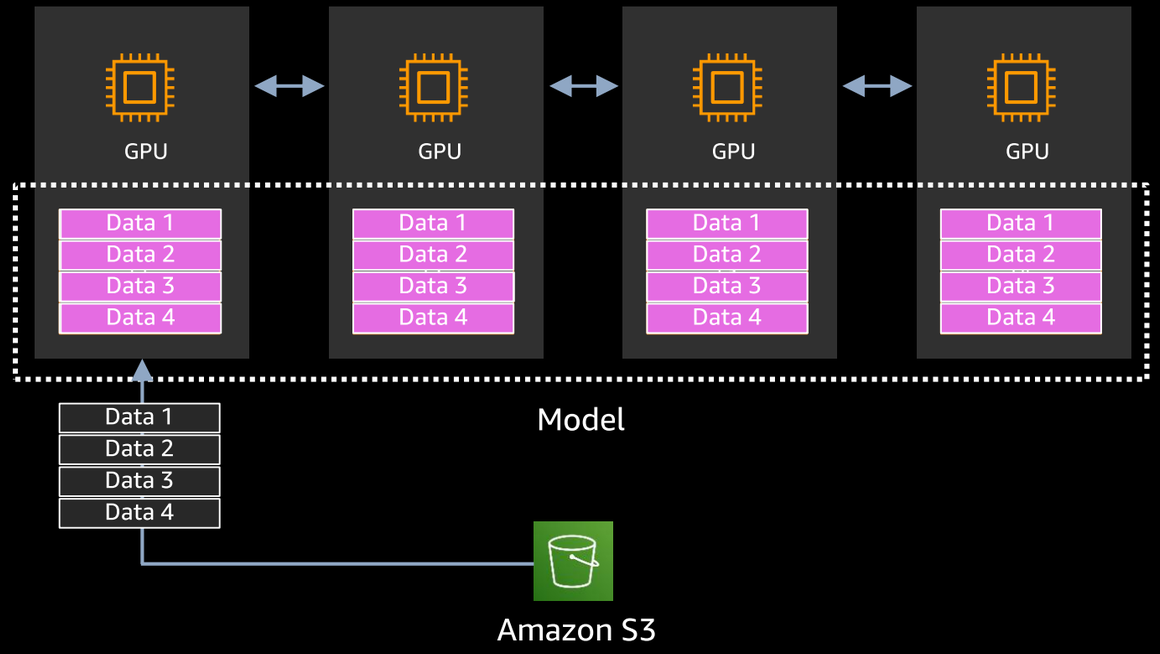
train_ds = tf.data.Dataset.from_tensor_slices("/data").batch(128, drop_remainder=True)
ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

@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)
```

`@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

STEP 2: MODIFY TRAINING SCRIPT

Step 2.3: Extract forward and backward passes into a `@smp.step`-decorated function

```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)
```

`@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



Using model parallelism on Amazon SageMaker with PyTorch



AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```
import torch
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

model = Model()
optimizer = torch.optim.Adadelta(model.parameters(), lr=args.lr)
model.train()

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
    loss.backward()
    optimizer.step()

    print("Loss:", loss)
```

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```
import torch
import torch.nn as nn
import smdistributed.modelparallel.torch as smp
smp.init()

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

model = Model()
optimizer = torch.optim.Adadelta(model.parameters(), lr=args.lr)
model.train()

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
    loss.backward()
    optimizer.step()

    print("Loss:", loss)
```

Import and initialize `modelparallel` library

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```python
import torch
import torch.nn as nn
import smdistributed.modelparallel.torch as smp
smp.init()

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

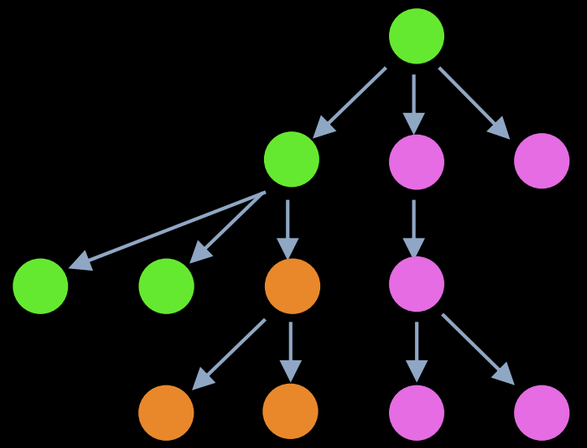
device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

model = smp.DistributedModel(Model())
optimizer = smp.DistributedOptimizer(torch.optim.Adadelta(model.parameters(), lr=args.lr))
model.train()

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
    loss.backward()
    optimizer.step()

    print("Loss:", loss)
```

