See playlist (https://www.youtube.com/playlist?list=PLQY2H8rRoyvxtOJfAzfs0Cq5FqBuYO8e8)

Multi-worker training with Keras


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

Overview

This tutorial demonstrates multi-worker distributed training with Keras model using `tf.distribute.Strategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) API, specifically `tf.distribute.MultiWorkerMirroredStrategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy). With the help of this strategy, a Keras model that was designed to run on single-worker can seamlessly work on multiple workers with minimal code change.

Distributed Training in TensorFlow (https://www.tensorflow.org/guide/distributed_training) guide is available for an overview of the distribution strategies TensorFlow supports for those interested in a deeper understanding of `tf.distribute.Strategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) APIs.

Setup

First, some necessary imports.

```
import json
import os
import sys
```

Before importing TensorFlow, make a few changes to the environment.

Disable all GPUs. This prevents errors caused by the workers all trying to use the same GPU. For a real application each worker would be on a different machine.

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

Reset the `TF_CONFIG` environment variable, you'll see more about this later.

```
os.environ.pop('TF_CONFIG', None)
```

Be sure that the current directory is on python's path. This allows the notebook to import the files written by `%%writefile` later.

```
if '.' not in sys.path:
  sys.path.insert(0, '.')
```

Now import TensorFlow.

```
import tensorflow as tf
```

Dataset and model definition

Next create an `mnist.py` file with a simple model and dataset setup. This python file will be used by the worker-processes in this tutorial:

```
%%writefile mnist.py

import os
import tensorflow as tf
import numpy as np

def mnist_dataset(batch_size):
  (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
  # The `x` arrays are in uint8 and have values in the range [0, 255].
  # You need to convert them to float32 with values in the range [0, 1]
  x_train = x_train / np.float32(255)
  y_train = y_train.astype(np.int64)
  train_dataset = tf.data.Dataset.from_tensor_slices(
      (x_train, y_train)).shuffle(60000).repeat().batch(batch_size)
  return train_dataset

def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
      tf.keras.Input(shape=(28, 28)),
      tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
      tf.keras.layers.Conv2D(32, 3, activation='relu'),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(128, activation='relu'),
      tf.keras.layers.Dense(10)
  ])
  model.compile(
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
```

```
      metrics=['accuracy'])
  return model
```

```
Writing mnist.py
```

Try training the model for a small number of epochs and observe the results of a single worker to make sure everything works correctly. As training progresses, the loss should drop and the accuracy should increase.

```
import mnist

batch_size = 64
single_worker_dataset = mnist.mnist_dataset(batch_size)
single_worker_model = mnist.build_and_compile_cnn_model()
single_worker_model.fit(single_worker_dataset, epochs=3, steps_per_epoch=70)
```

```
WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential
Epoch 1/3
70/70 [==============================] - 1s 13ms/step - loss: 2.2756 - accuracy: 0.1835
Epoch 2/3
70/70 [==============================] - 1s 13ms/step - loss: 2.2173 - accuracy: 0.4275
Epoch 3/3
70/70 [==============================] - 1s 13ms/step - loss: 2.1533 - accuracy: 0.5900
<tensorflow.python.keras.callbacks.History at 0x7f9b4c2545d0>
```

Multi-worker Configuration

Now let's enter the world of multi-worker training. In TensorFlow, the `TF_CONFIG` environment variable is required for training on multiple machines, each of which possibly has a different role. `TF_CONFIG` is a JSON string used to specify the cluster configuration on each worker that is part of the cluster.

Here is an example configuration:

```
tf_config = {
    'cluster': {
        'worker': ['localhost:12345', 'localhost:23456']
    },
    'task': {'type': 'worker', 'index': 0}
}
```

Here is the same `TF_CONFIG` serialized as a JSON string:

```
json.dumps(tf_config)
```

```
'{"cluster": {"worker": ["localhost:12345", "localhost:23456"]}, "task": {"type": "worker", "in
```

There are two components of `TF_CONFIG`: `cluster` and `task`.

