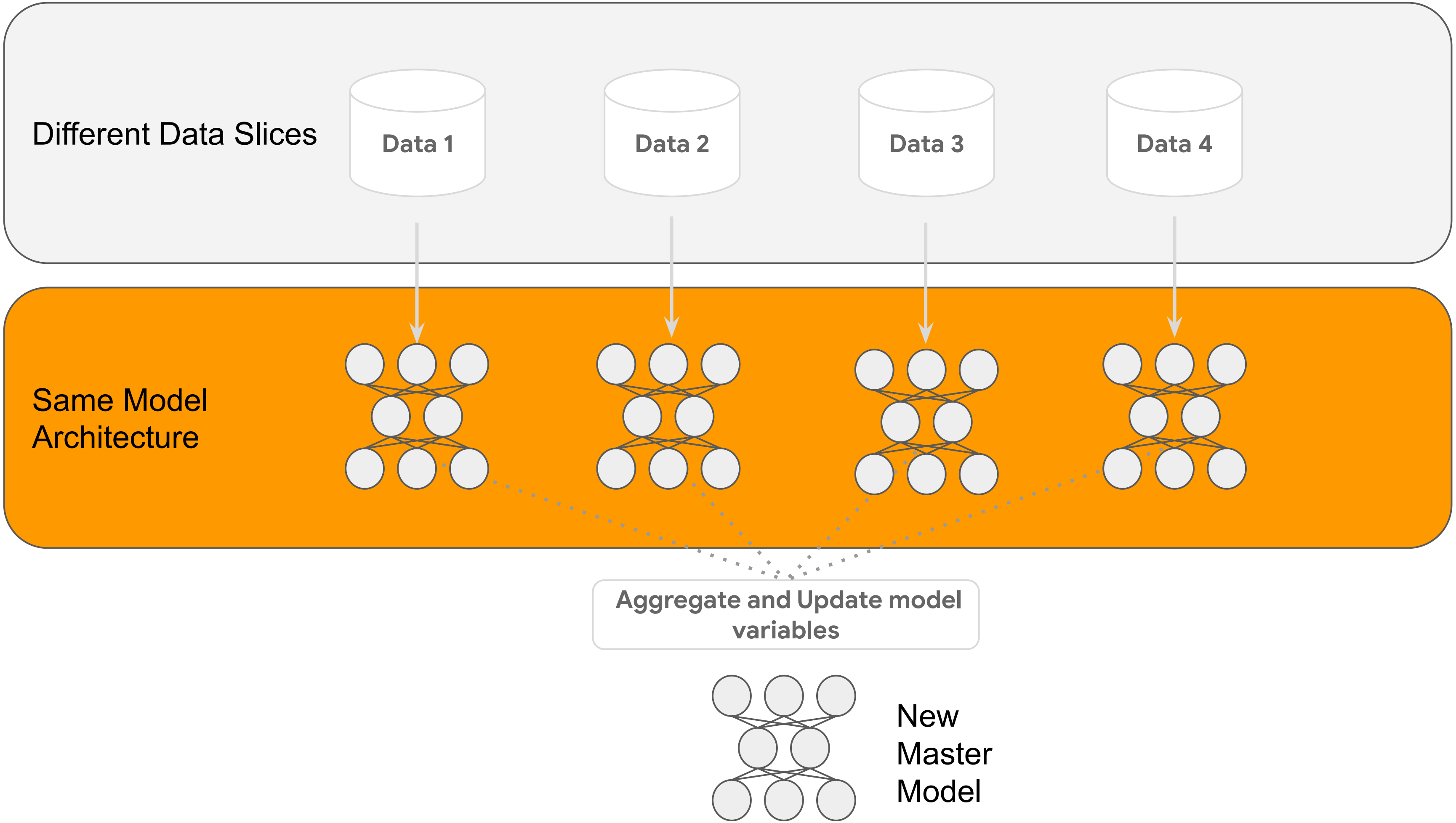