Wrap model and optimizer; these define the modules that will be partitioned



Partitioning occurs on the tree of `nn.Modules` forming the model – each `nn.Module` is owned by one device

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```
import torch
import torch.nn as nn
import smdistributed.modelparallel.torch as smp
smp.init()

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

torch.cuda.set_device(smp.local_rank())
device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

model = smp.DistributedModel(Model())
optimizer = smp.DistributedOptimizer(torch.optim.Adadelta(model.parameters(), lr=args.lr))
model.train()

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
    loss.backward()
    optimizer.step()

    print("Loss:", loss)
```

Set the device to `smp.local_rank()`

`smp.local_rank()`= GPU ID of the current process

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```
import torch
import torch.nn as nn
import smdistributed.modelparallel.torch as smp
smp.init()

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

torch.cuda.set_device(smp.local_rank())
device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

model = smp.DistributedModel(Model())
optimizer = smp.DistributedOptimizer(torch.optim.Adadelta(model.parameters(), lr=args.lr))
model.train()

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
    loss.backward()
    optimizer.step()

    print("Loss:", loss)
```

Using model parallelism with PyTorch

STEP 2: MODIFY TRAINING SCRIPT

```
import torch
import torch.nn as nn
import smdistributed.modelparallel.torch as smp
smp.init()

class Model(nn.Module):
    def __init__(self):
        super(Net1, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)
        return x

torch.cuda.set_device(smp.local_rank())
device = torch.device("cuda")
dataset = datasets.MNIST("/data", train=True)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=128)

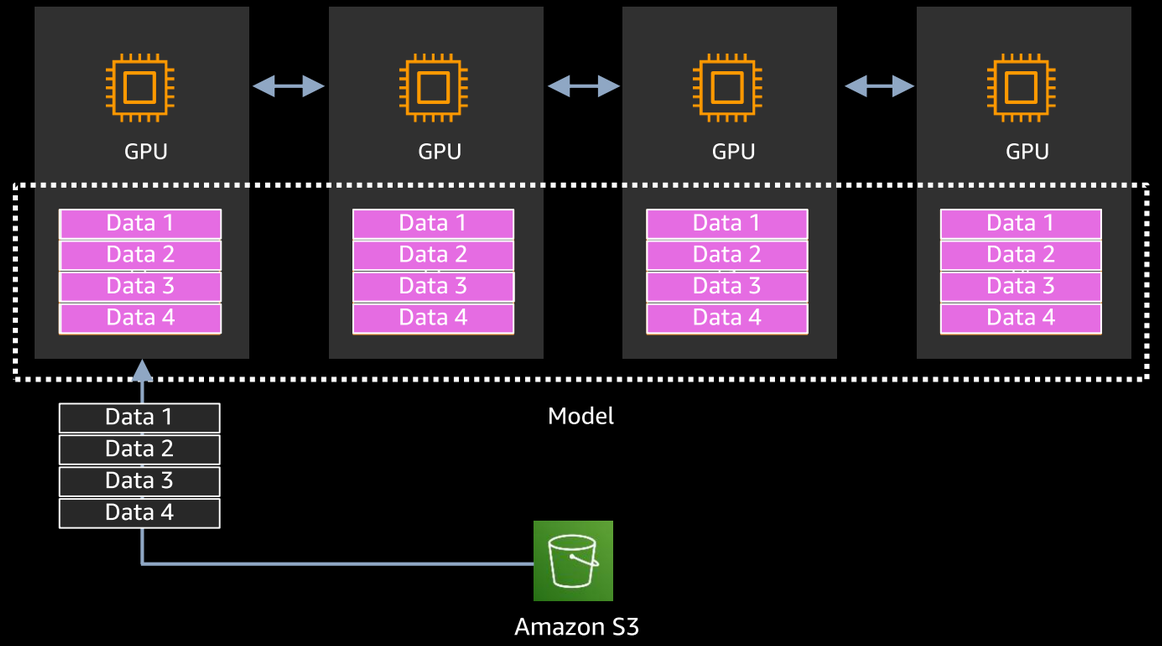
model = smp.DistributedModel(Model())
optimizer = smp.DistributedOptimizer(torch.optim.Adadelta(model.parameters(), lr=args.lr))
model.train()

1 @smp.step
def forward_backward(data, target):
    prediction = model(data)
    loss = F.nll_loss(prediction, target, reduction="mean")
2   model.backward(loss)
    return loss

for batch_idx, (data, target) in enumerate(train_loader):
    data, target = data.to(device), target.to(device)
    optimizer.zero_grad()
    loss = forward_backward(data, target)
    optimizer.step()

3 print("Loss:", loss.reduce_mean())
```