- `cluster` is the same for all workers and provides information about the training cluster, which is a dict consisting of different types of jobs such as `worker`. In multi-worker training with `MultiWorkerMirroredStrategy`, there is usually one `worker` that takes on a little more responsibility like saving checkpoint and writing summary file for TensorBoard in addition to what a regular `worker` does. Such a worker is referred to as the `chief` worker, and it is customary that the `worker` with `index` 0 is appointed as the chief `worker` (in fact this is how `tf.distribute.Strategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) is implemented).
- `task` provides information of the current task and is different on each worker. It specifies the `type` and `index` of that worker.

In this example, you set the task `type` to `"worker"` and the task `index` to `0`. This machine is the first worker and will be appointed as the chief worker and do more work than the others. Note that other machines will need to have the `TF_CONFIG` environment variable set as well, and it should have the same `cluster` dict, but different task `type` or task `index` depending on what the roles of those machines are.

For illustration purposes, this tutorial shows how one may set a `TF_CONFIG` with 2 workers on `localhost`. In practice, users would create multiple workers on external IP addresses/ports, and set `TF_CONFIG` on each worker appropriately.

In this example you will use 2 workers, the first worker's `TF_CONFIG` is shown above. For the second worker you would set `tf_config['task']['index']=1`

Above, `tf_config` is just a local variable in python. To actually use it to configure training, this dictionary needs to be serialized as JSON, and placed in the `TF_CONFIG` environment variable.

Environment variables and subprocesses in notebooks

Subprocesses inherit environment variables from their parent. So if you set an environment variable in this `jupyter notebook` process:

```
os.environ['GREETINGS'] = 'Hello TensorFlow!'
```

You can access the environment variable from a subprocesses:

```
$ echo ${GREETINGS}
```

```
Hello TensorFlow!
```

In the next section, you'll use this to pass the `TF_CONFIG` to the worker subprocesses. You would never really launch your jobs this way, but it's sufficient for the purposes of this tutorial: To demonstrate a minimal multi-worker example.

Choose the right strategy

In TensorFlow there are two main forms of distributed training:

- Synchronous training, where the steps of training are synced across the workers and replicas, and
- Asynchronous training, where the training steps are not strictly synced.

`MultiWorkerMirroredStrategy`, which is the recommended strategy for synchronous multi-worker training, will be demonstrated in this guide. To train the model, use an instance of `tf.distribute.MultiWorkerMirroredStrategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy).

`MultiWorkerMirroredStrategy` creates copies of all variables in the model's layers on each device across all workers. It uses `CollectiveOps`, a TensorFlow op for collective communication, to aggregate gradients and keep the variables in sync. The `tf.distribute.Strategy` guide (https://www.tensorflow.org/guide/distributed_training) has more details about this strategy.

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

```
WARNING:tensorflow:Collective ops is not configured at program startup. Some performance featu
INFO:tensorflow:Single-worker MultiWorkerMirroredStrategy with local_devices = ('/device:CPU:0
```

TF_CONFIG is parsed and TensorFlow's GRPC servers are started at the time **MultiWorkerMirroredStrategy()** is called, so the **NFIG** environment variable must be set before a **tf.distribute.Strategy** ://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) instance is created. Since **TF_CONFIG** is not set yet the above stra ctively single-worker training.

`MultiWorkerMirroredStrategy` provides multiple implementations via the `CommunicationOptions` (https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CommunicationOptions) parameter. `RING` implements ring-based collectives using gRPC as the cross-host communication layer. `NCCL` uses Nvidia's NCCL (https://developer.nvidia.com/nccl) to implement collectives. `AUTO` defers the choice to the runtime. The best choice of collective implementation depends upon the number and kind of GPUs, and the network interconnect in the cluster.