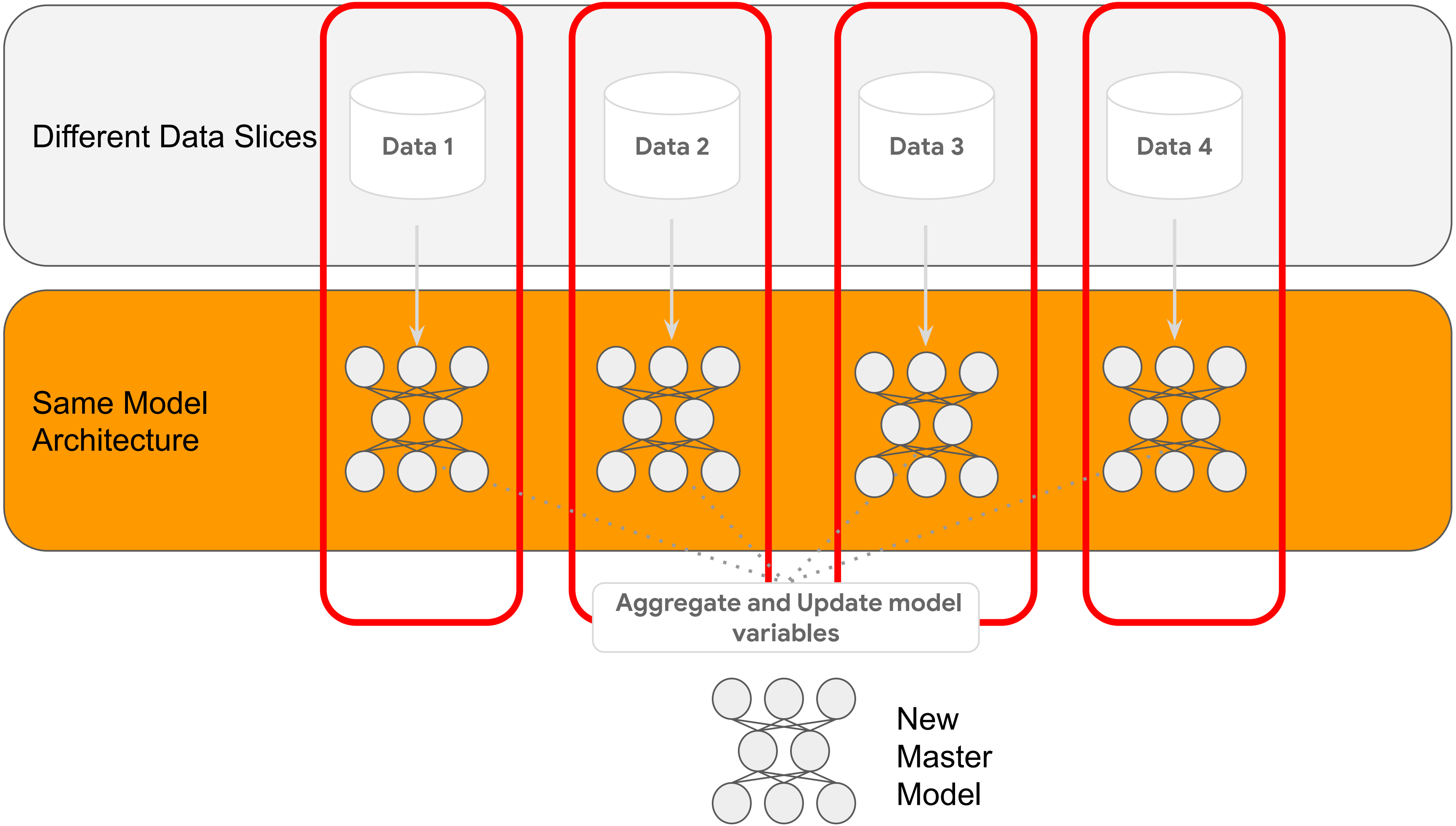
