Distributed training with Keras

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/dist

Overview

The `tf.distribute.Strategy` (/api_docs/python/tf/distribute/Strategy) API provides an abstraction for distributing your training across multiple processing units. The goal is to allow users to enable distributed training using existing models and training code, with minimal changes.

This tutorial uses the `tf.distribute.MirroredStrategy` (/api_docs/python/tf/distribute/MirroredStrategy), which does in-graph replication with synchronous training on many GPUs on one machine. Essentially, it copies all of the model's variables to each processor. Then, it uses all-reduce (http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/) to combine the gradients from all processors and applies the combined value to all copies of the model.

`MirroredStrategy` is one of several distribution strategy available in TensorFlow core. You can read about more strategies at distribution strategy guide (https://www.tensorflow.org/guide/distributed_training).

Keras API

This example uses the `tf.keras` (/api_docs/python/tf/keras) API to build the model and training loop. For custom training loops, see the tf.distribute.Strategy with training loops (/tutorials/distribute/training_loops) tutorial.

Import dependencies


```
ort TensorFlow and TensorFlow Datasets
```

```
't tensorflow_datasets as tfds
't tensorflow as tf
disable_progress_bar()

't os
```

```
:(tf.__version__)
```

```
)
```

Download the dataset

Download the MNIST dataset and load it from [TensorFlow Datasets](https://www.tensorflow.org/datasets) (https://www.tensorflow.org/datasets). This returns a dataset in [`tf.data`](/api_docs/python/tf/data) (/api_docs/python/tf/data) format.

Setting `with_info` to `True` includes the metadata for the entire dataset, which is being saved here to `info`. Among other things, this metadata object includes the number of train and test examples.

```
;ets, info = tfds.load(name='mnist', with_info=True, as_supervised=True)

:_train, mnist_test = datasets['train'], datasets['test']
```

Define distribution strategy

Create a `MirroredStrategy` object. This will handle distribution, and provides a context manager ([`tf.distribute.MirroredStrategy.scope`](/api_docs/python/tf/distribute/MirroredStrategy#scope) (/api_docs/python/tf/distribute/MirroredStrategy#scope)) to build your model inside.


```
:egy = tf.distribute.MirroredStrategy()
```

```
tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/tas

tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/tas
```

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

```
er of devices: 1
```

Setup input pipeline

When training a model with multiple GPUs, you can use the extra computing power effectively by increasing the batch size. In general, use the largest batch size that fits the GPU memory, and tune the learning rate accordingly.

```
 can also do info.splits.total_num_examples to get the total
iber of examples in the dataset.

:rain_examples = info.splits['train'].num_examples
:est_examples = info.splits['test'].num_examples

:R_SIZE = 10000

I_SIZE_PER_REPLICA = 64
I_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

Pixel values, which are 0-255, have to be normalized to the 0-1 range (https://en.wikipedia.org/wiki/Feature_scaling). Define this scale in a function.

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

:urn image, label
```

Apply this function to the training and test data, shuffle the training data, and batch it for training (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch). Notice we are also keeping an in-memory cache of the training data to improve performance.

```
ı_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SI
.dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

Create the model

Create and compile the Keras model in the context of `strategy.scope`.

```
 strategy.scope():
lel = 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)


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

Define the callbacks

The callbacks used here are:

- *TensorBoard*: This callback writes a log for TensorBoard which allows you to visualize the graphs.
- *Model Checkpoint*: This callback saves the model after every epoch.
- *Learning Rate Scheduler*: Using this callback, you can schedule the learning rate to change after every epoch/batch.

For illustrative purposes, add a print callback to display the *learning rate* in the notebook.

```
'ine the checkpoint directory to store the checkpoints

:point_dir = './training_checkpoints'
ıe of the checkpoint files
:point_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")
```

```
ıction for decaying the learning rate.
ı can define any decay function you need.
lecay(epoch):
 epoch < 3:
'eturn 1e-3
.f epoch >= 3 and epoch < 7:
'eturn 1e-4
;e:
'eturn 1e-5
```

```
.lback for printing the LR at the end of each epoch.
; PrintLR(tf.keras.callbacks.Callback):
' on_epoch_end(self, epoch, logs=None):
ırint('\nLearning rate for epoch {} is {}'.format(epoch + 1,
                                                  model.optimizer.lr.numpy()))
```

```
ıacks = [
:f.keras.callbacks.TensorBoard(log_dir='./logs'),
:f.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
                                   save_weights_only=True),
:f.keras.callbacks.LearningRateScheduler(decay),
'rintLR()
```

Train and evaluate

Now, train the model in the usual way, calling `fit` on the model and passing in the dataset created at the beginning of the tutorial. This step is the same whether you are distributing the training or not.

```
..fit(train_dataset, epochs=12, callbacks=callbacks)
```

```
ı 1/12
tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc
```