Train the model

With the integration of **`tf.distribute.Strategy`** (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) API into **`tf.keras`** (https://www.tensorflow.org/api_docs/python/tf/keras), the only change you will make to distribute the training to multiple-workers is enclosing the model building and **`model.compile()`** call inside **`strategy.scope()`**. The distribution strategy's scope dictates how and where the variables are created, and in the case of **`MultiWorkerMirroredStrategy`**, the variables created are **`MirroredVariable`**s, and they are replicated on each of the workers.

```
with strategy.scope():
  # Model building/compiling need to be within `strategy.scope()`.
  multi_worker_model = mnist.build_and_compile_cnn_model()
```

```
WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential
```

Currently there is a limitation in **`MultiWorkerMirroredStrategy`** where TensorFlow ops need to be created after the instance of
gy is created. If you see **`RuntimeError: Collective ops must be configured at program startup`**, try creating the instanc
`WorkerMirroredStrategy` at the beginning of the program and put the code that may create ops after the strategy is instantiate

To actually run with **`MultiWorkerMirroredStrategy`** you'll need to run worker processes and pass a **`TF_CONFIG`**
to them.

Like the **`mnist.py`** file written earlier, here is the **`main.py`** that each of the workers will run:

```
%%writefile main.py

import os
import json

import tensorflow as tf
import mnist

per_worker_batch_size = 64
tf_config = json.loads(os.environ['TF_CONFIG'])
num_workers = len(tf_config['cluster']['worker'])

strategy = tf.distribute.MultiWorkerMirroredStrategy()

global_batch_size = per_worker_batch_size * num_workers
multi_worker_dataset = mnist.mnist_dataset(global_batch_size)

with strategy.scope():
  # Model building/compiling need to be within `strategy.scope()`.
  multi_worker_model = mnist.build_and_compile_cnn_model()
```

```
multi_worker_model.fit(multi_worker_dataset, epochs=3, steps_per_epoch=70)
```

```
Writing main.py
```

In the code snippet above note that the `global_batch_size`, which gets passed to `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch), is set to `per_worker_batch_size * num_workers`. This ensures that each worker processes batches of `per_worker_batch_size` examples regardless of the number of workers.

The current directory now contains both Python files:

```
$ ls *.py
```

```
main.py
mnist.py
```

So json-serialize the `TF_CONFIG` and add it to the environment variables:

```
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now, you can launch a worker process that will run the `main.py` and use the `TF_CONFIG`:

```
# first kill any previous runs
%killbgscripts
```

```
All background processes were killed.
```

```
$ python main.py &> job_0.log
```

There are a few things to note about the above command:

1. It uses the `%%bash` which is a notebook "magic" (https://ipython.readthedocs.io/en/stable/interactive/magics.html) to run some bash commands.
2. It uses the `--bg` flag to run the `bash` process in the background, because this worker will not terminate. It waits for all the workers before it starts.

The backgrounded worker process won't print output to this notebook, so the `&>` redirects its output to a file, so you can see what happened.

So, wait a few seconds for the process to start up:

```
import time
time.sleep(10)
```

Now look what's been output to the worker's logfile so far:

```
$ cat job_0.log
```

```
2021-05-19 11:19:32.632620: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Suc
2021-05-19 11:19:33.772965: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Suc
2021-05-19 11:19:33.780793: E tensorflow/stream_executor/cuda/cuda_driver.cc:328] failed call t
2021-05-19 11:19:33.780836: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retrievi
2021-05-19 11:19:33.780844: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostname
2021-05-19 11:19:33.780918: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:200] libcuda
2021-05-19 11:19:33.780945: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:204] kernel r
2021-05-19 11:19:33.780952: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:310] kernel v
2021-05-19 11:19:33.781503: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlo
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-05-19 11:19:33.786536: I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:301] Init
2021-05-19 11:19:33.787013: I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:411] S
```

The last line of the log file should say: `Started server with target: grpc://localhost:12345`. The first worker is now ready, and is waiting for all the other worker(s) to be ready to proceed.

So update the `tf_config` for the second worker's process to pick up:

