C2W4_Assignment

May 30, 2021

1 Week 4 Assignment: Custom training with tf.distribute.Strategy

Welcome to the final assignment of this course! For this week, you will implement a distribution strategy to train on the Oxford Flowers 102 dataset. As the name suggests, distribution strategies allow you to setup training across multiple devices. We are just using a single device in this lab but the syntax you'll apply should also work when you have a multi-device setup. Let's begin!

1.1 Imports

[1]:
```
from __future__ import absolute_import, division, print_function,␣
 ↪unicode_literals

import tensorflow as tf
import tensorflow_hub as hub

# Helper libraries
import numpy as np
import os
from tqdm import tqdm
```

1.2 Download the dataset

[2]:
```
import tensorflow_datasets as tfds
tfds.disable_progress_bar()
```

[3]:
```
splits = ['train[:80%]', 'train[80%:90%]', 'train[90%:]']

(train_examples, validation_examples, test_examples), info = tfds.
 ↪load('oxford_flowers102', with_info=True, as_supervised=True, split =␣
 ↪splits, data_dir='data/')

num_examples = info.splits['train'].num_examples
num_classes = info.features['label'].num_classes
```

1.3 Create a strategy to distribute the variables and the graph

How does `tf.distribute.MirroredStrategy` strategy work?

- All the variables and the model graph are replicated on the replicas.
- Input is evenly distributed across the replicas.
- Each replica calculates the loss and gradients for the input it received.
- The gradients are synced across all the replicas by summing them.
- After the sync, the same update is made to the copies of the variables on each replica.

[4]:
```python
# If the list of devices is not specified in the
# `tf.distribute.MirroredStrategy` constructor, it will be auto-detected.
strategy = tf.distribute.MirroredStrategy()
```

```
WARNING:tensorflow:There are non-GPU devices in `tf.distribute.Strategy`, not
using nccl allreduce.

WARNING:tensorflow:There are non-GPU devices in `tf.distribute.Strategy`, not
using nccl allreduce.

INFO:tensorflow:Using MirroredStrategy with devices
('/job:localhost/replica:0/task:0/device:CPU:0',)

INFO:tensorflow:Using MirroredStrategy with devices
('/job:localhost/replica:0/task:0/device:CPU:0',)
```

[5]:
```python
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

```
Number of devices: 1
```

1.4 Setup input pipeline

Set some constants, including the buffer size, number of epochs, and the image size.

[6]:
```python
BUFFER_SIZE = num_examples
EPOCHS = 10
pixels = 224
MODULE_HANDLE = 'data/resnet_50_feature_vector'
IMAGE_SIZE = (pixels, pixels)
print("Using {} with input size {}".format(MODULE_HANDLE, IMAGE_SIZE))
```

```
Using data/resnet_50_feature_vector with input size (224, 224)
```

Define a function to format the image (resizes the image and scales the pixel values to range from [0,1].

[7]:
```python
def format_image(image, label):
    image = tf.image.resize(image, IMAGE_SIZE) / 255.0
    return  image, label
```

1.5 Set the global batch size (please complete this section)

Given the batch size per replica and the strategy, set the global batch size. - The global batch size is the batch size per replica times the number of replicas in the strategy.

Hint: You'll want to use the `num_replicas_in_sync` stored in the strategy.

[8]:
```python
# GRADED FUNCTION
def set_global_batch_size(batch_size_per_replica, strategy):
    '''
    Args:
        batch_size_per_replica (int) - batch size per replica
        strategy (tf.distribute.Strategy) - distribution strategy
    '''

    # set the global batch size
    ### START CODE HERE ###
    global_batch_size = batch_size_per_replica*strategy.num_replicas_in_sync
    ### END CODD HERE ###

    return global_batch_size
```

Set the GLOBAL_BATCH_SIZE with the function that you just defined

[9]:
```python
BATCH_SIZE_PER_REPLICA = 64
GLOBAL_BATCH_SIZE = set_global_batch_size(BATCH_SIZE_PER_REPLICA, strategy)

print(GLOBAL_BATCH_SIZE)
```

```
64
```

Expected Output:

```
64
```

Create the datasets using the global batch size and distribute the batches for training, validation and test batches

[10]:
```python
train_batches = train_examples.shuffle(num_examples // 4).map(format_image).
↪batch(BATCH_SIZE_PER_REPLICA).prefetch(1)
validation_batches = validation_examples.map(format_image).
↪batch(BATCH_SIZE_PER_REPLICA).prefetch(1)
test_batches = test_examples.map(format_image).batch(1)
```

1.6 Define the distributed datasets (please complete this section)

Create the distributed datasets using `experimental_distribute_dataset()` of the Strategy class and pass in the training batches. - Do the same for the validation batches and test batches.