1. Extract the forward and backward pass logic into a `@smp.step`-decorated function – this defines the computations that will be pipelined

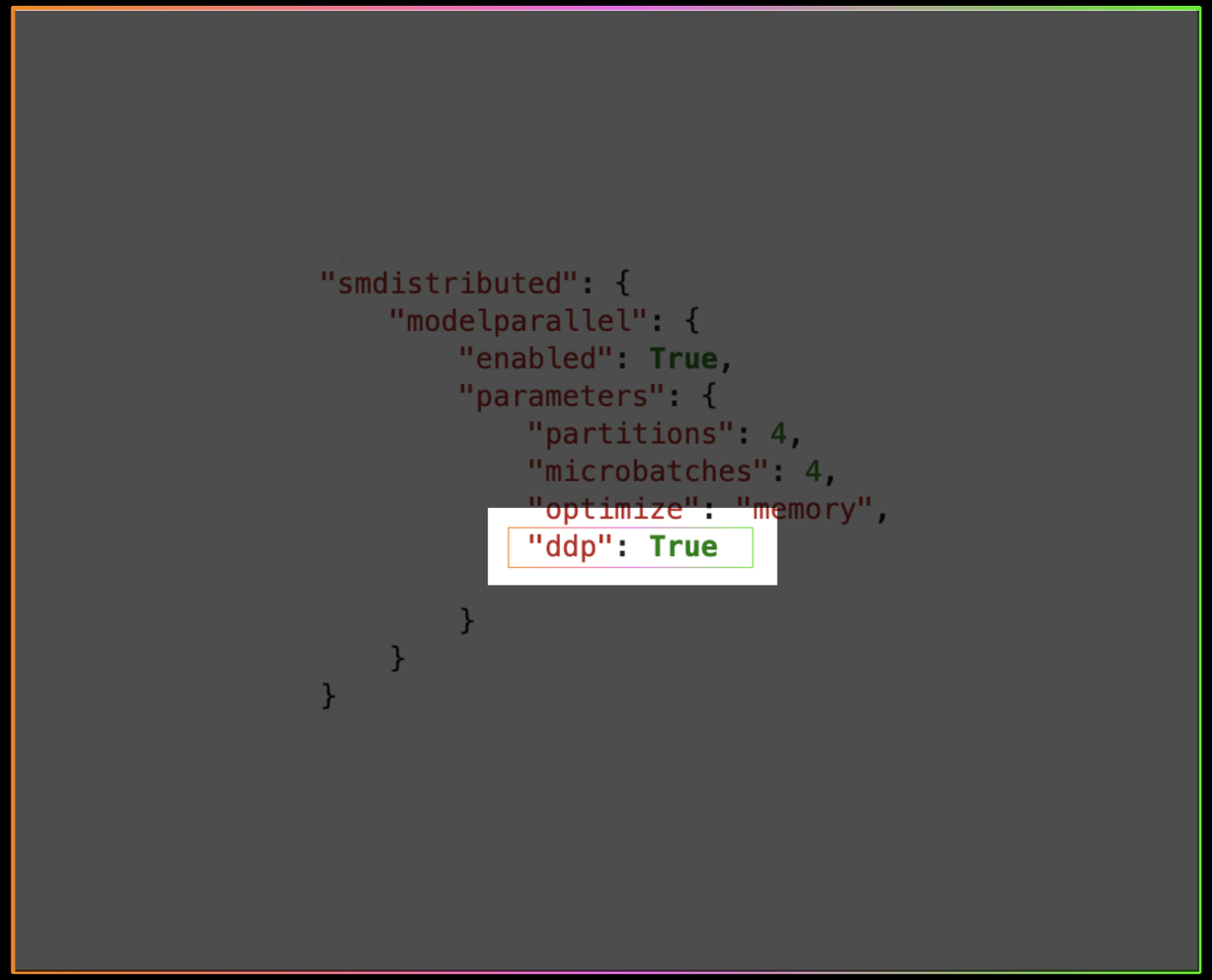


2. Replace `loss.backward()` with `model.backward(loss)`

3. Average loss across microbatches

Using model parallelism with PyTorch

COMBINING WITH DATA PARALLELISM



Set ddp to True

Example training logs

1.5B Parameter model over 4 partitions (TensorFlow)

```
[2020-11-05 07:48:33.701: I modelparallel/tensorflow/auto.py:298] Total variables in device 0: 354435392
[2020-11-05 07:48:33.701: I modelparallel/tensorflow/auto.py:298] Total variables in device 1: 390439002
[2020-11-05 07:48:33.701: I modelparallel/tensorflow/auto.py:298] Total variables in device 2: 486174000
[2020-11-05 07:48:33.701: I modelparallel/tensorflow/auto.py:298] Total variables in device 3: 268451840
```

3B Parameter model over 8 partitions (PyTorch)

```
[2020-11-05 08:02:41.298: I modelparallel/torch/model.py:264] Number of parameters on partition 1 are 146. 146 require grads
[2020-11-05 08:02:41.302: I modelparallel/torch/model.py:280] Number of buffers on partition 1 are 0.
[2020-11-05 08:02:41.388: I modelparallel/torch/model.py:264] Number of parameters on partition 5 are 144. 144 require grads
[2020-11-05 08:02:41.392: I modelparallel/torch/model.py:280] Number of buffers on partition 5 are 0.
[2020-11-05 08:02:41.686: I modelparallel/torch/model.py:264] Number of parameters on partition 2 are 62. 62 require grads
[2020-11-05 08:02:41.690: I modelparallel/torch/model.py:280] Number of buffers on partition 2 are 0.
[2020-11-05 08:02:41.838: I modelparallel/torch/model.py:264] Number of parameters on partition 3 are 72. 72 require grads
[2020-11-05 08:02:41.842: I modelparallel/torch/model.py:280] Number of buffers on partition 3 are 0.