```
tf_config['task']['index'] = 1
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now launch the second worker. This will start the training since all the workers are active (so there's no need to background this process):

```
$ python main.py
```

```
Epoch 1/3
70/70 [==============================] - 7s 59ms/step - loss: 2.2653 - accuracy: 0.1921
Epoch 2/3
70/70 [==============================] - 4s 57ms/step - loss: 2.1911 - accuracy: 0.3874
```

```
Epoch 3/3
70/70 [==============================] - 4s 57ms/step - loss: 2.1096 - accuracy: 0.5037
2021-05-19 11:19:42.747562: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Su
2021-05-19 11:19:43.954566: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Su
2021-05-19 11:19:43.962677: E tensorflow/stream_executor/cuda/cuda_driver.cc:328] failed call
2021-05-19 11:19:43.962747: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retriev
2021-05-19 11:19:43.962756: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostnam
2021-05-19 11:19:43.962851: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:200] libcuda
2021-05-19 11:19:43.962880: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:204] kernel
```

Now if you recheck the logs written by the first worker you'll see that it participated in training that model:

```
$ cat job_0.log
```

```
2021-05-19 11:19:32.632620: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Su
2021-05-19 11:19:33.772965: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Su
2021-05-19 11:19:33.780793: E tensorflow/stream_executor/cuda/cuda_driver.cc:328] failed call
2021-05-19 11:19:33.780836: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:169] retriev
2021-05-19 11:19:33.780844: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:176] hostnam
2021-05-19 11:19:33.780918: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:200] libcuda
2021-05-19 11:19:33.780945: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:204] kernel
2021-05-19 11:19:33.780952: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:310] kernel
2021-05-19 11:19:33.781503: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFl
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-05-19 11:19:33.786536: I tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:301] Ini
2021-05-19 11:19:33.787013: I tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:411]
WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential
```

Unsurprisingly this ran *slower* than the the test run at the beginning of this tutorial. Running multiple workers on a single machine only adds overhead. The goal here was not to improve the training time, but only to give an example of multi-worker training.

```
# Delete the `TF_CONFIG`, and kill any background tasks so they don't affect the next section.
os.environ.pop('TF_CONFIG', None)
%killbgscripts
```

```
All background processes were killed.
```

Multi worker training in depth

So far this tutorial has demonstrated a basic multi-worker setup. The rest of this document looks in detail other factors which may be useful or important for real use cases.

Dataset sharding

In multi-worker training, dataset sharding is needed to ensure convergence and performance.

The example in the previous section relies on the default autosharding provided by the `tf.distribute.Strategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) API. You can control the sharding by setting the `tf.data.experimental.AutoShardPolicy` (https://www.tensorflow.org/api_docs/python/tf/data/experimental/AutoShardPolicy) of the `tf.data.experimental.DistributeOptions` (https://www.tensorflow.org/api_docs/python/tf/data/experimental/DistributeOptions). To learn more about auto-sharding see the Distributed input guide (https://www.tensorflow.org/tutorials/distribute/input#sharding).

Here is a quick example of how to turn OFF the auto sharding, so each replica processes every example (not recommended):

```
options = tf.data.Options()
options.experimental_distribute.auto_shard_policy = tf.data.experimental.AutoShardPolicy.OFF

global_batch_size = 64
multi_worker_dataset = mnist.mnist_dataset(batch_size=64)
dataset_no_auto_shard = multi_worker_dataset.with_options(options)
```

Evaluation

If you pass `validation_data` into `model.fit`, it will alternate between training and evaluation for each epoch. The evaluation taking `validation_data` is distributed across the same set of workers and the evaluation results are aggregated and available for all workers. Similar to training, the validation dataset is automatically sharded at the file level. You need to set a global batch size in the validation dataset and set `validation_steps`. A repeated dataset is also recommended for evaluation.

Alternatively, you can also create another task that periodically reads checkpoints and runs the evaluation. This is what Estimator does. But this is not a recommended way to perform evaluation and thus its details are omitted.

