Big Data Coursework - Questions

Data Processing and Machine Learning in the Cloud

This is the **INM432 Big Data coursework 2024**. This coursework contains extended elements of **theory** and **practice**, mainly around parallelisation of tasks withSpark and a bit about parallel training using TensorFlow.

Code and Report

Your tasks parallelization of tasks in PySpark, extension, evaluation, and theoretical reflection. Please complete and submit the **coding tasks** in a copy of **this notebook**. Write your code in the **indicated cells** and **include** the **output** in the submitted notebook.
Make sure that **your code contains comments** on its **stucture** and explanations of its **purpose**.

Provide also a **report** with the **textual answers in a separate document**.
Include **screenshots** from the Google Cloud web interface (don't use the SCREENSHOT function that Google provides, but take a picture of the graphs you see for the VMs) and result tables, as well as written text about the analysis.

Submission

Download and submit **your version of this notebook** as an **.ipynb** file and also submit a **shareable link** to your notebook on Colab in your report (created with the Colab 'Share' function) (**and don't change the online version after submission**).

Further, provide your **report as a PDF document**. **State the number of words** in the document at the end. The report should **not have more than 2000 words**.

Please also submit **a PDF of your Jupyter notebook**.

Introduction and Description

This coursework focuses on parallelisation and scalability in the cloud with Spark and TesorFlow/Keras. We start with code based on **lessons 3 and 4** of the *Fast and Lean Data Science* course by Martin Gorner. The course is based on Tensorflow for data processing and MachineLearning. Tensorflow's data processing approach is somewhat similar to that of Spark, but you don't need to study Tensorflow, just make sure you understand the high-level structure.
What we will do here is **parallelising pre-processing**, and **measuring** performance, and we will perform **evaluation** and **analysis** on the cloud performance, as well as **theoretical discussion**.

This coursework contains **3 sections**.

Section 0

This section just contains some necessary code for setting up the environment. It has no tasks for you (but do read the code and comments).

Section 1

Section 1 is about preprocessing a set of image files. We will work with a public dataset "Flowers" (3600 images, 5 classes). This is not a vast dataset, but it keeps the tasks more manageable for development and you can scale up later, if you like.

In '**Getting Started**' we will work through the data preprocessing code from *Fast and Lean Data Science* which uses TensorFlow's `tf.data` package. There is no task for you here, but you will need to re-use some of this code later.

In **Task 1** you will **parallelise the data preprocessing in Spark**, using Google Cloud (GC) Dataproc. This involves adapting the code from 'Getting Started' to use Spark and running it in the cloud.

Section 2

In **Section 2** we are going to **measure the speed of reading data** in the cloud. In **Task 2** we will **paralellize the measuring** of different configurations **using Spark**.

Section 3

This section is about the theoretical discussion, based on one paper, in **Task 3**. The answers should be given in the PDF report.

General points

For **all coding tasks**, take the **time of the operations** and for the cloud operations, get performance **information from the web interfaces** for your reporting and analysis.

The **tasks** are **mostly independent** of each other. The later tasks can mostly be addressed without needing the solution to the earlier ones.

Section 0: Set-up

As usual, you need to run the **imports and authentication every time you work with this notebook**. Use the **local Spark** installation for development before you send jobs to the cloud.

Read through this section once and **fill in the project ID the first time**, then you can just step straight throught this at the beginning of each session - except for the two authentication cells.

Imports

We import some **packages that will be needed throughout**. For the **code that runs in the cloud**, we will need **separate import sections** that will need to be partly different from the one below.

```
import os, sys, math
import numpy as np
import scipy as sp
import scipy.stats
import time
import datetime
import string
import random
from matplotlib import pyplot as plt
import tensorflow as tf
print("Tensorflow version " + tf.__version__)
import pickle
```

```
Tensorflow version 2.15.0
```

Cloud and Drive authentication

This is for **authenticating with with GCS Google Drive**, so that we can create and use our own buckets and access Dataproc and AI-Platform.

This section **starts with the two interactive authentications**.

First, we mount Google Drive for persistent local storage and create a directory `DB-CW` thay you can use for this work. Then we'll set up the cloud environment, including a storage bucket.

```
print('Mounting google drive...')
from google.colab import drive
drive.mount('/content/drive')
%cd "/content/drive/MyDrive"
!mkdir BD-CW
%cd "/content/drive/MyDrive/BD-CW"
```

```
Mounting google drive...
Mounted at /content/drive
/content/drive/MyDrive
mkdir: cannot create directory ‘BD-CW’: File exists
/content/drive/MyDrive/BD-CW
```

Next, we authenticate with the GCS to enable access to Dataproc and AI-Platform.

```
import sys
if 'google.colab' in sys.modules:
    from google.colab import auth
    auth.authenticate_user()
```

It is useful to **create a new Google Cloud project** for this coursework. You can do this on the GC Console page by clicking on the entry at the top, right of the *Google Cloud Platform* and choosing *New Project*. **Copy** the **generated project ID** to the next cell. Also **enable billing** and the **Compute, Storage and Dataproc** APIs like we did during the labs.

We also specify the **default project and region**. The REGION should be `us-central1` as that seems to be the only one that reliably works with the free credit. This way we don't have to specify this information every time we access the cloud.

```
PROJECT = 'big-data-cw-420116'  ### USE YOUR GOOGLE CLOUD PROJECT ID HERE. ### (DONE!)
!gcloud config set project $PROJECT
REGION = 'us-central1'
CLUSTER = '{}-cluster'.format(PROJECT)
!gcloud config set compute/region $REGION
!gcloud config set dataproc/region $REGION

!gcloud config list # show some information
```

```
Updated property [core/project].
WARNING: Property validation for compute/region was skipped.
Updated property [compute/region].
Updated property [dataproc/region].
[component_manager]
disable_update_check = True
[compute]
region = us-central1
[core]
account = antonio-jose.lopez-roldan@city.ac.uk
project = big-data-cw-420116
[dataproc]
region = us-central1

Your active configuration is: [default]
```

With the cell below, we **create a storage bucket** that we will use later for **global storage**. If the bucket exists you will see a "ServiceException: 409 ...", which does not cause any problems. **You must create your own bucket to have write access.**

```
BUCKET = 'gs://{}-storage'.format(PROJECT)
!gsutil mb $BUCKET
```

```
Creating gs://big-data-cw-420116-storage/...
ServiceException: 409 A Cloud Storage bucket named 'big-data-cw-420116-storage' already exists. Try another name. Bucket names must be g
```

The cell below just **defines some routines for displaying images** that will be **used later**. You can see the code by double-clicking, but you don't need to study this.

