Running TensorFlow with GPUs

GPU is short for graphics processing unit. It is a specialized processor designed for carrying out complex computations on large memory blocks. GPUs provide more efficient processing for building deep learning models.

TensorFlow can leverage processing on multiple GPUs to speed up computation especially when training a complex network architecture. To take advantage of parallel processing, a replica of the network architecture resides on each GPU machine and trains a subset of the data. However, for synchronous updates, the model parameters from each tower (or GPU machines) are stored and updated on a CPU. It turns out that CPUs are generally good at mean or averaging processing. A diagram of this operation is shown in Figure 30-10.

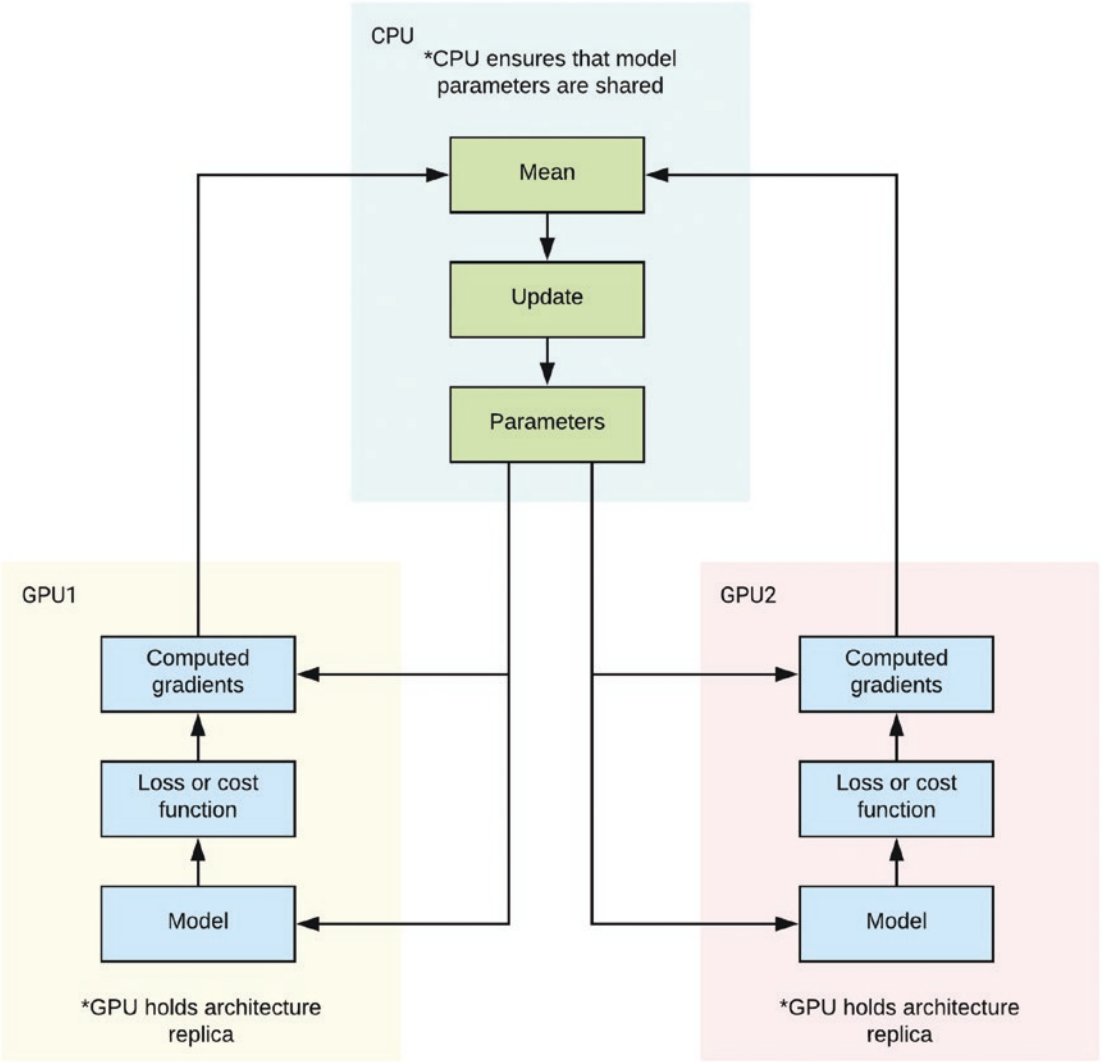

Figure 30-10. *Framework for training on multiple GPUs*

TensorFlow 2.0 performs distributed training across multiple machines (i.e., CPUs, GPUs, or TPUs) using the **'tf.distribute.Strategy'** API. To use GPUs on Google Colab, first change the runtime type to GPU and install the TensorFlow with GPU library by running the following code in the notebook cell:

```
!pip install -q tf-nightly-gpu-2.0-preview
```