Performance

You now have a Keras model that is all set up to run in multiple workers with `MultiWorkerMirroredStrategy`. You can try the following techniques to tweak performance of multi-worker training with `MultiWorkerMirroredStrategy`.

- `MultiWorkerMirroredStrategy` provides multiple collective communication implementations (https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/CommunicationImplementation). `RING` implements ring-based collectives using gRPC as the cross-host communication layer. `NCCL` uses Nvidia's NCCL (https://developer.nvidia.com/nccl) to implement collectives. `AUTO` defers the choice to the runtime. The best choice of collective implementation depends upon the number and kind of GPUs, and the network interconnect in the cluster. To override the automatic choice, specify `communication_options` parameter of `MultiWorkerMirroredStrategy`'s constructor, e.g.

`communication_options=tf.distribute.experimental.CommunicationOptions(implementation=tf.distribute.experimental.CollectiveCommunication.NCCL)`.

- Cast the variables to `tf.float` if possible. The official ResNet model includes an example (https://github.com/tensorflow/models/blob/8367cf6dabe11adf7628541706b660821f397dce/official/resnet/resnet_model.py#L466) of how this can be done.

Fault tolerance

In synchronous training, the cluster would fail if one of the workers fails and no failure-recovery mechanism exists. Using Keras with `tf.distribute.Strategy` (https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy) comes with the advantage of fault tolerance in cases where workers die or are otherwise unstable. You do this by preserving training state in the distributed file system of your choice, such that upon restart of the instance that previously failed or preempted, the training state is recovered.

When a worker becomes unavailable, other workers will fail (possibly after a timeout). In such cases, the unavailable worker needs to be restarted, as well as other workers that have failed.

Previously, the **`ModelCheckpoint`** callback provided a mechanism to restore training state upon restart from job failure for multi-w
g. The TensorFlow team are introducing a new **`BackupAndRestore`** (#scrollTo=kmH8uCUhfn4w) callback, to also add the support
worker training for a consistent experience, and removed fault tolerance functionality from existing **`ModelCheckpoint`** callback. F
n, applications that rely on this behavior should migrate to the new callback.

ModelCheckpoint callback

`ModelCheckpoint` callback no longer provides fault tolerance functionality, please use **`BackupAndRestore`** (#scrollTo=kmH8uCUhfn4w) callback instead.

The **`ModelCheckpoint`** callback can still be used to save checkpoints. But with this, if training was interrupted or successfully finished, in order to continue training from the checkpoint, the user is responsible to load the model manually.

Optionally the user can choose to save and restore model/weights outside **`ModelCheckpoint`** callback.

Model saving and loading

To save your model using `model.save` or `tf.saved_model.save` (https://www.tensorflow.org/api_docs/python/tf/saved_model/save), the destination for saving needs to be different for each worker. On the non-chief workers, you will need to save the model to a temporary directory, and on the chief, you will need to save to the provided model directory. The temporary directories on the worker need to be unique to prevent errors resulting from multiple workers trying to write to the same location. The model saved in all the directories are identical and typically only the model saved by the chief should be referenced for restoring or serving. You should have some cleanup logic that deletes the temporary directories created by the workers once your training has completed.

The reason you need to save on the chief and workers at the same time is because you might be aggregating variables during checkpointing which requires both the chief and workers to participate in the allreduce communication protocol. On the other hand, letting chief and workers save to the same model directory will result in errors due to contention.

With `MultiWorkerMirroredStrategy`, the program is run on every worker, and in order to know whether the current worker is chief, it takes advantage of the cluster resolver object that has attributes `task_type` and `task_id`. `task_type` tells you what the current job is (e.g. 'worker'), and `task_id` tells you the identifier of the worker. The worker with id 0 is designated as the chief worker.