Utility functions for image display **[RUN THIS TO ACTIVATE]**

Show code

```
# Check list of services working and enabled
!gcloud services list --enabled
```

```
NAME                                TITLE
analyticshub.googleapis.com         Analytics Hub API
artifactregistry.googleapis.com     Artifact Registry API
autoscaling.googleapis.com          Cloud Autoscaling API
bigquery.googleapis.com             BigQuery API
bigqueryconnection.googleapis.com   BigQuery Connection API
bigquerydatapolicy.googleapis.com   BigQuery Data Policy API
bigquerymigration.googleapis.com    BigQuery Migration API
bigqueryreservation.googleapis.com  BigQuery Reservation API
bigquerystorage.googleapis.com      BigQuery Storage API
cloudapis.googleapis.com            Google Cloud APIs
cloudresourcemanager.googleapis.com Cloud Resource Manager API
cloudtrace.googleapis.com           Cloud Trace API
compute.googleapis.com              Compute Engine API
container.googleapis.com            Kubernetes Engine API
containerfilesystem.googleapis.com  Container File System API
containerregistry.googleapis.com    Container Registry API
dataform.googleapis.com             Dataform API
dataplex.googleapis.com             Cloud Dataplex API
dataproc-control.googleapis.com     Cloud Dataproc Control API
dataproc.googleapis.com             Cloud Dataproc API
datastore.googleapis.com            Cloud Datastore API
dns.googleapis.com                  Cloud DNS API
gkebackup.googleapis.com            Backup for GKE API
iam.googleapis.com                  Identity and Access Management (IAM) API
iamcredentials.googleapis.com       IAM Service Account Credentials API
```

```
logging.googleapis.com              Cloud Logging API
monitoring.googleapis.com           Cloud Monitoring API
networkconnectivity.googleapis.com  Network Connectivity API
oslogin.googleapis.com              Cloud OS Login API
pubsub.googleapis.com               Cloud Pub/Sub API
servicemanagement.googleapis.com    Service Management API
serviceusage.googleapis.com         Service Usage API
sql-component.googleapis.com        Cloud SQL
storage-api.googleapis.com          Google Cloud Storage JSON API
storage-component.googleapis.com    Cloud Storage
storage.googleapis.com              Cloud Storage API
```

Install Spark locally for quick testing

You can use the cell below to **install Spark locally on this Colab VM** (like in the labs), to do quicker small-scale interactive testing. Using Spark in the cloud with **Dataproc is still required for the final version**.

[ ] ↳ *1 cell hidden*

Section 1: Data pre-processing

This section is about the **pre-processing of a dataset** for deep learning. We first look at a ready-made solution using Tensorflow and then we build a implement the same process with Spark. The tasks are about **parallelisation** and **analysis** the performance of the cloud implementations.

1.1 Getting started

In this section, we get started with the data pre-processing. The code is based on lecture 3 of the 'Fast and Lean Data Science' course.

This code is using the TensorFlow `tf.data` package, which supports map functions, similar to Spark. Your **task** will be to **re-implement the same approach in Spark**.

[ ] ↳ *12 cells hidden*

1.2 Improving Speed

Using individual image files didn't look very fast. The 'Lean and Fast Data Science' course introduced **two techniques to improve the speed**.

Recompress the images

By **compressing** the images in the **reduced resolution** we save on the size. This **costs some CPU time** upfront, but **saves network and disk bandwith**, especially when the data are **read multiple times**.

```
# This is a quick test to get an idea how long recompressions takes.
dataset4 = dsetResized.map(recompress_image)
test_set = dataset4.batch(10).take(10)
for image, label in test_set:
    print("Image batch shape {}, {})".format(image.numpy().shape, [lbl.decode('utf8') for lbl in label.numpy()]))
```

```
Image batch shape (10,), ['sunflowers', 'tulips', 'tulips', 'tulips', 'daisy', 'sunflowers', 'dandelion', 'roses', 'daisy', 'tulips'])
Image batch shape (10,), ['daisy', 'roses', 'roses', 'tulips', 'dandelion', 'tulips', 'daisy', 'roses', 'tulips', 'dandelion'])
Image batch shape (10,), ['dandelion', 'sunflowers', 'daisy', 'roses', 'sunflowers', 'dandelion', 'tulips', 'tulips', 'daisy', 'roses'])
Image batch shape (10,), ['daisy', 'tulips', 'dandelion', 'dandelion', 'roses', 'roses', 'daisy', 'roses', 'daisy', 'sunflowers'])
Image batch shape (10,), ['tulips', 'roses', 'daisy', 'daisy', 'dandelion', 'dandelion', 'dandelion', 'daisy', 'dandelion', 'dandelion']
Image batch shape (10,), ['tulips', 'dandelion', 'roses', 'daisy', 'tulips', 'sunflowers', 'dandelion', 'dandelion', 'roses', 'tulips'])
Image batch shape (10,), ['daisy', 'tulips', 'daisy', 'dandelion', 'daisy', 'daisy', 'tulips', 'tulips', 'sunflowers', 'roses'])
Image batch shape (10,), ['daisy', 'roses', 'sunflowers', 'tulips', 'dandelion', 'dandelion', 'dandelion', 'dandelion', 'dandelion', 'da
Image batch shape (10,), ['dandelion', 'sunflowers', 'dandelion', 'roses', 'roses', 'roses', 'daisy', 'dandelion', 'dandelion', 'sunflow
Image batch shape (10,), ['tulips', 'daisy', 'dandelion', 'dandelion', 'dandelion', 'daisy', 'tulips', 'sunflowers', 'sunflowers', 'rose
```

Write the dataset to TFRecord files

By writing **multiple preprocessed samples into a single file**, we can make further speed gains. We distribute the data over **partitions** to facilitate **parallelisation** when the data are used. First we need to **define a location** where we want to put the file.

```
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names
```

Now we can **write the TFRecord files** to the bucket.

Running the cell takes some time and **only needs to be done once** or not at all, as you can use the publicly available data for the next few cells. For convenience I have commented out the call to `write_tfrecords` at the end of the next cell. You don't need to run it (it takes some time), but you'll need to use the code below later (but there is no need to study it in detail).

There is a **ready-made pre-processed data** versions available here: `gs://flowers-public/tfrecords-jpeg-192x192-2/`, that we can use for testing.

```
# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints): # int64
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(tfrec_filewriter, img_bytes, label): # Create tf data records
    class_num = np.argmax(np.array(CLASSES)==label) # 'roses' => 2 (order defined in CLASSES)
    one_hot_class = np.eye(len(CLASSES))[class_num]     # [0, 0, 1, 0, 0] for class #2, roses
    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "class": _int_feature([class_num]) #,      # one class in the list
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

def write_tfrecords(GCS_PATTERN,GCS_OUTPUT,partition_size): # write the images to files.
    print("Writing TFRecords")
    tt0 = time.time()
    filenames = tf.data.Dataset.list_files(GCS_PATTERN)
    dataset1 = filenames.map(decode_jpeg_and_label)
    dataset2 = dataset1.map(resize_and_crop_image)
    dataset3 = dataset2.map(recompress_image)
    dataset4 = dataset3.batch(partition_size) # partitioning: there will be one "batch" of images per file
    for partition, (image, label) in enumerate(dataset4):
        # batch size used as partition size here
        partition_size = image.numpy().shape[0]
        # good practice to have the number of records in the filename
        filename = GCS_OUTPUT + "{:02d}-{}.tfrec".format(partition, partition_size)
        # You need to change GCS_OUTPUT to your own bucket to actually create new files
        with tf.io.TFRecordWriter(filename) as out_file:
            for i in range(partition_size):
                example = to_tfrecord(out_file,
                                      image.numpy()[i], # re-compressed image: already a byte string
                                      label.numpy()[i] #
                                      )
                out_file.write(example.SerializeToString())
        print("Wrote file {} containing {} records".format(filename, partition_size))
    print("Total time: "+str(time.time()-tt0))

write_tfrecords(GCS_PATTERN,GCS_OUTPUT,partition_size) # uncomment to run this cell
```

```
Writing TFRecords
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers00-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers01-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers02-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers03-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers04-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers05-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers06-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers07-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers08-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers09-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers10-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers11-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers12-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers13-230.tfrec containing 230 records
```

```
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers14-230.tfrec containing 230 records
Wrote file gs://big-data-cw-420116-storage/tfrecords-jpeg-192x192-2/flowers15-220.tfrec containing 220 records
Total time: 223.18424677848816
```