The following code block uses GPUs for model training. In this example we train a simple regression model on the Boston housing dataset. The method **'tf.distribute.MirroredStrategy()'** implements a distribution strategy called MirroredStrategy. This strategy supports distributed training with multiple GPUs on a single machine. The code is similar to the previous code for linear regression with TensorFlow 2.0. However, minimal changes are added to make the components such as variables, layers, models, optimizers, metrics, summaries, and checkpoints strategy-aware using the strategy scope().

```
# import TensorFlow 2.0 with GPU
!pip install -q tf-nightly-gpu-2.0-preview

# confirm tensorflow can see GPU
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

# import other packages
import numpy as np
from tensorflow.keras.datasets import boston_housing
from tensorflow.keras import Model
from sklearn.preprocessing import StandardScaler

# load dataset and split in train and test sets
(X_train, y_train), (X_test, y_test) = boston_housing.load_data()

# standardize the dataset
scaler_X_train = StandardScaler().fit(X_train)
scaler_X_test = StandardScaler().fit(X_test)
X_train = scaler_X_train.transform(X_train)
X_test = scaler_X_test.transform(X_test)

# reshape y-data to become column vector
y_train = np.reshape(y_train, [-1, 1])
y_test = np.reshape(y_test, [-1, 1])
```

```
# build the linear model
class LinearRegressionModel(Model):
  def __init__(self):
    super(LinearRegressionModel, self).__init__()
    # initialize weight and bias variables
    self.weight = tf.Variable(
        initial_value = tf. random.normal(
            [13, 1], dtype=tf.float64),
        trainable=True)
    self.bias = tf.Variable(initial_value = tf.constant(
        1.0, shape=[], dtype=tf.float64), trainable=True)

  def call(self, inputs):
    return tf.add(tf.matmul(inputs, self.weight), self.bias)

# create a strategy to distribute the variables and the graph
strategy = tf.distribute.MirroredStrategy()

# print number of machines with GPUs
print ('Number of devices: {}'.format(strategy.num_replicas_in_sync))

# parameters
batch_size_per_replica = 32
global_batch_size = batch_size_per_replica * strategy.num_replicas_in_sync

learning_rate = 0.01

# create the distributed datasets inside a strategy.scope:
with strategy.scope():
  train_ds = tf.data.Dataset.from_tensor_slices(
      (X_train, y_train)).shuffle(len(X_train)).batch(global_batch_size)
  train_dist_ds = strategy.experimental_distribute_dataset(train_ds)

  test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test)).
  batch(global_batch_size)
  test_dist_ds = strategy.experimental_distribute_dataset(test_ds)
```

```
# define the loss function
with strategy.scope():
  # Set reduction to `none` so we can do the reduction afterwards and
  divide by
  # global batch size.
  loss_object = tf.keras.losses.MeanSquaredError(
      reduction=tf.keras.losses.Reduction.NONE)

  def compute_loss(labels, predictions):
    per_example_loss = loss_object(labels, predictions)
    return tf.reduce_sum(per_example_loss) * (1. / global_batch_size)

# define metrics to track loss and rmse
with strategy.scope():
  test_loss = tf.keras.metrics.Mean(name='test_loss')

  train_rmse = tf.keras.metrics.RootMeanSquaredError(
      name='train_rmse')
  test_rmse = tf.keras.metrics.RootMeanSquaredError(
      name='test_rmse')

# model and optimizer must be created under `strategy.scope`.
with strategy.scope():
  model = LinearRegressionModel()
  optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

with strategy.scope():
  def train_step(inputs, labels):
    with tf.GradientTape() as tape:
      predictions = model(inputs)
      loss = compute_loss(labels, predictions)

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

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