```
[11]: # GRADED FUNCTION
def distribute_datasets(strategy, train_batches, validation_batches,␣
 ↪test_batches):

    ### START CODE HERE ###
    train_dist_dataset = strategy.experimental_distribute_dataset(train_batches)
    val_dist_dataset = strategy.
 ↪experimental_distribute_dataset(validation_batches)
    test_dist_dataset = strategy.experimental_distribute_dataset(test_batches)
    ### END CODE HERE ###

    return train_dist_dataset, val_dist_dataset, test_dist_dataset
```

Call the function that you just defined to get the distributed datasets.

```
[12]: train_dist_dataset, val_dist_dataset, test_dist_dataset =␣
 ↪distribute_datasets(strategy, train_batches, validation_batches,␣
 ↪test_batches)
```

Take a look at the type of the train_dist_dataset

```
[13]: print(type(train_dist_dataset))
print(type(val_dist_dataset))
print(type(test_dist_dataset))
```

```
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
```

Expected Output:

```
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
<class 'tensorflow.python.distribute.input_lib.DistributedDataset'>
```

Also get familiar with a single batch from the train_dist_dataset: - Each batch has 64 features and labels

```
[15]: # Take a look at a single batch from the train_dist_dataset
x = iter(train_dist_dataset).get_next()

print(f"x is a tuple that contains {len(x)} values ")
print(f"x[0] contains the features, and has shape {x[0].shape}")
print(f"  so it has {x[0].shape[0]} examples in the batch, each is an image␣
 ↪that is {x[0].shape[1:]}")
print(f"x[1] contains the labels, and has shape {x[1].shape}")
```

```
x is a tuple that contains 2 values
x[0] contains the features, and has shape (64, 224, 224, 3)
```

```
  so it has 64 examples in the batch, each is an image that is (224, 224, 3)
x[1] contains the labels, and has shape (64,)
```

1.7 Create the model

Use the Model Subclassing API to create model `ResNetModel` as a subclass of `tf.keras.Model`.

[16]:
```
class ResNetModel(tf.keras.Model):
    def __init__(self, classes):
        super(ResNetModel, self).__init__()
        self._feature_extractor = hub.KerasLayer(MODULE_HANDLE,
                                                 trainable=False)
        self._classifier = tf.keras.layers.Dense(classes, activation='softmax')

    def call(self, inputs):
        x = self._feature_extractor(inputs)
        x = self._classifier(x)
        return x
```

Create a checkpoint directory to store the checkpoints (the model's weights during training).

[17]:
```
# Create a checkpoint directory to store the checkpoints.
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
```

1.8 Define the loss function

You'll define the `loss_object` and `compute_loss` within the `strategy.scope()`. - `loss_object` will be used later to calculate the loss on the test set. - `compute_loss` will be used later to calculate the average loss on the training data.

You will be using these two loss calculations later.

[18]:
```
with strategy.scope():
    # Set reduction to `NONE` so we can do the reduction afterwards and divide␣
 ↪by
    # global batch size.
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
        reduction=tf.keras.losses.Reduction.NONE)
    # or loss_fn = tf.keras.losses.sparse_categorical_crossentropy
    def compute_loss(labels, predictions):
        per_example_loss = loss_object(labels, predictions)
        return tf.nn.compute_average_loss(per_example_loss,␣
 ↪global_batch_size=GLOBAL_BATCH_SIZE)