Test the TFRecord files

We can now **read from the TFRecord files**. By default, we use the files in the public bucket. Comment out the 1st line of the cell below to use the files written in the cell above.

```
 1
 2 def read_tfrecord(example):
 3     features = {
 4         "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
 5         "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
 6     }
 7     # decode the TFRecord
 8     example = tf.io.parse_single_example(example, features)
 9     image = tf.image.decode_jpeg(example['image'], channels=3)
10     image = tf.reshape(image, [*TARGET_SIZE, 3])
11     class_num = example['class']
12     return image, class_num
13
14 def load_dataset(filenames):
15     # read from TFRecords. For optimal performance, read from multiple
16     # TFRecord files at once and set the option experimental_deterministic = False
17     # to allow order-altering optimizations.
18     option_no_order = tf.data.Options()
19     option_no_order.experimental_deterministic = False
20
21     dataset = tf.data.TFRecordDataset(filenames)
22     dataset = dataset.with_options(option_no_order)
23     dataset = dataset.map(read_tfrecord)
24     return dataset
25
26 filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
27 datasetTfrec = load_dataset(filenames)
```

Let's have a look **if reading from the TFRecord** files is **quicker**.

```
1 batched_dataset = datasetTfrec.batch(10)
2 sample_set = batched_dataset.take(10)
3 for image, label in sample_set:
4     print("Image batch shape {}, {})".format(image.numpy().shape, \
5                         [str(lbl) for lbl in label.numpy()]))
```

```
Image batch shape (10, 192, 192, 3), ['1', '0', '4', '4', '1', '3', '2', '1', '4', '2'])
Image batch shape (10, 192, 192, 3), ['1', '0', '4', '1', '2', '1', '1', '4', '0', '1'])
Image batch shape (10, 192, 192, 3), ['1', '2', '3', '1', '0', '1', '2', '4', '4', '3'])
Image batch shape (10, 192, 192, 3), ['1', '3', '1', '4', '2', '3', '0', '3', '0', '3'])
Image batch shape (10, 192, 192, 3), ['3', '3', '3', '4', '3', '0', '1', '0', '4', '2'])
Image batch shape (10, 192, 192, 3), ['2', '4', '1', '3', '0', '3', '4', '3', '3', '1'])
Image batch shape (10, 192, 192, 3), ['0', '4', '3', '4', '0', '4', '3', '3', '0', '3'])
Image batch shape (10, 192, 192, 3), ['0', '0', '1', '0', '2', '4', '2', '2', '3', '0'])
Image batch shape (10, 192, 192, 3), ['2', '1', '1', '4', '1', '2', '3', '3', '0', '3'])
Image batch shape (10, 192, 192, 3), ['0', '4', '0', '4', '4', '4', '1', '1', '1', '1'])
```

Wow, we have a **massive speed-up**! The repackageing is worthwhile :-)

Task 1: Write TFRecord files to the cloud with Spark (40%)

Since recompressing and repackaging is very effective, we would like to be able to do it inparallel for large datasets. This is a relatively straightforward case of **parallelisation**. We will **use Spark to implement** the same process as above, but in parallel.

1a) Create the script (14%)

Re-implement the pre-processing in Spark, using Spark mechanisms for **distributing** the workload **over multiple machines**.

You need to:

i) **Copy** over the **mapping functions** (see section 1.1) and **adapt** the resizing and recompression functions **to Spark** (only one argument). (3%)

ii) **Replace** the TensorFlow **Dataset objects with RDDs**, starting with an RDD that contains the list of image filenames. (3%)

iii) **Sample** the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor of 0.02 for short tests. (1%)

iv) Then **use the functions from above** to write the TFRecord files. (3%)