```
  def test_step(inputs, labels):
    predictions = model(inputs)
    t_loss = loss_object(labels, predictions)

    test_loss.update_state(t_loss)
    test_rmse.update_state(labels, predictions)

num_epochs = 1000

with strategy.scope():
  # `experimental_run_v2` replicates the provided computation and runs it
  # with the distributed input.
  @tf.function
  def distributed_train_step(inputs, labels):
    per_replica_losses = strategy.experimental_run_v2(train_step,
                                                      args=(inputs, labels))
    return strategy.reduce(tf.distribute.ReduceOp.SUM, per_replica_losses,
                           axis=None)

  @tf.function
  def distributed_test_step(inputs, labels):
    return strategy.experimental_run_v2(test_step, args=(inputs, labels))

  for epoch in range(num_epochs):
    # Train loop
    total_loss = 0.0
    num_batches = 0
    for train_inputs, train_labels in train_dist_ds:
      total_loss += distributed_train_step(train_inputs, train_labels)
      num_batches += 1
    train_loss = total_loss / num_batches

    # Test loop
    for test_inputs, test_labels in test_dist_ds:
      distributed_test_step(test_inputs, test_labels)

    if (epoch+1) % 100 == 0:
      template = ("Epoch {}, Loss: {}, RMSE: {}, Test Loss: {}, "
                  "Test RMSE: {}")
```

```
      print (template.format(epoch+1, train_loss,
                             train_rmse.result(), test_loss.result(),
                             test_rmse.result()))

    test_loss.reset_states()
    train_rmse.reset_states()
    test_rmse.reset_states()

'Output:'
Epoch 100, Loss: 21.673020569627965, RMSE: 4.724063396453857, Test Loss:
20.915191650390625, Test RMSE: 4.573312759399414
Epoch 200, Loss: 21.594741116702117, RMSE: 4.715524196624756, Test Loss:
20.994861602783203, Test RMSE: 4.582014560699463
Epoch 300, Loss: 21.590902259189097, RMSE: 4.7151055335998535, Test Loss:
21.02731704711914, Test RMSE: 4.585555076599121
Epoch 400, Loss: 21.59074064145569, RMSE: 4.715087413787842, Test Loss:
21.03565216064453, Test RMSE: 4.5864644050598145
Epoch 500, Loss: 21.590740279510765, RMSE: 4.715087413787842, Test Loss:
21.037595748901367, Test RMSE: 4.586676120758057
Epoch 600, Loss: 21.590742194311133, RMSE: 4.715087890625, Test Loss:
21.03803825378418, Test RMSE: 4.586724281311035
Epoch 700, Loss: 21.59074262401866, RMSE: 4.715087890625, Test Loss:
21.03813934326172, Test RMSE: 4.586735248565674
Epoch 800, Loss: 21.59074272223048, RMSE: 4.715087413787842, Test Loss:
21.038162231445312, Test RMSE: 4.586737632751465
Epoch 900, Loss: 21.59074286927267, RMSE: 4.715087413787842, Test Loss:
21.03816795349121, Test RMSE: 4.586737632751465
Epoch 1000, Loss: 21.590742907190307, RMSE: 4.715087413787842, Test Loss:
21.03816795349121, Test RMSE: 4.586738109588623
```

Please note the following from the preceding code block:

- When writing a custom training loop, sum the per example losses and divide the sum by the global batch size. In the code tf.reduce_sum(per_example_loss) * (1. / global_batch_size). This needs to be done because after calculation on each replica, the gradients are synced across the replicas by summing them. When using tf.keras.losses classes, the loss reduction needs to be explicitly specified to be one of NONE or SUM.

TensorFlow High-Level APIs: Using Estimators

In this section, we will use the high-level TensorFlow Estimator API for modeling with premade Estimators. Estimators provide another high-level API for building TensorFlow models for execution on CPUs, GPUs, or TPUs with minimal code modification.

The following steps are typically followed when working with premade Estimators:

1. Write the **'input_fn'** to handle the data pipeline.
2. Define the type of data attributes into the model using feature columns **'tf.feature_column'**.
3. Instantiate one of the premade Estimators by passing in the feature columns and other relevant attributes.
4. Use the **'train()'**, **'evaluate()'**, and **'predict()'** methods to train and evaluate the model on evaluation dataset and use the model to make prediction/inference.

Let's see a simple example of working with a TensorFlow premade Estimator again using the Boston housing dataset.

Note Reset session before running the following cells and change runtime type to None.

```
# import packages
import datetime
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.datasets import boston_housing
from tensorflow.keras import Model
from sklearn.preprocessing import StandardScaler

# load dataset and split in train and test sets
(X_train, y_train), (X_test, y_test) = boston_housing.load_data()
```