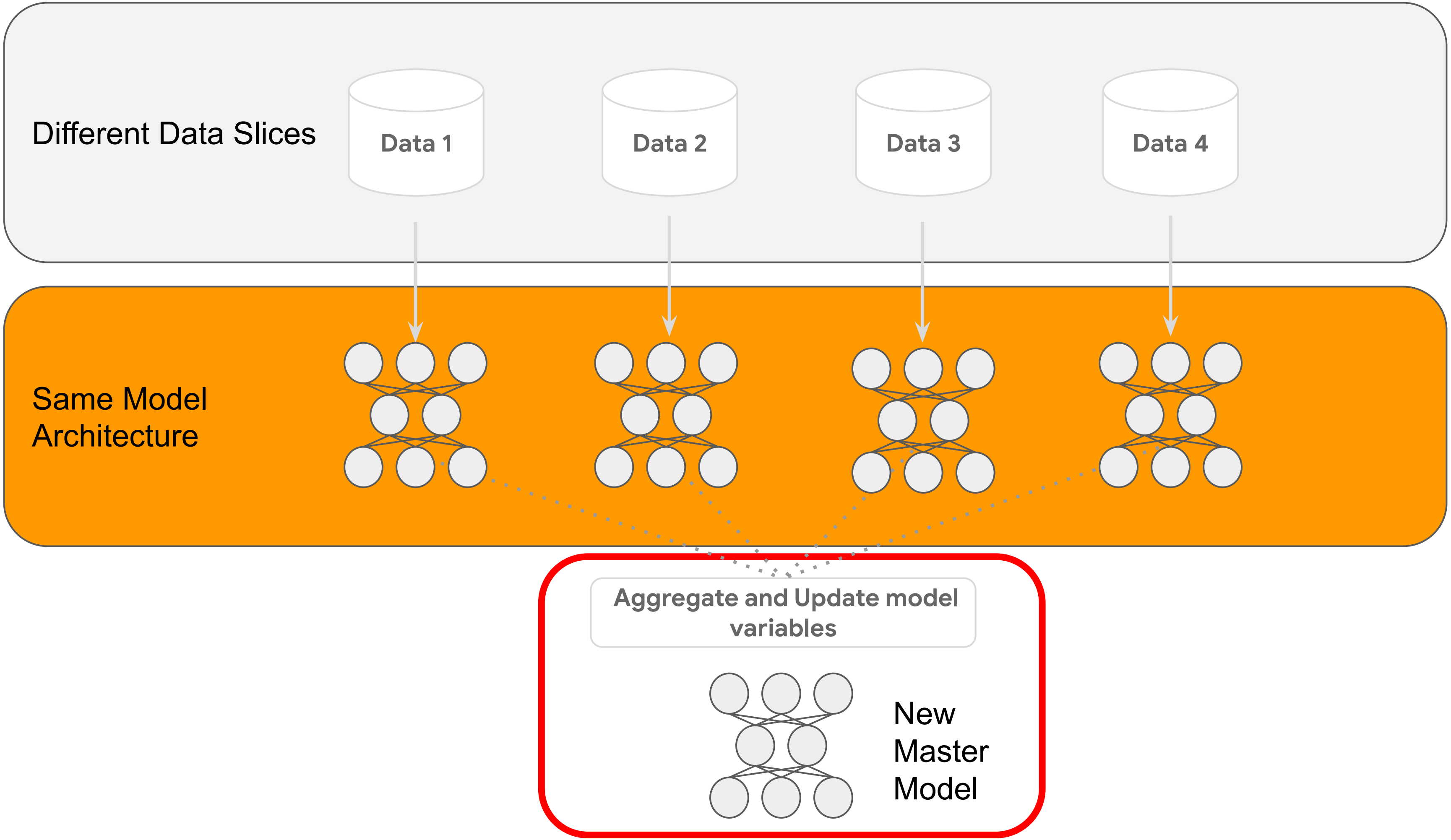
Data Parallelism



