C2_W4_Lab_2_multi-GPU-mirrored-strategy

February 10, 2021

1 Multi-GPU Mirrored Strategy

In this ungraded lab, you'll go through how to set up a Multi-GPU Mirrored Strategy. The lab environment only has a CPU but we placed the code here in case you want to try this out for yourself in a multiGPU device.

Notes: - If you are running this on Coursera, you'll see it gives a warning about no presence of GPU devices. - If you are running this in Colab, make sure you have selected your `runtime` to be `GPU`. - In both these cases, you'll see there's only 1 device that is available.
- One device is sufficient for helping you understand these distribution strategies.

1.1 Imports

[ ]:
```
import tensorflow as tf
import numpy as np
import os
```

1.2 Setup Distribution Strategy

[ ]:
```
# Note that it generally has a minimum of 8 cores, but if your GPU has
# less, you need to set this. In this case one of my GPUs has 4 cores
os.environ["TF_MIN_GPU_MULTIPROCESSOR_COUNT"] = "4"

# If the list of devices is not specified in the
# `tf.distribute.MirroredStrategy` constructor, it will be auto-detected.
# If you have *different* GPUs in your system, you probably have to set up␣
↪cross_device_ops like this
strategy = tf.distribute.MirroredStrategy(cross_device_ops=tf.distribute.
↪HierarchicalCopyAllReduce())
print ('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

1.3 Prepare the Data

[ ]:
```
# Get the data
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
 ↪load_data()

# Adding a dimension to the array -> new shape == (28, 28, 1)
# We are doing this because the first layer in our model is a convolutional
# layer and it requires a 4D input (batch_size, height, width, channels).
# batch_size dimension will be added later on.
train_images = train_images[..., None]
test_images = test_images[..., None]

# Normalize the images to [0, 1] range.
train_images = train_images / np.float32(255)
test_images = test_images / np.float32(255)

# Batch the input data
BUFFER_SIZE = len(train_images)
BATCH_SIZE_PER_REPLICA = 64
GLOBAL_BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

# 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)
```

1.4 Define the Model

[ ]:
```
# Create the model architecture
def create_model():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu'),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Conv2D(64, 3, activation='relu'),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
```

```
    ])
  return model
```

1.5 Configure custom training

Instead of `model.compile()`, we're going to do custom training, so let's do that within a strategy scope.

```
[ ]: with strategy.scope():
         # We will use sparse categorical crossentropy as always. But, instead of␣
      ↪having the loss function
         # manage the map reduce across GPUs for us, we'll do it ourselves with a␣
      ↪simple algorithm.
         # Remember -- the map reduce is how the losses get aggregated
         # Set reduction to `none` so we can do the reduction afterwards and divide␣
      ↪byglobal batch size.
         loss_object = tf.keras.losses.
      ↪SparseCategoricalCrossentropy(from_logits=True, reduction=tf.keras.losses.
      ↪Reduction.NONE)

         def compute_loss(labels, predictions):
             # Compute Loss uses the loss object to compute the loss
             # Notice that per_example_loss will have an entry per GPU
             # so in this case there'll be 2 -- i.e. the loss for each replica
             per_example_loss = loss_object(labels, predictions)
             # You can print it to see it -- you'll get output like this:
             # Tensor("sparse_categorical_crossentropy/weighted_loss/Mul:0",␣
      ↪shape=(48,), dtype=float32, device=/job:localhost/replica:0/task:0/device:
      ↪GPU:0)
             # Tensor("replica_1/sparse_categorical_crossentropy/weighted_loss/Mul:
      ↪0", shape=(48,), dtype=float32, device=/job:localhost/replica:0/task:0/
      ↪device:GPU:1)
             # Note in particular that replica_0 isn't named in the weighted_loss --␣
      ↪the first is unnamed, the second is replica_1 etc
             print(per_example_loss)
             return tf.nn.compute_average_loss(per_example_loss,␣
      ↪global_batch_size=GLOBAL_BATCH_SIZE)

         # We'll just reduce by getting the average of the losses
         test_loss = tf.keras.metrics.Mean(name='test_loss')

         # Accuracy on train and test will be SparseCategoricalAccuracy
         train_accuracy = tf.keras.metrics.
      ↪SparseCategoricalAccuracy(name='train_accuracy')
```

```
    test_accuracy = tf.keras.metrics.
↪SparseCategoricalAccuracy(name='test_accuracy')

    # Optimizer will be Adam
    optimizer = tf.keras.optimizers.Adam()

    # Create the model within the scope
    model = create_model()
```

1.6 Train and Test Steps Functions

Let's define a few utilities to facilitate the training.

```
[ ]: # `run` replicates the provided computation and runs it
     # with the distributed input.
     @tf.function
     def distributed_train_step(dataset_inputs):
       per_replica_losses = strategy.run(train_step, args=(dataset_inputs,))
       #tf.print(per_replica_losses.values)
       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, 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

     #######################
     # Test Steps Functions
     #######################
     @tf.function
     def distributed_test_step(dataset_inputs):
       return strategy.run(test_step, args=(dataset_inputs,))

     def test_step(inputs):
       images, labels = inputs

       predictions = model(images, training=False)
       t_loss = loss_object(labels, predictions)
```

```
  test_loss.update_state(t_loss)
  test_accuracy.update_state(labels, predictions)
```

1.7 Training Loop

We can now start training the model.

```
[ ]: 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

  # Do Testing
  for batch in test_dist_dataset:
    distributed_test_step(batch)

  template = ("Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, " "Test␣
↪Accuracy: {}")

  print (template.format(epoch+1, train_loss, train_accuracy.result()*100,␣
↪test_loss.result(), test_accuracy.result()*100))

  test_loss.reset_states()
  train_accuracy.reset_states()
  test_accuracy.reset_states()
```