v) The code for **writing to the TFRecord files** needs to be put into a function, that can be applied to every partition with the 'RDD.mapPartitionsWithIndex' function. The return value of that function is not used here, but you should return the filename, so that you have a list of the created TFRecord files. (4%)

```
                                ### CODING TASK ### ("Done & Working")
# Setting the working environment
import argparse
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
#import scipy as sp
#import scipy.stats
import time
import tensorflow as tf

#from matplotlib import pyplot as plt
from pyspark.sql import SQLContext
from pyspark.sql import Row

# define variables
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
PROJECT = 'big-data-cw-420116' # my project id
BUCKET = 'gs://{}-storage'.format(PROJECT) # my own bucket storage
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket

#i) Copy over the mapping functions (see section 1.1) and adapt the resizing and recompression functions to Spark (only one argument). (3
def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

def recompress_image(image, label):
    # this reduces the amount of data, but takes some time
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)
    return image, label

# ii) Replace the TensorFlow Dataset objects with RDDs, starting with an RDD that contains the list of image filenames. (3%)

filenames = tf.io.gfile.glob(GCS_PATTERN)

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Create RDD for files
filenames_rdd = sc.parallelize(filenames)

# iii) Sample the the RDD to a smaller number at an appropriate position in the code. Specify a sampling factor of 0.02 for short tests.

# Sampling the rdd
```

```
sample_rdd = filenames_rdd.sample(False, 0.02)

#Adapting tensorflow to spark, rdd for decode jpeg and label, rdd for resize and crop image, rdd for recompress image
decode_jpeg_and_label_rdd = filenames_rdd.map(decode_jpeg_and_label)
resize_and_crop_image_rdd = decode_jpeg_and_label_rdd.map(lambda x: resize_and_crop_image(x[0],x[1]))
recompress_image_rdd = resize_and_crop_image_rdd.map(lambda x: recompress_image(x[0],x[1]))

# iv) Then use the functions from above to write the TFRecord files, using an RDD as the vehicle for parallelisation but not for storing

# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints): # int64
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(tfrec_filewriter, img_bytes, label): # Create tf data records
    class_num = np.argmax(np.array(CLASSES)==label) # 'roses' => 2 (order defined in CLASSES)
    one_hot_class = np.eye(len(CLASSES))[class_num]     # [0, 0, 1, 0, 0] for class #2, roses
    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "class": _int_feature([class_num]) #,      # one class in the list
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

def write_tfrecords(partition_index,partition):  # write the images to files.
  # tt0 = time.time()
  # good practice to have the number of records in the filename
  filename = GCS_OUTPUT + "{}.tfrec".format(partition_index)
  # (CHANGED) You need to change GCS_OUTPUT to your own bucket to actually create new files
  with tf.io.TFRecordWriter(filename) as out_file:
    for element in partition:
      image=element[0]
      label=element[1]
      example = to_tfrecord(out_file,
                            image.numpy(), # re-compressed image: already a byte string
                            label.numpy() #
                            )
      out_file.write(example.SerializeToString())
              #print("Wrote file {} containing {} records".format(filename, partition_size))
    #print("Total time: "+str(time.time()-tt0))
  return (filename)

# v) The code for writing to the TFRecord files needs to be put into a function, that can be applied to every partition with the 'RDD.map
#    The return value of that function is not used here, but you should return the filename, so that you have a list of the created TFRec

partitions_rdd = recompress_image_rdd.repartition(PARTITIONS)
filenames_rdd = recompress_image_rdd.mapPartitionsWithIndex(write_tfrecords)
print("Writing TFRecords")

# Extra resources checked:
# https://sparkbyexamples.com/pyspark-rdd/
# https://spark.apache.org/docs/latest/rdd-programming-guide.html
# https://stackoverflow.com/questions/69205589/how-preprocess-image-using-pyspark
# https://www.tensorflow.org/guide/distributed_training#sparkstrategy
# https://www.w3schools.com/python/python_lambda.asp
# https://pages.databricks.com/rs/094-YMS-629/images/LearningSpark2.0.pdf
```

```
Writing TFRecords
```

```
# First elements of the RDD
decode_jpeg_and_label_rdd.take(1)
```

```
[(<tf.Tensor: shape=(263, 320, 3), dtype=uint8, numpy=
  array([[[133, 135, 132],
          [136, 138, 135],
          [140, 142, 139],
          ...,
          [152, 152, 150],
          [155, 155, 153],
          [148, 148, 146]],

         [[133, 135, 132],