Data Parallelism



Data Parallelism



`tf.distribute.Strategy`

- High-level APIs
- Custom training loops
- TensorFlow 2: eager mode & graph mode
- Supported on multiple configurations.
- Convenient to use with little to no code changes

Commonly used terms

- Device
- Replica
- Worker
- Mirrored variable

Commonly used terms

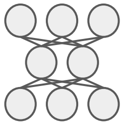
- Device
- Replica
- Worker
- Mirrored variable

CPU
Accelerator: GPU, TPU

Commonly used terms

- Device
- Replica
- Worker
- Mirrored variable

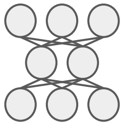
CPU
Accelerator: GPU, TPU



Commonly used terms

- Device
- Replica
- Worker
- Mirrored variable

CPU
Accelerator: GPU, TPU

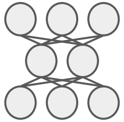


Commonly used terms

- Device

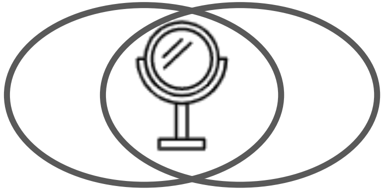
CPU
Accelerator: GPU, TPU

- Replica



- Worker

- Mirrored variable



Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

Classifying strategies

Hardware platforms

- Single-machine multi-device

- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

Classifying strategies

Hardware platforms

- Single-machine multi-device

- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

Classifying strategies

Hardware platforms

- Single-machine multi-device
- Multi-machine (with 0 or more accelerators)

Training

- Synchronous (All-reduce)
- Asynchronous (Parameter Server)

MirroredStrategy

MultiWorkerMirroredStrategy

ParameterServerStrategy

DefaultStrategy

TPUStrategy

CentralStorageStrategy

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

ParameterServerStrategy

DefaultStrategy

TPUStrategy

CentralStorageStrategy

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

ParameterServerStrategy

DefaultStrategy

TPUStrategy

- Same as MirroredStrategy
- All-reduce across ***TPU cores***

CentralStorageStrategy

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

- Multi-machine multi-GPU
- Replicates variables per device ***across workers***
- All-reduce based on
  - hardware
  - network topology
  - tensor sizes

ParameterServerStrategy

DefaultStrategy

TPUStrategy

- Same as MirroredStrategy
- All-reduce across ***TPU cores***

CentralStorageStrategy

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

- Multi-machine multi-GPU
- Replicates variables per device ***across workers***
- All-reduce based on
  - hardware
  - network topology
  - tensor sizes

ParameterServerStrategy

DefaultStrategy

TPUStrategy

- Same as MirroredStrategy
- All-reduce across ***TPU cores***

CentralStorageStrategy

- Variables are ***not mirrored***

(instead placed on the CPU)

- Done in-memory on a device

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

- Multi-machine multi-GPU
- Replicates variables per device ***across workers***
- All-reduce based on
  - hardware
  - network topology
  - tensor sizes

ParameterServerStrategy

- Some machines designated as ***workers***
- Some others as ***parameter servers***

DefaultStrategy

TPUStrategy

- Same as MirroredStrategy
- All-reduce across ***TPU cores***

CentralStorageStrategy

- Variables are ***not mirrored***

(instead placed on the CPU)

- Done in-memory on a device

OneDeviceStrategy

MirroredStrategy

- Single-machine multi-GPU
- Creates a replica per ***GPU***
- Each variable is ***mirrored***
- All-reduce ***across devices***

MultiWorkerMirroredStrategy

- Multi-machine multi-GPU
- Replicates variables per device ***across workers***
- All-reduce based on
  - hardware
  - network topology
  - tensor sizes

ParameterServerStrategy

- Some machines designated as ***workers***
- Some others as ***parameter servers***