In the code snippet below, `write_filepath` provides the file path to write, which depends on the worker id. In the case of chief (worker with id 0), it writes to the original file path; for others, it creates a temporary directory (with id in the directory path) to write in:

```
model_path = '/tmp/keras-model'

def _is_chief(task_type, task_id):
  # Note: there are two possible `TF_CONFIG` configuration.
  #   1) In addition to `worker` tasks, a `chief` task type is use;
  #      in this case, this function should be modified to
  #      `return task_type == 'chief'`.
  #   2) Only `worker` task type is used; in this case, worker 0 is
  #      regarded as the chief. The implementation demonstrated here
  #      is for this case.
  # For the purpose of this colab section, we also add `task_type is None`
  # case because it is effectively run with only single worker.
  return (task_type == 'worker' and task_id == 0) or task_type is None

def _get_temp_dir(dirpath, task_id):
  base_dirpath = 'workertemp_' + str(task_id)
  temp_dir = os.path.join(dirpath, base_dirpath)
  tf.io.gfile.makedirs(temp_dir)
  return temp_dir

def write_filepath(filepath, task_type, task_id):
  dirpath = os.path.dirname(filepath)
  base = os.path.basename(filepath)
  if not _is_chief(task_type, task_id):
    dirpath = _get_temp_dir(dirpath, task_id)
  return os.path.join(dirpath, base)

task_type, task_id = (strategy.cluster_resolver.task_type,
                      strategy.cluster_resolver.task_id)
write_model_path = write_filepath(model_path, task_type, task_id)
```

With that, you're now ready to save:

```
multi_worker_model.save(write_model_path)
```

```
INFO:tensorflow:Assets written to: /tmp/keras-model/assets
INFO:tensorflow:Assets written to: /tmp/keras-model/assets
```

As described above, later on the model should only be loaded from the path chief saved to, so let's remove the temporary ones the non-chief workers saved:

```
if not _is_chief(task_type, task_id):
  tf.io.gfile.rmtree(os.path.dirname(write_model_path))
```

Now, when it's time to load, let's use convenient `tf.keras.models.load_model` (https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model) API, and continue with further work. Here, assume only using single worker to load and continue training, in which case you do not call `tf.keras.models.load_model` (https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model) within another `strategy.scope()`.

```
loaded_model = tf.keras.models.load_model(model_path)

# Now that the model is restored, and can continue with the training.
loaded_model.fit(single_worker_dataset, epochs=2, steps_per_epoch=20)
```

```
Epoch 1/2
20/20 [==============================] - 1s 15ms/step - loss: 2.2944 - accuracy: 0.0789
Epoch 2/2
20/20 [==============================] - 0s 14ms/step - loss: 2.2923 - accuracy: 0.0789
<tensorflow.python.keras.callbacks.History at 0x7f9beaf34d90>
```

Checkpoint saving and restoring

On the other hand, checkpointing allows you to save model's weights and restore them without having to save the whole model. Here, you'll create one `tf.train.Checkpoint` (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint) that tracks the model, which is managed by a `tf.train.CheckpointManager` (https://www.tensorflow.org/api_docs/python/tf/train/CheckpointManager) so that only the latest checkpoint is preserved.

```
checkpoint_dir = '/tmp/ckpt'

checkpoint = tf.train.Checkpoint(model=multi_worker_model)
write_checkpoint_dir = write_filepath(checkpoint_dir, task_type, task_id)
checkpoint_manager = tf.train.CheckpointManager(
    checkpoint, directory=write_checkpoint_dir, max_to_keep=1)
```

Once the `CheckpointManager` is set up, you're now ready to save, and remove the checkpoints non-chief workers saved.

```
checkpoint_manager.save()
if not _is_chief(task_type, task_id):
  tf.io.gfile.rmtree(write_checkpoint_dir)
```