          [136, 138, 135],
          [140, 142, 139],
          ...,
```

```
                [153, 153, 151],
                [155, 155, 153],
                [147, 147, 145]],

               [[132, 134, 129],
                [135, 137, 134],
                [139, 141, 138],
                ...,
                [152, 152, 150],
                [154, 154, 152],
                [146, 146, 144]],

               ...,

               [[ 44,  48,  25],
                [ 44,  48,  25],
                [ 44,  48,  25],
                ...,
                [127, 126, 122],
                [127, 126, 122],
                [127, 126, 122]],

               [[ 44,  48,  25],
                [ 44,  48,  25],
                [ 44,  48,  25],
                ...,
                [128, 127, 123],
                [128, 127, 123],
                [128, 127, 123]],

               [[ 43,  47,  24],
                [ 43,  47,  24],
                [ 43,  47,  24],
                ...,
                [129, 128, 124],
                [129, 128, 124],
                [130, 129, 125]]], dtype=uint8)>,
      <tf.Tensor: shape=(), dtype=string, numpy=b'daisy'>)]
```

```
1 # first elements of the RDD
2 resize_and_crop_image_rdd.take(1)
```

```
    [(<tf.Tensor: shape=(192, 192, 3), dtype=float32, numpy=
      array([[[154.31648 , 158.31648 , 161.31648 ],
              [154.03465 , 158.03465 , 161.03465 ],
              [152.40732 , 156.40732 , 159.40732 ],
              ...,
              [166.26291 , 167.93884 , 169.71353 ],
              [164.40128 , 168.40128 , 169.40128 ],
              [164.      , 168.      , 169.      ]],

             [[168.3533  , 172.16167 , 175.54494 ],
              [166.39734 , 170.39734 , 173.39734 ],
              [163.65054 , 167.65054 , 170.65054 ],
              ...,
              [165.4297  , 167.10564 , 168.88033 ],
              [164.40128 , 168.40128 , 169.40128 ],
              [164.      , 168.      , 169.      ]],

             [[164.95058 , 168.      , 172.90114 ],
              [163.81895 , 167.81895 , 170.81895 ],
              [162.34184 , 166.34184 , 169.34184 ],
              ...,
              [166.90747 , 167.95851 , 169.9415  ],
              [166.25023 , 167.47679 , 169.40128 ],
              [165.84895 , 167.07552 , 169.      ]],

             ...,

             [[120.384514, 118.384514, 119.384514],
              [123.10339 , 121.159195, 122.131294],
              [124.0755  , 124.0755  , 124.878075],
              ...,
              [127.89167 , 127.34229 , 124.66636 ],
              [126.44648 , 127.44648 , 122.44648 ],
              [126.97421 , 127.97421 , 122.97421 ]],

             [[121.46287 , 119.46287 , 120.46287 ],
              [124.460785, 122.51658 , 123.48868 ],
              [124.8213  , 124.8213  , 125.62388 ],
              ...,
              [129.9137  , 129.46465 , 126.78871 ],
              [126.821304, 128.      , 123.      ],
```

```
            [127.      , 128.      , 123.      ]],

           [[122.18799 , 120.18799 , 121.18799 ],
            [124.977264, 123.03305 , 124.00516 ],
            [123.783615, 123.783615, 124.5862  ],
            ...,
            [131.46323 , 131.13916 , 128.46323 ],
            [126.925804, 128.8151  , 123.815094],
            [127.      , 128.02274 , 123.022736]]], dtype=float32)>,
     <tf.Tensor: shape=(), dtype=string, numpy=b'daisy'>)]
```

```
1 # First elements of the RDD
2 recompress_image_rdd.take(1)
```

```
[(<tf.Tensor: shape=(), dtype=string,
numpy=b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x01,\x01,\x00\x00\xff\xdb\x00C\x00\x02\x01\x01\x01\x01\x01\x02\x01\x01\x01\x02\x
\x00\xfd5\xbc:l\xbfg\xfbK[)\xaa\xa3\x1d\xfb@;\x8f\xe9\x91\xf9cM9\x96)\x00r\xa2+6F\xd9e+\x1d\x03\xd1\xb6\x7fq\xb9(\xef\xf2n\xff\x00\xd
c\xda\x97\xad\xb1X\x97\x8bd)_N\xa0\xf3Z`y\x05>A_?
\xe6>\xba\x10\x89\xd1\xba\xc2\x92A]\xb7\x15\x8e\x92\xedo[\xb5\xa2Q,N2\x08\x1d\xd4\xfc\x8f\xd4j\xb2g*\x1b\x81I\r\n\xb2\x1cH0\xf1v`}\xd
\xf6\x98\xc1Pb\x0b\x8eC\xf7|gJ=\xd96\xac\x17]\xb9C\xf6\xa8\x1aR0Q\xc0\xe4?
\xb6\x8d\x05l*\x1c/(\xeb\x9d9\x92"Ja\x91\xbb\xfduw\xb8\xaaU)\xc9\xe3\rP"9\xc6y\x03\xee>\x9aQ\xce\xf7"\x80i.\xdd5\xeb%\x17\xd9\xd8\x10
Kc\x07Khi{r\\\xfe\'N@\x08\x88!\xbb(m\xedsx\xa2g\x07\xca\x80\x07\xd7R\xf7P\xc2\x86\x82\xb2\x1b\xbc\x8a\xd5/\x17\x12\xceX\xfa\x87K;!\x1
X\xdb@\xc7\xef\x01\xfd\xf4\x8c\xa0\xf6R\xbcm\xbd\xd1u\xda\x97\x11]1\x9c\xa9=\x99s\xf0\xba\xff\x00\t\xf9\xfb\xe8qN\xe6\x11\x95\x05k\x9
nJq\xd8\x8d\nRM\x95\x00R\'f*Esk{7\xc3P\x84\x00|\x07\x1d\xc7\xfa\x8f\xcfQ\xa5\x97k\xe8\xa9Ln\xb4,\xa9\x95\x18*pA\xf9i\x99\r\x1bU\xabR\
@w\x990\xa0\xfe\x1a\x1e\xf5e\xcaA
\x94,hr\x17\xe1\\\xe8w}\xd7&\x14\x14\xce"\x12N\x01o\x90\xd1ZqJ\x08\xb2\x98\xc6\xe8\xa7\xb0\xfcs\xa30P\\\x05(\x8b}\xd6\xb2\xdd\xc6XKI\
s\xb3p\xcac\xb1S\xdb\x1aZAH\xa1!\xa2\xdckAX)%\x9f\xd1Wo\x86F\x19U?\xcd\xf3_\x9f\xd3\xbf\xb6\x96~M.<#\xea\xd2*\xd4z\x88#
(c8\x9a\x12Fco\x97\xd4\x1f