DefaultStrategy

- Simple Passthrough

OneDeviceStrategy

- Single device

TPUStrategy

- Same as MirroredStrategy
- All-reduce across ***TPU cores***

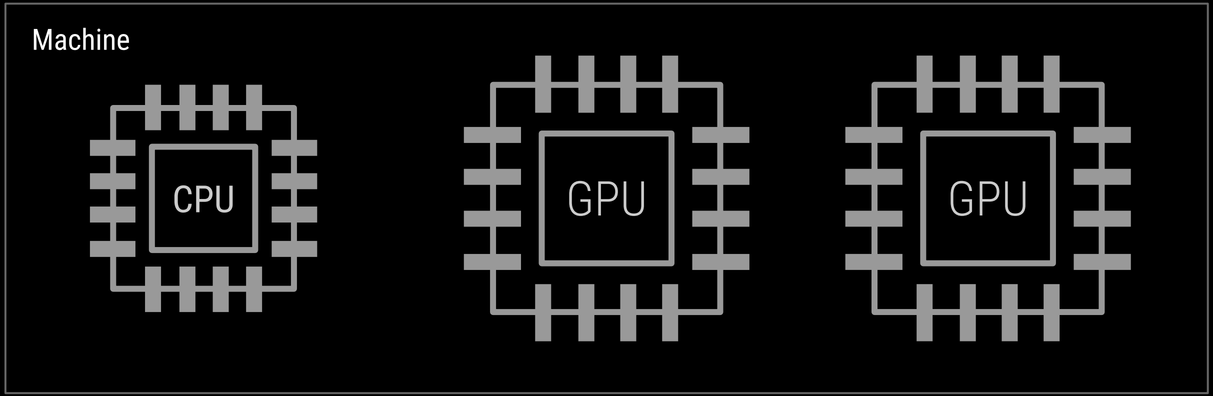
CentralStorageStrategy

- Variables are ***not mirrored***

(instead placed on the CPU)

- Done in-memory on a device

Mirrored Strategy



Mirrored Strategy

- Model declaration
- Data preprocessing

```python
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

```python
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
```

```python
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE = 64

train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```python
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE = 64

train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```python
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255
    return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE = 64

train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```
strategy = tf.distribute.MirroredStrategy()


print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

```python
strategy = tf.distribute.MirroredStrategy()


print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

```
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```python
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```python
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255

  return image, label

num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

```python
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])


model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])
```

```python
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])


model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])
```

```python
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])


model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])
```

```
Epoch 1/12
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
INFO:tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadcast to
```

Training across local GPUs

`tf.distribute.MirroredStrategy`

- Each variable in the model is mirrored across all replicas
- Variables are treated as `MirroredVariable`
- **Synchronization** done with NVIDIA NCCL

```python
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                     .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                     .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

```python
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                    .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                    .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

```
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                     .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                     .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

```python
# Create Datasets from the batches
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
                    .shuffle(BUFFER_SIZE).batch(GLOBAL_BATCH_SIZE)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
                    .batch(GLOBAL_BATCH_SIZE)

# Create Distributed Datasets from the datasets
train_dist_dataset = strategy.experimental_distribute_dataset(train_dataset)
test_dist_dataset = strategy.experimental_distribute_dataset(test_dataset)
```

```python
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

```python
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

```python
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

```
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

```python
EPOCHS = 10
for epoch in range(EPOCHS):
  # Do Training
  total_loss = 0.0
  num_batches = 0
  for batch in train_dist_dataset:
    total_loss += distributed_train_step(batch)
    num_batches += 1
  train_loss = total_loss / num_batches
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