[2020-11-05 08:02:41.858: I modelparallel/torch/model.py:264] Number of parameters on partition 6 are 60. 60 require grads
[2020-11-05 08:02:41.862: I modelparallel/torch/model.py:280] Number of buffers on partition 6 are 0.
[2020-11-05 08:02:41.877: I modelparallel/torch/model.py:264] Number of parameters on partition 4 are 60. 60 require grads
[2020-11-05 08:02:41.880: I modelparallel/torch/model.py:280] Number of buffers on partition 4 are 0.
[2020-11-05 08:02:41.911: I modelparallel/torch/model.py:264] Number of parameters on partition 7 are 72. 72 require grads
[2020-11-05 08:02:41.915: I modelparallel/torch/model.py:280] Number of buffers on partition 7 are 0.
[2020-11-05 08:02:42.610: I modelparallel/torch/model.py:264] Number of parameters on partition 0 are 116. 116 require grads
[2020-11-05 08:02:42.613: I modelparallel/torch/model.py:280] Number of buffers on partition 0 are 0.
```

Benchmarks

GPT-2 XL (1.5B parameters), TensorFlow

# Instances	Total batch size	Throughput without `modelparallel` (seq/s)	Throughput with `modelparallel` (seq/s)
1 P3dn.24xlarge	64	OOM	32
2 P3dn.24xlarge	128	OOM	46.4
4 P3dn.24xlarge	256	OOM	76.8

T5-3B (3B parameters), PyTorch

# Instances	Total batch size	Throughput without `modelparallel` (seq/s)	Throughput with `modelparallel` (seq/s)
1 P3dn.24xlarge	32	OOM	19.5
2 P3dn.24xlarge	64	OOM	36
8 P3dn.24xlarge	256	OOM	102

Resources

To get started, check out the `smdistributed.modelparallel` notebooks:

https://github.com/aws/amazon-sagemaker-examples/tree/master/training/distributed_training/tensorflow/model_parallel

https://github.com/aws/amazon-sagemaker-examples/tree/master/training/distributed_training/pytorch/model_parallel

Thank you!



AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.



Please complete the session survey

AWS re:Invent

© 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.