\xfc\xb5\x9c\xf1\xb4\xa9D\xec\xdd\xe5_\xb2o\xf1\xdc\xe92T\x10\'\x84\xf8u\xf7\xfc\xfeZ\x985\x067\xd8\xe1q\x00\x85\xb8\xcd\xbb\xf6\xadU
\x83\xfd\xb4f.A\xde\xef\xd1\xc0\x1a\x9a\x9c\x8e\xfd\xd9\xbd\xb4n\xd8\\\x92\x8bl\xf4\xbd\xc2\xfe\xf6Y\x08\xf6\xd5\xa8\xa8\xdc\x17\x89)
(\xcap%\x1e\xe3\xf1\xd5ZK\x0e\x17*\x07\xb9Gw\x89+\xa3B\x0e\n\xca\x87\xf7[\xfe\xfaOR\xd3\x1b\xb77\x82\xb9,\xb8\xb4t\xb4\xed3/\xde\xec>
-\xaa\xed\xcd^\x88\x87\x93\x1f\x80\x11\xc7\xe7\xadi%\xb7*\x0c\x05\x19Zfw`\xa3\x00\x8c\xaa\x01\xef\xf5\xd0\x8c\x96)
[\x94\xa7q1\xaf\xf6\x81\x0c[\x1a\xb0+\xdb&\x91%\xbf+\xb7\xc3QN\xbd\xc5)_,$n<\x87\x8fLH\xa7\xbb\xae\x9d\x88S,\xaeZm\x0c\x11\xa4h\xbe\x
hs\xf7F\x8c\x1c8\\\x91\xa2UV\xb9\xee\xccI\xee\x07\xb9\xf9h\x8d"\x89\\p\xb4\xa6\xdbi[\x18u\x88\x06#\xe1c\xe0\xe9\xb2/\x84\x0b!)\xbam:\
\xd3K\x13\xe4\x16\xe5\x8c*\x8c\xea\xb2j\xe3\x8c\xd9+\xb6\x94\xf6\x8e\x07\xb8\xc4\xd3\xbcA@\x18=\xbb\xe99\xb5\xb2m\xdc\xd1aue\x01p\xa4
{\xd9\xb5\xa8\xf9\x1b44\x80A\x08\x8d\xc1Z\x1a\x826\x0c2;\x90O\xd7^zMH\xf4\xf6\x92\xae\xd6\x95\x1dM<\x95{\xda\x95\xd0\x90#\x91X\xf7\xf
s\x8a\r\x14\x96i\x11eW\xa9\x1f\x070\xaa\x84\xfd\xf2H\x00\x7f\xa7\xe7\xa3\xe9]\xeb\xbdA\x14\xa8vm\x84[\xe9~!\xceY~9e\x03\x1c\xdb\xf8\x
\x9f\x1a4N\xb1er\x12\xe5wX\xa1\xfb=;\x92|gF\x04\x95\xcb\x95\x9a\xcb-
\xca\xa3\x99\r\x8f,\xc4\xe8\xcd\xa6\x8c\xaeH:\xb7I\xd4\r\xdf\x1b\xf4\xf7\xa5\x8d\x15\xba\x9d\x93\x8d\xdbpT\xbb(\x8dO\x98
U\x1c\xa4r\x0eK\x8f\x80\x03\x8e\\\xb2\x16\xee\x8d\xd2\x8a\x06\x82\xa6\xe2\x80\xe97\xe91\xb86\x8bEl\xbd\x13s\xb7\xa8\xc2\xd3\xcb&$\x88
c\xdb\xca#\x98\xd3\xc2\xdf6\x87V\xfaO\xd4\x1at\x8e\x8e\xfd\x145\x0e0\xf4u\x9f\xb2\x91I\xf6\x1c\xbb7\x7fu$kA\x9a\x88\xe5\x18(%\x94\x8f
<H\xecA\xf3\xc4\xfc\x88\xd2:\xb7\xb5\xa7{\r\xb4\xf7\xf0\xa6;\xaa!H\xd5\xcd%+\x11\x13\x11\xdf\xdck\x0bW#\x1f\x19.Eo(\xcb|\xb1VQ\x8a\xb
\xc6\xb6\x95G\x04/\x8f#\xdf\x1e\xff\x00\xfc\xef\xae\xb2\x15\x83|\xab\xde\x9f~\x90[\xe7i2%%\xdeG\x85p\x1a\x9e\xa9\xc9\x18\xf9\x03\xe4}
<\x81\xf2\xd31j\xa5guGF;-sl~\x91\x9bos\xc4)o\xb4)\x14\x8d\xd8\x82\x00\xe4\x0f\xd7\xee\xb7\xf6?
M:\xcd\\r|\xc8E\x85\xab\xb6\xe2\xda\xf6\xbb\xad!\xb9m\xe7Y\xe3#>\x988*~X\xf6:\xab\xda\xceB\x80J\xcd\xf7\x05\x96JY\x19\r+\x18\xcfu+\xf
<\xf6W\r<\x85\xf6\xe9\x19\xb8\xc5%bFc\xab\xa68\xab\x88\x0e\xe7\xf9\xc7\xd3\xe7\xaf1\xd4\'s#.o\xe6\x99`\xceWk\x1e\xe1\xaf\xb71\x92d\xc
GY:\xc7\xbbPI\r\x00c\xb8#\xe9\xf5)\x83\x17\xa6\xd0\xd1\xca\xa3\xd8\xf50\\.t\xd2\x00\xca\x19\x89R\xc3\x00\x80pH\'\xc8\x04\x11\xadM7\xc
8!\xd3\xb7\x10\x1c0\xc9\'8\xf6\xf7\xf1\x1a\xae\xbb\xaa=RMk\t\xb7\x1bp\xff\x00\x89\xec\x0f\x9e\nz=&\xf8\xb6\xd7\nN\xc7MW\r\xfe-
\xbfo\xa6\x91\x9a\x97\x94\xf7I\xc2abf\xe3\xe9\xa7o\x04\xa9$\x0f8BN5\xeb\xfe\x19\xea:\x89\xb5\xac\x8d\xc6\xb7\x1a\x04\x9ey\xb2O\x81@ZK
\xdc\xf5\xfb\x9e\x96\xc756\xd5\xb1\x9a\xea\xf2\x9e\x95\xa6\xdf,\xe2\x05\xac\xa9
p\x0f+ab\x8f\x969HH\x00\x03\xdf\xb6\xbe\x95\'\xc5\xfa&k\xbd\x00\xea\x8d\xb4\x1c@\xb3Wg\x8b>
(\x01d\xac\xf6\xe9\x0e\xc2{\xac\xdf\xff\x00\xa7\x0e\xc7\xbbmn\x94\xdd\xb7>\xea\xbb-
u\xd7q\xee\x9a\xdb\x95mcS\xc9\x13I!+\x1b\xe5\x1c\x021"I\xf4\xd7\xd3\xe2\xeaq\x17\x86\x0c\x10\x06\rw\xfb\x1f\x14\xb3\x8c/h\xca\xdd\xf7
[\x95=>*\x04\xb2qI\x00\x03\xb9o\x1d\xfe\x9f]|\x87\xe2\x8f\x8b\x84\xba\x11\xe8]I\x8b\x1d\x88\xf29\xbf!j\xe9\xb4\x80>\xdc8N\xf7%\x8e\x8
(gU*\xe8\xc5\x10\x80\x88Y\xbb\x02y\x85\x01\xb2\x18\x8dL\xba\xe7G\xd4\x8e\xc8\xc1c\x8dpM\x8el\xe6\xed\xdc};f\x95\x1c\xfb\x01\xc52\xdb;
\xae\x82\xa3\xec\xea\xb5\xee\xe0\xbc\x91\x86*LG\x93\xb9\x1f\x03c\xe1\x00d\x1d/\xa9\x97I\x0b\x1dd\xee.$8\x8b\xf6\x0b\xc1\xf0p\x05w\xe7
\xd2\xba-
\xb4\xe2Nn\xf1\xf4\xfd\xfe\xc9\xb8\xe4{l\x03\x82\x8c\x8bh\xdc\x92\xfd]O@!\x10\x96T\xb6\xd2L\x9e\x9dF\x02\xb1nm\x8e,\xc4\xf8r{\xf2\x00
I\xc9"R\x00\xecG\x8f\xa8?
\xdb\x1a\xf6\x9a/\xfcC\xebR\xea\x19\x06\xa0\x02l\x0b7u\xe1f\xcb\xd3\xe2km\xaa~\xfb\xb9!\x8e\xa3\x84\xcc\xc5\x87\x80\xbd\xf1\xaf\xb1\x
(\xa2?
\xd4\x8d{]4\xefx\xb9\r%\x0bE\xaf\x17\x1b\xf5\x1d\xa2\x98\xd3\xd1\xa0g\xf7o\xae\xb6c\xd44\x8c!\xed)\x0c\xf55\x17i\x83\xd6\xd5\xb1\xc9\
\xb0o.\xa3g\x04\x9c\xd8\xcf#\x91\xd9z\x00\xc6\xb8\x84\xef\xa6\x97\xbd\x87\xbcx\xeem\xbfZ\xb6\xea\xbb-
\xbaj*J\xfb\x94i.e\xe2\t\x95\x80\xe3\xe0\x10\xa3\x1f\x17\xc4{\xb1\xc6\x15\xd7\xc3\xaa\xd2;\xd1\x7f\xcb!\x04\x86\xd85|s\x9b\xef~\x06\x
i6\xc6\xe7N\xe8\x9a\x08\xc7z\xc0\xc1\xcf\xf1\xf5!\x05\xee
]\xaf\x14\xb77\x16\x99\xf7\x17L\xb7\nz\x12\xd4\xc6\x8bew\xe4\xf2F\xec\x0bpyXrnD\xbe\x0ep1\x96\xc6\xac\xed8\x91\xdbg\xf6\xb8w\xae\x0f\
\xe2\xab\xf7q\xf7\xf0\xf9\x00v$\x1f\x85\x87\xd3$\xdc\x83\xb8\x06\x8bG\x8f\xe7\xf6*\xa2P8\tu\xa7\x7f\xd9\xae\x16jK5\x81\x8d%\xcaj\xb6Z
\xeb4\xc2\x1a\xd6\xbe1l\xb0h\xfd\x0ew_a\xe4\n\xe4r\x8e$\xdc,\x1aG\xdc\xe4\xaf\xbf1\xd2\x94\x94\xf4\x91\xd7\xd2\xc4XQ\xdc\xe9@s\x84#\x
\x17l\xf6\xca\xf9\xf7\xd0\xe3\r\x87QN\'i<\xb7<\xf7\xfc\xd5\xc1\x04\x8caF\xeeK\xce\xe0\xb4\xd2\xd1\xd9\xb7\r\xb2\xe8k\xea\xe02\xb5\r%\
\xba\xc1O\xe9n\xd6\xa8T-
%\x02\x96\xce\x15\x83`g\xf0\xc6\x9e\x8c6\xee\xd5l\x95\xdcOx\xac^p\\~/\xe1\x8d\x80\x00i\x88\xd8G\x0b\x97H,\x17\xe9\xfe)\xa9\xdd\xbee\x
```

```
1 # checking name
2 filenames_rdd.take(1)
```

```
['g']
```

