UNIVERSIDAD DE LAS FUERZAS ARMADAS “ESPE”

Hojas técnicas sobre las TPU

TPU Initialization

Los TPU generalmente están en los trabajadores de Cloud TPU que son diferentes del proceso local que ejecuta el programa python del usuario. Por lo tanto, se debe realizar un trabajo de inicialización para conectarse al clúster remoto e inicializar TPU. Tenga en cuenta que el argumento tpu para TPUClusterResolver es una dirección especial solo para Colab. En el caso de que esté ejecutando Google Compute Engine (GCE), en su lugar, debe pasar el nombre de su CloudTPU

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc
://' + os.environ['COLAB_TPU_ADDR'])
tf.config.experimental_connect_to_cluster(resolver)
# This is the TPU initialization code that has to be at the beginning.
tf.tpu.experimental.initialize_tpu_system(resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))
```

Manual device placement

Después de inicializar el TPU, puede usar la colocación manual del dispositivo para colocar el cálculo en un solo dispositivo TPU.

[ ]

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
with tf.device('/TPU:0'):
  c = tf.matmul(a, b)
print("c device: ", c.device)
print(c)
```

Distribution strategies

La mayoría de las veces los usuarios desean ejecutar el modelo en múltiples TPU de forma paralela a los datos. Una estrategia de distribución es una abstracción que se puede utilizar para conducir modelos en CPU, GPU o TPU. Simplemente cambie la estrategia de distribución y el modelo se ejecutará en el dispositivo dado. Consulte la guía de estrategia de distribución para obtener más información.

Classification on TPUs

Como hemos aprendido los conceptos básicos, es hora de mirar un ejemplo más concreto. Esta guía muestra cómo usar la estrategia de distribución `tf.distribute.experimental.TPUStrategy` conducir un Cloud TPU y entrenar un modelo Keras.

Define a Keras model

A continuación se muestra la definición del modelo MNIST usando Keras, sin cambios de lo que usaría en la CPU o GPU. Tenga en cuenta que la creación del modelo Keras debe estar dentro de strategy.scope, por lo que las variables se pueden crear en cada dispositivo TPU. No es necesario que otras partes del código estén dentro del alcance de la estrategia.

[ ]

```
def create_model():
  return tf.keras.Sequential(
      [tf.keras.layers.Conv2D(256, 3, activation='relu', input_shape=(
28, 28, 1)),
       tf.keras.layers.Conv2D(256, 3, activation='relu'),
       tf.keras.layers.Flatten(),
       tf.keras.layers.Dense(256, activation='relu'),
       tf.keras.layers.Dense(128, activation='relu'),
       tf.keras.layers.Dense(10)])
```

Input datasets

El uso eficiente de la API tf.data.Dataset es fundamental cuando se usa un TPU en la nube, ya que es imposible usar los TPU en la nube a menos que pueda alimentarlos con la suficiente rapidez.

Consulte la Guía de rendimiento de la canalización de entrada para obtener detalles sobre el rendimiento del conjunto de datos. Para todos los experimentos, excepto los más simples (usando tf.data.Dataset.from_tensor_slices u otros datos en el gráfico), necesitará almacenar todos los archivos de datos leídos por el conjunto de datos en cubos de Google Cloud Storage (GCS).

Para la mayoría de los casos de uso, se recomienda convertir sus datos al formato TFRecord y usar un tf.data.TFRecordDataset para leerlo. Vea TFRecord y tf. Tutorial de ejemplo para obtener detalles sobre cómo hacer esto. Sin embargo, esto no es un requisito difícil y puede usar otros lectores de conjuntos de datos (FixedLengthRecordDataset o TextLineDataset) si lo prefiere.

Los conjuntos de datos pequeños se pueden cargar completamente en la memoria usando tf.data.Dataset.cache. Independientemente del formato de datos utilizado, se recomienda encarecidamente que utilice archivos grandes, del orden de 100 MB. Esto es especialmente

importante en esta configuración de red, ya que la sobrecarga de abrir un archivo es significativamente mayor.

Aquí debe usar el módulo tensorflow_datasets para obtener una copia de los datos de entrenamiento de MNIST. Tenga en cuenta que try_gcs se especifica para usar una copia que está disponible en un depósito público de GCS. Si no especifica esto, el TPU no podrá acceder a los datos descargados

[ ]

```
def get_dataset(batch_size, is_training=True):
  split = 'train' if is_training else 'test'
  dataset, info = tfds.load(name='mnist', split=split, with_info=True,
                            as_supervised=True, try_gcs=True)

  def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.0

    return image, label

  dataset = dataset.map(scale)

  # Only shuffle and repeat the dataset in training. The advantage to have a
  # infinite dataset for training is to avoid the potential last partial batch
  # in each epoch, so users don't need to think about scaling the gradients
  # based on the actual batch size.
  if is_training:
    dataset = dataset.shuffle(10000)
    dataset = dataset.repeat()

  dataset = dataset.batch(batch_size)

  return dataset
```