    test_loss = tf.keras.metrics.Mean(name='test_loss')
```

1.9 Define the metrics to track loss and accuracy

These metrics track the test loss and training and test accuracy. - You can use `.result()` to get the accumulated statistics at any time, for example, `train_accuracy.result()`.

[19]:
```
with strategy.scope():
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='train_accuracy')
    test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='test_accuracy')
```

1.10 Instantiate the model, optimizer, and checkpoints

This code is given to you. Just remember that they are created within the `strategy.scope()`. - Instantiate the ResNetModel, passing in the number of classes - Create an instance of the Adam optimizer. - Create a checkpoint for this model and its optimizer.

[20]:
```
# model and optimizer must be created under `strategy.scope`.
with strategy.scope():
    model = ResNetModel(classes=num_classes)
    optimizer = tf.keras.optimizers.Adam()
    checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
```

1.11 Training loop (please complete this section)

You will define a regular training step and test step, which could work without a distributed strategy. You can then use `strategy.run` to apply these functions in a distributed manner. - Notice that you'll define `train_step` and `test_step` inside another function `train_testp_step_fns`, which will then return these two functions.

1.11.1 Define train_step

Within the strategy's scope, define `train_step(inputs)` - `inputs` will be a tuple containing `(images, labels)`. - Create a gradient tape block. - Within the gradient tape block: - Call the model, passing in the images and setting training to be `True` (complete this part). - Call the `compute_loss` function (defined earlier) to compute the training loss (complete this part). - Use the gradient tape to calculate the gradients. - Use the optimizer to update the weights using the gradients.

1.11.2 Define test_step

Also within the strategy's scope, define `test_step(inputs)` - `inputs` is a tuple containing `(images, labels)`. - Call the model, passing in the images and set training to `False`, because the model is not going to train on the test data. (complete this part). - Use the `loss_object`, which will compute the test loss. Check `compute_loss`, defined earlier, to see what parameters to pass into `loss_object`. (complete this part). - Next, update `test_loss` (the running test loss) with the `t_loss` (the loss for the current batch). - Also update the `test_accuracy`.

[27]:
```python
# GRADED FUNCTION
def train_test_step_fns(strategy, model, compute_loss, optimizer,
 ↪train_accuracy, loss_object, test_loss, test_accuracy):
    with strategy.scope():
        def train_step(inputs):
            images, labels = inputs

            with tf.GradientTape() as tape:
                ### START CODE HERE ###
                predictions = model(images)
                loss = compute_loss(labels, predictions)
                ### END CODE HERE ###

            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

            train_accuracy.update_state(labels, predictions)
            return loss

        def test_step(inputs):
            images, labels = inputs

            ### START CODE HERE ###
            predictions = model(images, training = False)
            t_loss = compute_loss(labels, predictions)
            ### END CODE HERE ###

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

        return train_step, test_step
```

Use the `train_test_step_fns` function to produce the `train_step` and `test_step` functions.

[28]:
```python
train_step, test_step = train_test_step_fns(strategy, model, compute_loss,
 ↪optimizer, train_accuracy, loss_object, test_loss, test_accuracy)
```

1.12 Distributed training and testing (please complete this section)

The `train_step` and `test_step` could be used in a non-distributed, regular model training. To apply them in a distributed way, you'll use strategy.run.

`distributed_train_step` - Call the `run` function of the `strategy`, passing in the train step function (which you defined earlier), as well as the arguments that go in the train step function. - The run

function is defined like this `run(fn, args=() )`.
- `args` will take in the dataset inputs

`distributed_test_step` - Similar to training, the distributed test step will use the `run` function of your strategy, taking in the test step function as well as the dataset inputs that go into the test step function.

Hint:

- You saw earlier that each batch in `train_dist_dataset` is tuple with two values:
  - a batch of features
  - a batch of labels.

Let's think about how you'll want to pass in the dataset inputs into `args` by running this next cell of code:

[29]:
```
#See various ways of passing in the inputs

def fun1(args=()):
    print(f"number of arguments passed is {len(args)}")


list_of_inputs = [1,2]
print("When passing in args=list_of_inputs:")
fun1(args=list_of_inputs)
print()
print("When passing in args=(list_of_inputs)")
fun1(args=(list_of_inputs))
print()
print("When passing in args=(list_of_inputs,)")
fun1(args=(list_of_inputs,))
```

```
When passing in args=list_of_inputs:
number of arguments passed is 2

When passing in args=(list_of_inputs)
number of arguments passed is 2

When passing in args=(list_of_inputs,)
number of arguments passed is 1
```

Notice that depending on how `list_of_inputs` is passed to `args` affects whether `fun1` sees one or two positional arguments.
- If you see an error message about positional arguments when running the training code later, please come back to check how you're passing in the inputs to `run`.

Please complete the following function.

[33]:

```
def distributed_train_test_step_fns(strategy, train_step, test_step, model,␣
↪compute_loss, optimizer, train_accuracy, loss_object, test_loss,␣
↪test_accuracy):
    with strategy.scope():
        @tf.function
        def distributed_train_step(dataset_inputs):
            ### START CODE HERE ###
            per_replica_losses = strategy.run(train_step,␣
↪args=(dataset_inputs,))
            ### END CODE HERE ###
            return strategy.reduce(tf.distribute.ReduceOp.SUM,␣
↪per_replica_losses,
                                   axis=None)

        @tf.function
        def distributed_test_step(dataset_inputs):
            ### START CODE HERE ###

            ### END CODE HERE ###
            strategy.run(test_step, args=(dataset_inputs,))
            return None
            ### END CODE HERE ###

        return distributed_train_step, distributed_test_step
```

Call the function that you just defined to get the distributed train step function and distributed test step function.

[34]:
```
distributed_train_step, distributed_test_step =␣
↪distributed_train_test_step_fns(strategy, train_step, test_step, model,␣
↪compute_loss, optimizer, train_accuracy, loss_object, test_loss,␣
↪test_accuracy)
```

An important note before you continue:

The following sections will guide you through how to train your model and save it to a .zip file. These sections are **not** required for you to pass this assignment but you are encouraged to continue anyway. If you consider no more work is needed in previous sections, please submit now and carry on.

After training your model, you can download it as a .zip file and upload it back to the platform to know how well it performed. However, training your model takes around 20 minutes within the Coursera environment. Because of this, there are two methods to train your model:

Method 1

If 20 mins is too long for you, we recommend to download this notebook (after submitting it for grading) and upload it to Colab to finish the training in a GPU-enabled runtime. If you decide to do this, these are the steps to follow:

- Save this notebok.
- Click the `jupyter` logo on the upper left corner of the window. This will take you to the Jupyter workspace.
- Select this notebook (C2W4_Assignment.ipynb) and click `Shutdown`.
- Once the notebook is shutdown, you can go ahead and download it.
- Head over to Colab and select the `upload` tab and upload your notebook.
- Before running any cell go into `Runtime` –> `Change Runtime Type` and make sure that `GPU` is enabled.
- Run all of the cells in the notebook. After training, follow the rest of the instructions of the notebook to download your model.

Method 2

If you prefer to wait the 20 minutes and not leave Coursera, keep going through this notebook. Once you are done, follow these steps: - Click the `jupyter` logo on the upper left corner of the window. This will take you to the jupyter filesystem. - In the filesystem you should see a file named `mymodel.zip`. Go ahead and download it.

Independent of the method you choose, you should end up with a `mymodel.zip` file which can be uploaded for evaluation after this assignment. Once again, this is optional but we strongly encourage you to do it as it is a lot of fun.

With this out of the way, let's continue.

1.13 Run the distributed training in a loop

You'll now use a for-loop to go through the desired number of epochs and train the model in a distributed manner. In each epoch: - Loop through each distributed training set - For each training batch, call `distributed_train_step` and get the loss. - After going through all training batches, calculate the training loss as the average of the batch losses. - Loop through each batch of the distributed test set. - For each test batch, run the distributed test step. The test loss and test accuracy are updated within the test step function. - Print the epoch number, training loss, training accuracy, test loss and test accuracy. - Reset the losses and accuracies before continuing to another epoch.

[35]:
```
# Running this cell in Coursera takes around 20 mins
with strategy.scope():
    for epoch in range(EPOCHS):
        # TRAIN LOOP
        total_loss = 0.0
        num_batches = 0
        for x in tqdm(train_dist_dataset):
            total_loss += distributed_train_step(x)
            num_batches += 1
        train_loss = total_loss / num_batches

        # TEST LOOP
        for x in test_dist_dataset:
            distributed_test_step(x)
```

```
        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()
```

```
13it [01:42,  7.87s/it]
0it [00:00, ?it/s]

Epoch 1, Loss: 0.7939639687538147, Accuracy: 90.44117736816406, Test Loss:
0.029853200539946556, Test Accuracy: 57.05521774291992

9it [01:17,  8.59s/it]


       ␣
 ↪---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

        <ipython-input-35-80229b9c4a36> in <module>
          6         num_batches = 0
          7         for x in tqdm(train_dist_dataset):
    ----> 8             total_loss += distributed_train_step(x)
          9             num_batches += 1
         10         train_loss = total_loss / num_batches