Name shown: ['g'], should be flower name, I know the error is within the functon definition/names asigned to the variables, didn't have enought time to find the Root Cause of this issue and fix it.

1b) Testing (3%)

i) Read from the TFRecord Dataset, using `load_dataset` and `display_9_images_from_dataset` to test.

[ ] ↳ *4 cells hidden*

1c) Set up a cluster and run the script. (6%)

Following the example from the labs, set up a cluster to run PySpark jobs in the cloud. You need to set up so that TensorFlow is installed on all nodes in the cluster.

i) Single machine cluster

Set up a cluster with a single machine using the maximal SSD size (100) and 8 vCPUs.

Enable **package installation** by passing a flag `--initialization-actions` with argument `gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh` (this is a public script that will read metadata to determine which packages to install). Then, the **packages are specified** by providing a `--metadata` flag with the argument `PIP_PACKAGES=tensorflow==2.4.0`.

Note: consider using `PIP_PACKAGES="tensorflow numpy"` or `PIP_PACKAGES=tensorflow` in case an older version of tensorflow is causing issues.

When the cluster is running, run your script to check that it works and keep the output cell output. (3%)

```
!gcloud dataproc clusters delete big-data-cw-420116-cluster --region us-central1
```

```
The cluster 'big-data-cw-420116-cluster' and all attached disks will be deleted.

Do you want to continue (Y/n)?  y

ERROR: (gcloud.dataproc.clusters.delete) NOT_FOUND: Not found: Cluster projects/big-data-cw-420116/regions/us-central1/clusters/big-data
```

```
                                    ### CODING TASK ###
#Set up a cluster with a single machine using the maximal SSD size (100) and 8 vCPUs.
#REGION = 'us-central1'
!gcloud dataproc clusters create $CLUSTER \
--bucket $PROJECT-storage \
--image-version 1.4-ubuntu18 --single-node \
--master-machine-type n1-standard-8 \
--master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
--max-idle 3600s \
--initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
--metadata PIP_PACKAGES=tensorflow==2.4.0

#--metadata "PIP_PACKAGES=tensorflow==2.4.0 scipy==1.4.1 matplotlib==3.7.0 numpy==1.25.2" # Tried this but didn't work

# to grant uniform access !gcloud storage buckets update $BUCKET --uniform-bucket-level-access
```

```
Waiting on operation [projects/big-data-cw-420116/regions/us-central1/operations/6ae51459-595f-3852-89e6-4c0d81876966].

WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security ri
Created [https://dataproc.googleapis.com/v1/projects/big-data-cw-420116/regions/us-central1/clusters/big-data-cw-420116-cluster] Cluster
```

```
#!gcloud storage buckets update $BUCKET --uniform-bucket-level-access
```

```
# Description of the cluster
!gcloud dataproc clusters describe $CLUSTER
```

```
clusterName: big-data-cw-420116-cluster
clusterUuid: 084e9d65-6812-45a2-9cab-372f310ba794
config:
  configBucket: big-data-cw-420116-storage
  endpointConfig: {}
  gceClusterConfig:
    internalIpOnly: false
    metadata:
      PIP_PACKAGES: tensorflow==2.4.0
    networkUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/global/networks/default
    serviceAccountScopes:
```

```
    - https://www.googleapis.com/auth/bigquery
    - https://www.googleapis.com/auth/bigtable.admin.table
    - https://www.googleapis.com/auth/bigtable.data
    - https://www.googleapis.com/auth/cloud.useraccounts.readonly
    - https://www.googleapis.com/auth/devstorage.full_control
    - https://www.googleapis.com/auth/devstorage.read_write
    - https://www.googleapis.com/auth/logging.write
    zoneUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/zones/us-central1-c
  initializationActions:
  - executableFile: gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
    executionTimeout: 600s
  lifecycleConfig:
    idleDeleteTtl: 3600s
    idleStartTime: '2024-05-04T09:42:30.961969Z'
  masterConfig:
    diskConfig:
      bootDiskSizeGb: 100
      bootDiskType: pd-ssd
    imageUri: https://www.googleapis.com/compute/v1/projects/cloud-dataproc/global/images/dataproc-1-4-ubu18-20220125-170200-rc01
    instanceNames:
    - big-data-cw-420116-cluster-m
    machineTypeUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/zones/us-central1-c/machineTypes/n1-standard-8
    minCpuPlatform: AUTOMATIC
    numInstances: 1
    preemptibility: NON_PREEMPTIBLE
  softwareConfig:
    imageVersion: 1.4.80-ubuntu18
    properties:
      capacity-scheduler:yarn.scheduler.capacity.root.default.ordering-policy: fair
      core:fs.gs.block.size: '134217728'
      core:fs.gs.metadata.cache.enable: 'false'
      core:hadoop.ssl.enabled.protocols: TLSv1,TLSv1.1,TLSv1.2
      dataproc:dataproc.allow.zero.workers: 'true'
      distcp:mapreduce.map.java.opts: -Xmx768m
      distcp:mapreduce.map.memory.mb: '1024'
      distcp:mapreduce.reduce.java.opts: -Xmx768m
      distcp:mapreduce.reduce.memory.mb: '1024'
      hdfs:dfs.datanode.address: 0.0.0.0:9866
      hdfs:dfs.datanode.http.address: 0.0.0.0:9864
      hdfs:dfs.datanode.https.address: 0.0.0.0:9865
      hdfs:dfs.datanode.ipc.address: 0.0.0.0:9867
      hdfs:dfs.namenode.handler.count: '20'
      hdfs:dfs.namenode.http-address: 0.0.0.0:9870
      hdfs:dfs.namenode.https-address: 0.0.0.0:9871
      hdfs:dfs.namenode.lifeline.rpc-address: big-data-cw-420116-cluster-m:8050
      hdfs:dfs.namenode.secondary.http-address: 0.0.0.0:9868
```

Run the script in the cloud and test the output.

```
1 # Submit jobs to cluster and measure execution time (Checked& NOT Working)
2 !gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_write_tfrec.py
3 %time
```

```
Job [e72d8ce747414b90b5b0ef0b944ed255] submitted.
Waiting for job output...
2024-05-04 09:45:21.057528: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynamic library 'libcudart.so
2024-05-04 09:45:21.057575: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU se
24/05/04 09:45:23 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
24/05/04 09:45:23 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
24/05/04 09:45:23 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
24/05/04 09:45:23 INFO org.spark_project.jetty.util.log: Logging initialized @5834ms to org.spark_project.jetty.util.log.Slf4jLog
24/05/04 09:45:24 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown; jvm 1.8.0_312-b07
24/05/04 09:45:24 INFO org.spark_project.jetty.server.Server: Started @5943ms
24/05/04 09:45:24 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@71dbb1cc{HTTP/1.1, (http/1.1)}{0.0.0.0:
24/05/04 09:45:24 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair Scheduler configuration file not found so jobs will be sc
24/05/04 09:45:25 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-cw-420116-cluster-m/10.128.15.22
24/05/04 09:45:25 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-data-cw-420116-cluster-m/
24/05/04 09:45:27 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1714815815041_0001
Writing TFRecords
24/05/04 09:45:34 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@71dbb1cc{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
Job [e72d8ce747414b90b5b0ef0b944ed255] finished successfully.
done: true
driverControlFilesUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/084e9d65-6812-45a2-9cab-372f310ba794/jobs/e72d8ce7
driverOutputResourceUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/084e9d65-6812-45a2-9cab-372f310ba794/jobs/e72d8c
jobUuid: 96b9484b-26bb-352a-a9f4-84042133fb34
placement:
  clusterName: big-data-cw-420116-cluster
  clusterUuid: 084e9d65-6812-45a2-9cab-372f310ba794
pysparkJob:
  mainPythonFileUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/084e9d65-6812-45a2-9cab-372f310ba794/jobs/e72d8ce747
reference:
  jobId: e72d8ce747414b90b5b0ef0b944ed255
```

```
  projectId: big-data-cw-420116
status:
  state: DONE
  stateStartTime: '2024-05-04T09:45:35.737564Z'
statusHistory:
- state: PENDING
  stateStartTime: '2024-05-04T09:45:16.346354Z'
- state: SETUP_DONE
  stateStartTime: '2024-05-04T09:45:16.397866Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2024-05-04T09:45:16.694290Z'
yarnApplications:
- name: spark_write_tfrec.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-cw-420116-cluster-m:8088/proxy/application_1714815815041_0001/
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 9.54 µs
```

