Ungraded lab: Distributed Strategies with TF and Keras

Welcome, during this ungraded lab you are going to perform a distributed training strategy using TensorFlow and Keras, specifically the `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. In particular you will:

1. Perform training with a single worker.
2. Understand the requirements for a multi-worker setup (`tf_config` variable) and using context managers for implementing distributed strategies.
3. Use magic commands to simulate different machines.
4. Perform a multi-worker training strategy.

This notebook is based on the official Multi-worker training with Keras notebook, which covers some additional topics in case you want a deeper dive into this topic.

Distributed Training with TensorFlow guide is also available for an overview of the distribution strategies TensorFlow supports for those interested in a deeper understanding of `tf.distribute.Strategy` APIs.

Let's get started!

Setup

First, some necessary imports.

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

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.**
- Add the current directory to python's path so modules in this directory can be imported.

```
# Disable GPUs
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

# Add current directory to path
if '.' not in sys.path:
  sys.path.insert(0, '.')
```

The previous step is important since this notebook relies on writting files using the magic command `%%writefile` and then importing them as modules.

Now that the environment configuration is ready, import TensorFlow.

```
import tensorflow as tf

# Ignore warnings
tf.get_logger().setLevel('ERROR')
```

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.

The name of this file derives from the dataset you will be using which is called [mnist](#) and consists of 60,000 28x28 grayscale images of the first 10 digits.

```
%%writefile mnist.py

# import os
import tensorflow as tf
import numpy as np

def mnist_dataset(batch_size):
  # Load the data
  (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
  # Normalize pixel values for x_train and cast to float32
  x_train = x_train / np.float32(255)
  # Cast y_train to int64
  y_train = y_train.astype(np.int64)
  # Define repeated and shuffled dataset
  train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(60
  return train_dataset


def build_and_compile_cnn_model():
  # Define simple CNN model using Keras Sequential
  model = tf.keras.Sequential([
      tf.keras.layers.InputLayer(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)
  ])

  # Compile model
  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
```

Check that the file was succesfully created:

```
!ls *.py
```

```
mnist.py
```

Import the mnist module you just created and try training the model for a small number of epochs to observe the results of a single worker to make sure everything works correctly.

```
# Import your mnist model
import mnist

# Set batch size
batch_size = 64

# Load the dataset
single_worker_dataset = mnist.mnist_dataset(batch_size)

# Load compiled CNN model
single_worker_model = mnist.build_and_compile_cnn_model()

# As training progresses, the loss should drop and the accuracy should increase.
single_worker_model.fit(single_worker_dataset, epochs=3, steps_per_epoch=70)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datas
11493376/11490434 [==============================] - 0s 0us/step
Epoch 1/3
70/70 [==============================] - 3s 31ms/step - loss: 2.2864 - accurac
Epoch 2/3
70/70 [==============================] - 2s 30ms/step - loss: 2.2535 - accurac
Epoch 3/3
70/70 [==============================] - 2s 31ms/step - loss: 2.2147 - accurac
<tensorflow.python.keras.callbacks.History at 0x7fe1dee73310>
```

Everything is working as expected!

Now you will see how multiple workers can be used as a distributed strategy.

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.

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

Let's dive into how they are used:

`cluster`:

- **It 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` is implemented).

`task`:

- Provides information of the current task and is different on each worker. It specifies the `type` and `index` of that worker.

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", "index": 0}}'

Explaining the TF_CONFIG example

In this example you 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.

Since 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**.

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 instance, for the second worker you would set `tf_config['task']['index']=1`.

Quick Note on Environment variables and subprocesses in notebooks

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.

In the next section, you'll spawn new subprocesses for each worker using the `%%bash` magic command. Subprocesses inherit environment variables from their parent, so they can access `TF_CONFIG`.

You would never really launch your jobs this way (as subprocesses of an interactive Python runtime), but it's how you will do it for the purposes of this tutorial.

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 is the one you will be using.

To train the model, use an instance of `tf.distribute.MultiWorkerMirroredStrategy`.

```
strategy = 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 [official TF distributed training guide] has more details about this.

Implement Distributed Training via Context Managers

To distribute the training to multiple-workers all you need to do is to enclose 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.

```
# Implementing distributed strategy via a context manager
with strategy.scope():
  multi_worker_model = mnist.build_and_compile_cnn_model()
```

Note: `TF_CONFIG` is parsed and TensorFlow's GRPC servers are started at the time `MultiWorkerMirroredStrategy()` is called, so the `TF_CONFIG` environment variable must be set before a `tf.distribute.Strategy` instance is created.

Since `TF_CONFIG` is not set yet the above strategy is effectively single-worker training.

Train the model

Create training script

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 # Your module

# Define batch size
per_worker_batch_size = 64

# Get TF_CONFIG from the env variables and save it as JSON
tf_config = json.loads(os.environ['TF_CONFIG'])

# Infer number of workers from tf_config
num_workers = len(tf_config['cluster']['worker'])

# Define strategy
strategy = tf.distribute.MultiWorkerMirroredStrategy()

# Define global batch size
global_batch_size = per_worker_batch_size * num_workers

# Load dataset
multi_worker_dataset = mnist.mnist_dataset(global_batch_size)

# Create and compile model following the distributed strategy
with strategy.scope():
  multi_worker_model = mnist.build_and_compile_cnn_model()

# Train the model
```

```
# Train the 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`, 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 should now contain both Python files:

```
!ls *.py
```

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

Set TF_CONFIG environment variable

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

```
# Set TF_CONFIG env variable
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

And terminate all background processes:

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

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

Launch the first worker

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

```
%%bash --bg
python main.py &> job_0.log
```

```
Starting job # 0 in a separate thread.
```

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

1. It uses the `%%bash` which is a [notebook "magic"] 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:

```
# Wait for logs to be written to the file
time.sleep(10)
```

Now look what's been output to the worker's logfile so far using the `cat` command:

```
%%bash
cat job_0.log
```

```
2021-07-31 04:40:43.021073: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:40:44.424359: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:40:44.446245: E tensorflow/stream_executor/cuda/cuda_driver.cc:3
2021-07-31 04:40:44.446315: I tensorflow/stream_executor/cuda/cuda_diagnostics
2021-07-31 04:40:44.465005: I tensorflow/core/distributed_runtime/rpc/grpc_cha
2021-07-31 04:40:44.465240: I tensorflow/core/distributed_runtime/rpc/grpc_ser
```

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.

Launch the second worker

Now 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):

```
%%bash
python main.py
```

```
Epoch 1/3
70/70 [==============================] - 8s 103ms/step - loss: 2.2600 - accura
Epoch 2/3
70/70 [==============================] - 7s 104ms/step - loss: 2.1824 - accura
Epoch 3/3
70/70 [==============================] - 7s 104ms/step - loss: 2.0922 - accura
2021-07-31 04:41:01.866895: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:41:03.290199: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:41:03.301525: E tensorflow/stream_executor/cuda/cuda_driver.cc:3
2021-07-31 04:41:03.301645: I tensorflow/stream_executor/cuda/cuda_diagnostics
2021-07-31 04:41:03.305420: I tensorflow/core/distributed_runtime/rpc/grpc_cha
```

```
2021-07-31 04:41:03.305740: I tensorflow/core/distributed_runtime/rpc/grpc_ser
2021-07-31 04:41:04.324747: W tensorflow/core/grappler/optimizers/data/auto_sh
op: "TensorSliceDataset"
input: "Placeholder/_0"
input: "Placeholder/_1"
attr {
  key: "Toutput_types"
  value {
    list {
      type: DT_FLOAT
      type: DT_INT64
    }
  }
}
attr {
  key: "output_shapes"
  value {
    list {
      shape {
        dim {
          size: 28
        }
        dim {
          size: 28
        }
      }
      shape {
      }
    }
  }
}

2021-07-31 04:41:04.586118: I tensorflow/compiler/mlir/mlir_graph_optimizatior
2021-07-31 04:41:04.586574: I tensorflow/core/platform/profile_utils/cpu_utils
```

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

```
%%bash
cat job_0.log
```

```
2021-07-31 04:40:43.021073: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:40:44.424359: I tensorflow/stream_executor/platform/default/dso_
2021-07-31 04:40:44.446245: E tensorflow/stream_executor/cuda/cuda_driver.cc:3
2021-07-31 04:40:44.446315: I tensorflow/stream_executor/cuda/cuda_diagnostics
2021-07-31 04:40:44.465005: I tensorflow/core/distributed_runtime/rpc/grpc_cha
2021-07-31 04:40:44.465240: I tensorflow/core/distributed_runtime/rpc/grpc_ser
2021-07-31 04:41:04.320778: W tensorflow/core/grappler/optimizers/data/auto_sh
op: "TensorSliceDataset"
input: "Placeholder/_0"
input: "Placeholder/_1"
attr {
  key: "Toutput_types"
  value {
    list {
      type: DT_FLOAT
      type: DT_INT64
    }
```

```
    }
  }
  attr {
    key: "output_shapes"
    value {
      list {
        shape {
          dim {
            size: 28
          }
          dim {
            size: 28
          }
        }
        shape {
        }
      }
    }
  }

  2021-07-31 04:41:04.522674: I tensorflow/compiler/mlir/mlir_graph_optimizatior
  2021-07-31 04:41:04.523131: I tensorflow/core/platform/profile_utils/cpu_utils
  Epoch 1/3
  70/70 [==============================] - 8s 103ms/step - loss: 2.2600 - accura
  Epoch 2/3
  70/70 [==============================] - 7s 104ms/step - loss: 2.1824 - accura
  Epoch 3/3
  70/70 [==============================] - 7s 104ms/step - loss: 2.0922 - accura
```

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.

Congratulations on finishing this ungraded lab! Now you should have a clearer understanding of how to implement distributed strategies with Tensorflow and Keras.

Although this tutorial didn't show the true power of a distributed strategy since this will require multiple machines operating under the same network, you now know how this process looks like at a high level.

In practice and especially with very big models, distributed strategies are commonly used as they provide a way of better managing resources to perform time-consuming tasks, such as training in a fraction of the time that it will take without the strategy.

Keep it up!

✓ 0초 오후 1:41에 완료됨