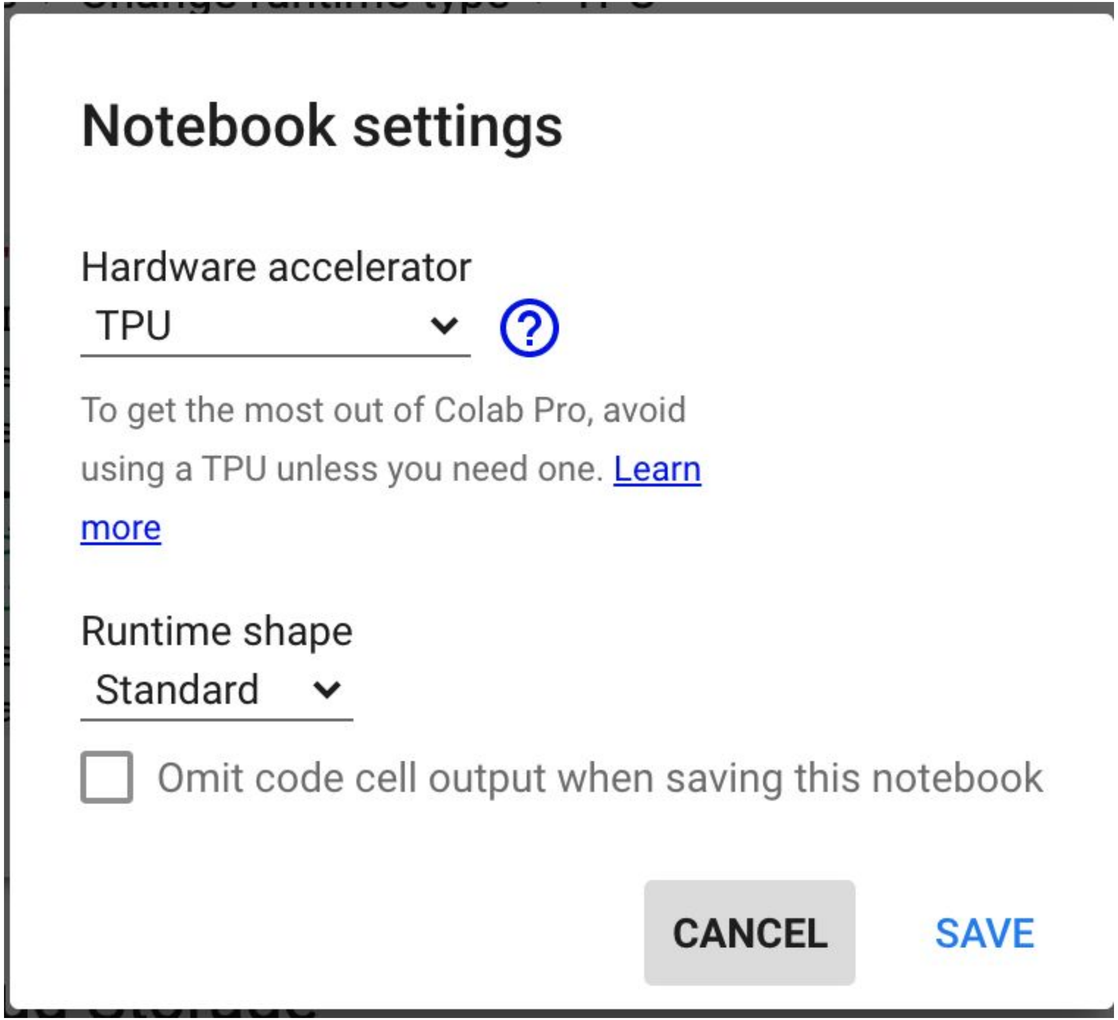
```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images, training=True)
    loss = compute_loss(labels, predictions)

  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM,
                         per_replica_losses, axis=None)
```


Notebook settings
Hardware accelerator
TPU
To get the most out of Colab Pro, avoid using a TPU unless you need one. Learn more
Runtime shape
Standard
Omit code cell output when saving this notebook
CANCEL
SAVE

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```python
# Detect hardware
try:
    tpu_address = 'grpc://' + os.environ['COLAB_TPU_ADDR']
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_address)
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
    print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])
    print("Number of accelerators: ", strategy.num_replicas_in_sync)

except ValueError:
    print('TPU failed to initialize.')
```

```
INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

Running on TPU  ['10.109.132.10:8470']
Number of accelerators:  8
```

```
INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:TPU_SYSTEM:0,
TPU_SYSTEM, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

INFO:tensorflow:*** Available Device:
_DeviceAttributes(/job:worker/replica:0/task:0/device:XLA_CPU:0,
XLA_CPU, 0, 0)