        /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/
 ↪def_function.py in __call__(self, *args, **kwds)
        778       else:
        779         compiler = "nonXla"
    --> 780         result = self._call(*args, **kwds)
        781
        782       new_tracing_count = self._get_tracing_count()


        /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/
 ↪def_function.py in _call(self, *args, **kwds)
        812       # In this case we have not created variables on the first call.
 ↪ So we can
        813       # run the first trace but we should fail if variables are␣
 ↪created.
```

```
--> 814         results = self._stateful_fn(*args, **kwds)
    815         if self._created_variables:
    816           raise ValueError("Creating variables on a non-first call to␣
↪a function"


    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.
↪py in __call__(self, *args, **kwargs)
   2827     with self._lock:
   2828       graph_function, args, kwargs = self.
↪_maybe_define_function(args, kwargs)
-> 2829     return graph_function._filtered_call(args, kwargs)  # pylint:␣
↪disable=protected-access
   2830
   2831   @property


    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.
↪py in _filtered_call(self, args, kwargs, cancellation_manager)
   1846                              resource_variable_ops.
↪BaseResourceVariable))],
   1847         captured_inputs=self.captured_inputs,
-> 1848         cancellation_manager=cancellation_manager)
   1849
   1850   def _call_flat(self, args, captured_inputs,␣
↪cancellation_manager=None):


    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.
↪py in _call_flat(self, args, captured_inputs, cancellation_manager)
   1922       # No tape is watching; skip to running the function.
   1923       return self._build_call_outputs(self._inference_function.call(
-> 1924           ctx, args, cancellation_manager=cancellation_manager))
   1925     forward_backward = self._select_forward_and_backward_functions(
   1926         args,


    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.
↪py in call(self, ctx, args, cancellation_manager)
    548               inputs=args,
    549               attrs=attrs,
--> 550               ctx=ctx)
    551         else:
    552           outputs = execute.execute_with_cancellation(
```

```
    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/execute.
↪py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
         58     ctx.ensure_initialized()
         59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name,␣
↪op_name,
    ---> 60                                                 inputs, attrs, num_outputs)
         61   except core._NotOkStatusException as e:
         62     if name is not None:


    KeyboardInterrupt:
```

Things to note in the example above:

- We are iterating over the `train_dist_dataset` and `test_dist_dataset` using a `for x in ...` construct.
- The scaled loss is the return value of the `distributed_train_step`. This value is aggregated across replicas using the `tf.distribute.Strategy.reduce` call and then across batches by summing the return value of the `tf.distribute.Strategy.reduce` calls.
- `tf.keras.Metrics` should be updated inside `train_step` and `test_step` that gets executed by `tf.distribute.Strategy.experimental_run_v2`. *`tf.distribute.Strategy.experimental_run_v2` returns results from each local replica in the strategy, and there are multiple ways to consume this result. You can do `tf.distribute.Strategy.reduce` to get an aggregated value. You can also do `tf.distribute.Strategy.experimental_local_results` to get the list of values contained in the result, one per local replica.

2 Save the Model for submission (Optional)

You'll get a saved model of this trained model. You'll then need to zip that to upload it to the testing infrastructure. We provide the code to help you with that here:

2.1 Step 1: Save the model as a SavedModel

This code will save your model as a SavedModel

[ ]:
```
model_save_path = "./tmp/mymodel/1/"
tf.saved_model.save(model, model_save_path)
```

2.2 Step 2: Zip the SavedModel Directory into /mymodel.zip

This code will zip your saved model directory contents into a single file.

If you are on colab, you can use the file browser pane to the left of colab to find `mymodel.zip`. Right click on it and select 'Download'.

If the download fails because you aren't allowed to download multiple files from colab, check out the guidance here: https://ccm.net/faq/32938-google-chrome-allow-websites-to-perform-simultaneous-downloads

If you are in Coursera, follow the instructions previously provided.

It's a large file, so it might take some time to download.

[ ]:
```
import os
import zipfile

def zipdir(path, ziph):
    # ziph is zipfile handle
    for root, dirs, files in os.walk(path):
        for file in files:
            ziph.write(os.path.join(root, file))

zipf = zipfile.ZipFile('./mymodel.zip', 'w', zipfile.ZIP_DEFLATED)
zipdir('./tmp/mymodel/1/', zipf)
zipf.close()
```