As you can see below, the checkpoints are getting saved.

```
:ck the checkpoint directory
{checkpoint_dir}
```

```
:point            ckpt_4.data-00000-of-00002
.1.data-00000-of-00002   ckpt_4.data-00001-of-00002
.1.data-00001-of-00002   ckpt_4.index
.1.index              ckpt_5.data-00000-of-00002
.10.data-00000-of-00002  ckpt_5.data-00001-of-00002
.10.data-00001-of-00002  ckpt_5.index
.10.index             ckpt_6.data-00000-of-00002
.11.data-00000-of-00002  ckpt_6.data-00001-of-00002
.11.data-00001-of-00002  ckpt_6.index
.11.index             ckpt_7.data-00000-of-00002
.12.data-00000-of-00002  ckpt_7.data-00001-of-00002
.12.data-00001-of-00002  ckpt_7.index
.12.index             ckpt_8.data-00000-of-00002
.2.data-00000-of-00002   ckpt_8.data-00001-of-00002
```

To see how the model perform, load the latest checkpoint and call `evaluate` on the test data.

Call `evaluate` as before using appropriate datasets.

```
..load_weights(tf.train.latest_checkpoint(checkpoint_dir))

.loss, eval_acc = model.evaluate(eval_dataset)

:('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

tensorflow:Reduce to /job:localhost/replica:0/task:0/device:CPU:0 then broadc

57 [==============================] - 1s 6ms/step - accuracy: 0.9859 - loss:
loss: 0.040106043219566345, Eval Accuracy: 0.9858999848365784
```

To see the output, you can download and view the TensorBoard logs at the terminal.

```
ısorboard --logdir=path/to/log-directory
```

```
-sh ./logs
```

```
. 4.0K
train
```

Export to SavedModel

Export the graph and the variables to the platform-agnostic SavedModel format. After your model is saved, you can load it with or without the scope.

```
 = 'saved_model/'
```

```
..save(path, save_format='tf')
```

```
:NG:tensorflow:From /tmpfs/src/tf_docs_env/lib/python3.6/site-packages/tensorf
'uctions for updating:
;ing Keras pass *_constraint arguments to layers.

.ng:tensorflow:From /tmpfs/src/tf_docs_env/lib/python3.6/site-packages/tensorf
'uctions for updating:
;ing Keras pass *_constraint arguments to layers.

 tensorflow:Assets written to: saved_model/assets

 tensorflow:Assets written to: saved_model/assets
```

Load the model without `strategy.scope`.

```
)licated_model = tf.keras.models.load_model(path)

)licated_model.compile(
.oss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
)ptimizer=tf.keras.optimizers.Adam(),
ıetrics=['accuracy'])

.loss, eval_acc = unreplicated_model.evaluate(eval_dataset)

:('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
 57 [==============================] - 1s 5ms/step - loss: 0.0401 - accuracy:
 loss: 0.040106043219566345, Eval Accuracy: 0.9858999848365784
```


Load the model with `strategy.scope`.

```
strategy.scope():
licated_model = tf.keras.models.load_model(path)
licated_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from
                      optimizer=tf.keras.optimizers.Adam(),
                      metrics=['accuracy'])

l_loss, eval_acc = replicated_model.evaluate(eval_dataset)
nt ('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

```
57 [==============================] - 1s 6ms/step - accuracy: 0.9859 - loss:
loss: 0.040106043219566345, Eval Accuracy: 0.9858999848365784
```

Examples and Tutorials

Here are some examples for using distribution strategy with keras fit/compile:

1. Transformer (https://github.com/tensorflow/models/blob/master/official/nlp/transformer/transformer_main.py) example trained using `tf.distribute.MirroredStrategy` (/api_docs/python/tf/distribute/MirroredStrategy)
2. NCF (https://github.com/tensorflow/models/blob/master/official/recommendation/ncf_keras_main.py) example trained using `tf.distribute.MirroredStrategy` (/api_docs/python/tf/distribute/MirroredStrategy).

More examples listed in the Distribution strategy guide (https://www.tensorflow.org/guide/distributed_training#examples_and_tutorials)

Next steps

- Read the distribution strategy guide (https://www.tensorflow.org/guide/distributed_training).

- Read the Distributed Training with Custom Training Loops (/tutorials/distribute/training_loops) tutorial.

- Visit the Performance section (https://www.tensorflow.org/guide/function) in the guide to learn more about other strategies and tools (https://www.tensorflow.org/guide/profiler) you can use to optimize the performance of your TensorFlow models.

`tf.distribute.Strategy` (/api_docs/python/tf/distribute/Strategy) is actively under development and w
ling more examples and tutorials in the near future. Please give it a try. We welcome your feedback via iss
Hub (https://github.com/tensorflow/tensorflow/issues/new).

Last updated 2020-06-12.