Running on TPU  ['10.109.132.10:8470']
Number of accelerators:  8
```

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
    - Use strategy.run to call your usual training function across all replicas
    - Results will be in per-replica-losses structure
    - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
    - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
    - Use strategy.run to call your usual training function across all replicas
    - Results will be in per-replica-losses structure
    - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
    - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
  - Use strategy.run to call your usual training function across all replicas
  - Results will be in per-replica-losses structure
  - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
  - Use strategy.run to call your usual testing function across all replicas

Training with TPU strategy

- Use a custom training loop
- Call the distributed training function within the loop
    - Use strategy.run to call your usual training function across all replicas
    - Results will be in per-replica-losses structure
    - Use strategy.reduce to reduce losses
- Call the distributed testing function within the loop
    - Use strategy.run to call your usual testing function across all replicas

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

```python
@tf.function
def distributed_train_step(dataset_inputs):
  per_replica_losses = strategy.run(train_step,args=(dataset_inputs,))
  return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses, axis=None)


def train_step(inputs):
  images, labels = inputs
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = compute_loss(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
  train_accuracy.update_state(labels, predictions)
  return loss
```

Training on a single device

`tf.distribute.OneDeviceStrategy`

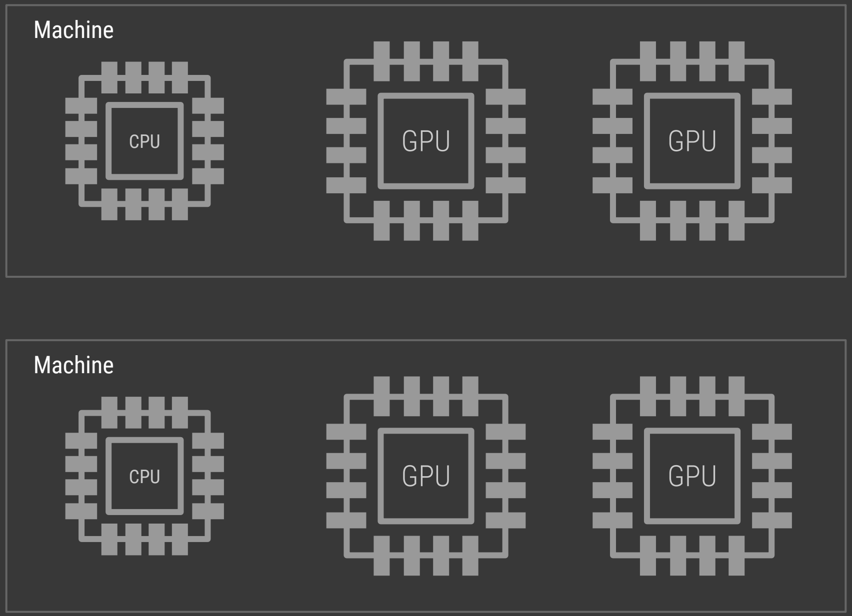
Input data is distributed

```
strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")
```

Training across many machines

`tf.distribute.experimental.MultiWorkerMirroredStrategy`

- Done across multiple workers, each with multiple GPUs
- Variables are replicated on each device across workers
- Fault tolerance with tf.keras.callbacks.ModelCheckpoint
- **Synchronization** done with CollectiveOps




```
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

Multi-worker training

- Run **workers** in a cluster
- **Tasks** (training/input pipelines)
- **Roles** (chief, worker, ps, evaluator)
- Configuring the cluster (next..)

Cluster specification

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", ...],
    },
   "task": {"type": "worker", "index": 0}
})
```

https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras

Other strategies

CentralStorageStrategy

Variables - not mirrored, but placed on CPU

Computation - replicated across local GPUs

ParameterServerStrategy

Some machines are designated as workers

... some as parameter servers

Variables - placed on one parameter server (ps)

Computation - replicated across GPUs of all the workers

Other strategies

CentralStorageStrategy

Variables - not mirrored, but placed on CPU

Computation - replicated across local GPUs

ParameterServerStrategy

Some machines are designated as workers

... some as parameter servers

Variables - placed on one parameter server (ps)

Computation - replicated across GPUs of all the workers