Train a model using Keras high level APIs

Puede entrenar un modelo simplemente con las API de ajuste / compilación de Keras. Nada aquí es específico de TPU, escribiría el mismo código a continuación si tuviera varias GPU y usara una estrategia MirroredStrategy en lugar de una TPUStrategy. Para obtener más información, consulte

el tutorial de Entrenamiento distribuido con Keras

[ ]

```
with strategy.scope():
  model = create_model()
  model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(fro
m_logits=True),
                metrics=['sparse_categorical_accuracy'])

batch_size = 200
steps_per_epoch = 60000 // batch_size

train_dataset = get_dataset(batch_size, is_training=True)
test_dataset = get_dataset(batch_size, is_training=False)

model.fit(train_dataset,
          epochs=5,
          steps_per_epoch=steps_per_epoch,
          validation_data=test_dataset)
```

Train a model using custom training loop.

También puede crear y entrenar sus modelos usando las API tf.function y tf.distribute directamente. El API strategy.experimental_distribute_datasets_from_function se utiliza para distribuir el conjunto de datos dada una función de conjunto de datos. Tenga en cuenta que el tamaño de lote pasado al conjunto de datos será por tamaño de lote de réplica en lugar del tamaño de lote global en este caso. Para obtener más información, consulte el tutorial de Entrenamiento personalizado con tf.distribute.Strategy

[ ]

```
# Create the model, optimizer and metrics inside strategy scope, so th
at the
# variables can be mirrored on each device.
with strategy.scope():
  model = create_model()
  optimizer = tf.keras.optimizers.Adam()
  training_loss = tf.keras.metrics.Mean('training_loss', dtype=tf.floa
t32)
  training_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
      'training_accuracy', dtype=tf.float32)
```

```
# Calculate per replica batch size, and distribute the datasets on eac
h TPU
# worker.
per_replica_batch_size = batch_size // strategy.num_replicas_in_sync

train_dataset = strategy.experimental_distribute_datasets_from_functio
n(
    lambda _: get_dataset(per_replica_batch_size, is_training=True))

@tf.function
def train_step(iterator):
  """The step function for one training step"""

  def step_fn(inputs):
    """The computation to run on each TPU device."""
    images, labels = inputs
    with tf.GradientTape() as tape:
      logits = model(images, training=True)
      loss = tf.keras.losses.sparse_categorical_crossentropy(
          labels, logits, from_logits=True)
      loss = tf.nn.compute_average_loss(loss, global_batch_size=batch_
size)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(list(zip(grads, model.trainable_variable
s)))
    training_loss.update_state(loss * strategy.num_replicas_in_sync)
    training_accuracy.update_state(labels, logits)

  strategy.run(step_fn, args=(next(iterator),))
```

Then run the training loop.

[ ]

```
steps_per_eval = 10000 // batch_size

train_iterator = iter(train_dataset)
for epoch in range(5):
  print('Epoch: {}/5'.format(epoch))

  for step in range(steps_per_epoch):
```

```
    train_step(train_iterator)
  print('Current step: {}, training loss: {}, accuracy: {}%'.format(
      optimizer.iterations.numpy(),
      round(float(training_loss.result()), 4),
      round(float(training_accuracy.result()) * 100, 2)))
  training_loss.reset_states()
  training_accuracy.reset_states()
```

Improving performance by multiple steps within `tf.function`

El rendimiento se puede mejorar ejecutando varios pasos dentro de una función tf. Esto se logra envolviendo la llamada de estrategia.run con un tf.range dentro de tf.function, AutoGraph lo convertirá en tf.while_loop en el trabajador TPU.

Aunque con un mejor rendimiento, hay compensaciones en comparación con un solo paso dentro de tf.function. La ejecución de múltiples pasos en una función tf.f es menos flexible, no puede ejecutar cosas con entusiasmo o código python arbitrario dentro de los pasos

[ ]

```
@tf.function
def train_multiple_steps(iterator, steps):
  """The step function for one training step"""

  def step_fn(inputs):
    """The computation to run on each TPU device."""
    images, labels = inputs
    with tf.GradientTape() as tape:
      logits = model(images, training=True)
      loss = tf.keras.losses.sparse_categorical_crossentropy(
          labels, logits, from_logits=True)
      loss = tf.nn.compute_average_loss(loss, global_batch_size=batch_
size)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(list(zip(grads, model.trainable_variable
s)))
    training_loss.update_state(loss * strategy.num_replicas_in_sync)
    training_accuracy.update_state(labels, logits)

  for _ in tf.range(steps):
    strategy.run(step_fn, args=(next(iterator),))
```

```
# Convert `steps_per_epoch` to `tf.Tensor` so the `tf.function` won't
get
# retraced if the value changes.
train_multiple_steps(train_iterator, tf.convert_to_tensor(steps_per_ep
och))

print('Current step: {}, training loss: {}, accuracy: {}%'.format(
      optimizer.iterations.numpy(),
      round(float(training_loss.result()), 4),
      round(float(training_accuracy.result()) * 100, 2)))
```

Netgrafia

https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tpu.ipynb#scrollTo=yDWaRxSpwBN1

https://github.com/google-coral/edgetpu