Now, when you need to restore, you can find the latest checkpoint saved using the convenient `tf.train.latest_checkpoint` (https://www.tensorflow.org/api_docs/python/tf/train/latest_checkpoint) function. After restoring the checkpoint, you can continue with training.

```
latest_checkpoint = tf.train.latest_checkpoint(checkpoint_dir)
checkpoint.restore(latest_checkpoint)
multi_worker_model.fit(multi_worker_dataset, epochs=2, steps_per_epoch=20)
```

```
Epoch 1/2
20/20 [==============================] - 3s 14ms/step - loss: 2.2972 - accuracy: 0.0867
Epoch 2/2
20/20 [==============================] - 0s 14ms/step - loss: 2.2923 - accuracy: 0.0883
<tensorflow.python.keras.callbacks.History at 0x7f9beada4850>
```

BackupAndRestore callback

BackupAndRestore callback provides fault tolerance functionality, by backing up the model and current epoch number in a temporary checkpoint file under `backup_dir` argument to `BackupAndRestore`. This is done at the end of each epoch.

Once jobs get interrupted and restart, the callback restores the last checkpoint, and training continues from the beginning of the interrupted epoch. Any partial training already done in the unfinished epoch before interruption will be thrown away, so that it doesn't affect the final model state.

To use it, provide an instance of `tf.keras.callbacks.experimental.BackupAndRestore` (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/experimental/BackupAndRestore) at the `tf.keras.Model.fit()` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) call.

With MultiWorkerMirroredStrategy, if a worker gets interrupted, the whole cluster pauses until the interrupted worker is restarted. Other workers will also restart, and the interrupted worker rejoins the cluster. Then, every worker reads the checkpoint file that was previously saved and picks up its former state, thereby allowing the cluster to get back in sync. Then the training continues.

`BackupAndRestore` callback uses `CheckpointManager` to save and restore the training state, which generates a file called checkpoint that tracks existing checkpoints together with the latest one. For this reason, `backup_dir` should not be re-used to store other checkpoints in order to avoid name collision.

Currently, **`BackupAndRestore`** callback supports single worker with no strategy, MirroredStrategy, and multi-worker with MultiWorkerMirroredStrategy. Below are two examples for both multi-worker training and single worker training.

```
# Multi-worker training with MultiWorkerMirroredStrategy.

callbacks = [tf.keras.callbacks.experimental.BackupAndRestore(backup_dir='/tmp/backup')]
with strategy.scope():
  multi_worker_model = mnist.build_and_compile_cnn_model()
multi_worker_model.fit(multi_worker_dataset,
                       epochs=3,
                       steps_per_epoch=70,
                       callbacks=callbacks)
```

```
WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential
WARNING:tensorflow:Please add `keras.layers.InputLayer` instead of `keras.Input` to Sequential
Epoch 1/3
70/70 [==============================] - 4s 14ms/step - loss: 2.2673 - accuracy: 0.1873
Epoch 2/3
70/70 [==============================] - 1s 15ms/step - loss: 2.1745 - accuracy: 0.4192
Epoch 3/3
70/70 [==============================] - 1s 14ms/step - loss: 2.0651 - accuracy: 0.5993
<tensorflow.python.keras.callbacks.History at 0x7f9beab25c10>
```

If you inspect the directory of **`backup_dir`** you specified in **`BackupAndRestore`**, you may notice some temporarily generated checkpoint files. Those files are needed for recovering the previously lost instances, and they will be removed by the library at the end of **`tf.keras.Model.fit()`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) upon successful exiting of your training.

Currently BackupAndRestore only supports eager mode. In graph mode, consider using Save/Restore Model
://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras#model_saving_and_loading) mentioned above, and by provi
`al_epoch` in **`model.fit()`**.

See also

1. Distributed Training in TensorFlow (https://www.tensorflow.org/guide/distributed_training) guide provides an overview of the available distribution strategies.
2. Official models (https://github.com/tensorflow/models/tree/master/official), many of which can be configured to run multiple distribution strategies.
3. The Performance section (https://www.tensorflow.org/guide/function) in the guide provides information about other strategies and tools (https://www.tensorflow.org/guide/profiler) you can use to optimize the performance of your TensorFlow models.

Last updated 2021-05-20 UTC.