Double-click (or enter) to edit

**Got the following Error: **

ERROR: (gcloud.dataproc.jobs.submit.pyspark) RESOURCE_EXHAUSTED: Multiple validation errors:

- No agent on master node(s) reported to be be active
- Unable to submit job, cluster 'big-data-cw-420116-cluster' is in state ERROR and cannot accept jobs. CPU times: user 5 µs, sys: 1 µs, total: 6 µs Wall time: 11 µs

In the free credit tier on Google Cloud, there are normally the following **restrictions** on compute machines:

- max 100GB of *SSD persistent disk*
- max 2000GB of *standard persistent disk*
- max 8 *vCPU*s
- no GPUs

See here for details The **disks are virtual** disks, where **I/O speed is limited in proportion to the size**, so we should allocate them evenly. This has mainly an effect on the **time the cluster needs to start**, as we are reading the data mainly from the bucket and we are not writing much to disk at all.

ii) Maximal cluster

Use the **largest possible cluster** within these constraints, i.e. **1 master and 7 worker nodes**. Each of them with 1 (virtual) CPU. The master should get the full *SSD* capacity and the 7 worker nodes should get equal shares of the *standard* disk capacity to maximise throughput.

Once the cluster is running, test your script. (3%)

[ ] ↳ *8 cells hidden*

1d) Optimisation, experiments, and discussion (17%)

i) Improve parallelisation

If you implemented a straightfoward version, you will **probably** observe that **all the computation** is done on only **two nodes**. This can be adressed by using the **second parameter** in the initial call to **parallelize**. Make the **suitable change** in the code you have written above and mark it up in comments as `### TASK 1d ###`.

Demonstrate the difference in cluster utilisation before and after the change based on different parameter values with **screenshots from Google Cloud** and measure the **difference in the processing time**. (6%)

ii) Experiment with cluster configurations.

In addition to the experiments above (using 8 VMs),test your program with 4 machines with double the resources each (2 vCPUs, memory, disk) and 1 machine with eightfold resources. Discuss the results in terms of disk I/O and network bandwidth allocation in the cloud. (7%)

iii) Explain the difference between this use of Spark and most standard applications like e.g. in our labs in terms of where the data is stored. What kind of parallelisation approach is used here? (4%)

Write the code below and your answers in the report.

```
                                  ### CODING TASK ### ("Done & Working")
# I) Improving parallelization; Script without second parameter
%%writefile task_1di.py
# Setting the working environment
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
#import scipy as sp scipy give error in this task.
#import scipy.stats
import time
import tensorflow as tf

#from matplotlib import pyplot as plt
from pyspark.sql import SQLContext
from pyspark.sql import Row

# define variables
PROJECT = 'big-data-cw-420116' # my project id
BUCKET = 'gs://{}-storage'.format(PROJECT) # my own bucket storage
CLUSTER = '{}-cluster'.format(PROJECT)
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

def recompress_image(image, label):
    # this reduces the amount of data, but takes some time
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)
    return image, label

# Retrieve file paths from GCS using a pattern
filenames = tf.io.gfile.glob(GCS_PATTERN)

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

### TASK 1d ###
filenames_rdd = sc.parallelize(filenames) # Adding a second parameter in the initiall call to parallelize.

# Sampling the rdd
```

```
sample_rdd = filenames_rdd.sample(False, 0.02)

# Adapting tensorflow to spark, rdd for decode jpeg and label, rdd for resize and crop image, rdd for recompress image
decode_jpeg_and_label_rdd = filenames_rdd.map(decode_jpeg_and_label)
resize_and_crop_image_rdd = decode_jpeg_and_label_rdd.map(lambda x: resize_and_crop_image(x[0],x[1]))
recompress_image_rdd = resize_and_crop_image_rdd.map(lambda x: recompress_image(x[0],x[1]))

# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints): # int64
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(tfrec_filewriter, img_bytes, label): # Create tf data records
    class_num = np.argmax(np.array(CLASSES)==label) # 'roses' => 2 (order defined in CLASSES)
    one_hot_class = np.eye(len(CLASSES))[class_num]     # [0, 0, 1, 0, 0] for class #2, roses
    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "class": _int_feature([class_num]) #,      # one class in the list
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

def write_tfrecords(index,partition):  # write the images to files.
   #tt0 = time.time()
  # good practice to have the number of records in the filename
  filename = GCS_OUTPUT + "{}.tfrec".format(index)
  # (CHANGED) You need to change GCS_OUTPUT to your own bucket to actually create new files
  with tf.io.TFRecordWriter(filename) as out_file:
    for element in partition:
      image=element[0]
      label=element[1]
      example = to_tfrecord(out_file,
                            image.numpy(), # re-compressed image: already a byte string
                            label.numpy() #
                            )
      out_file.write(example.SerializeToString())
              #print("Wrote file {} containing {} records".format(filename, partition_size))
    #print("Total time: "+str(time.time()-tt0))
  return (filename)

## Repartition RDD for parallel processing
partitions_rdd = recompress_image_rdd.repartition(PARTITIONS)
# Write TFRecords for each partition
filenames_rdd = recompress_image_rdd.mapPartitionsWithIndex(write_tfrecords)
```

```
Writing task_1di.py
```

```
#Was getting error to create the cluster so I deleted it to create it again.
!gcloud dataproc clusters delete big-data-cw-420116-cluster --region us-central1
```

```
ERROR: (gcloud.dataproc.clusters.delete) Error parsing [cluster].
The [cluster] resource is not properly specified.
Failed to find attribute [project]. The attribute can be set in the following ways:
- provide the argument `cluster` on the command line with a fully specified name
- provide the argument `--project` on the command line
- set the property `core/project`
```

```
                        ### CODING TASK ### (checked & NOT WORKING)
# Creatng Dataproc cluster with 1 master machine
#CHECK WORKERS
REGION = 'europe-west2'
import time
start_time = time.time()

!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --image-version 1.4-ubuntu18 \
    --master-machine-type n1-standard-1 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100\
    --num-workers 7 --worker-machine-type n1-standard-1 --worker-boot-disk-size 100\
    --max-idle 3600s \
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata PIP_PACKAGES=tensorflow==2.4.0
%time
end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")
```

```
ERROR: (gcloud.dataproc.clusters.create) argument --bucket: expected one argument
Usage: gcloud dataproc clusters create (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --action-on-failed-primary-workers | --no-address |
                         --async | --autoscaling-policy | --bucket |
                         --confidential-compute | --dataproc-metastore |
                         --driver-pool-accelerator |
                         --driver-pool-boot-disk-size |
                         --driver-pool-boot-disk-type | --driver-pool-id |
                         --driver-pool-local-ssd-interface |
                         --driver-pool-machine-type |
                         --driver-pool-min-cpu-platform | --driver-pool-size |
                         --enable-component-gateway | --enable-kerberos |
                         --expiration-time | --gce-pd-kms-key |
                         --gce-pd-kms-key-keyring | --gce-pd-kms-key-location |
                         --gce-pd-kms-key-project | --help |
                         --identity-config-file | --image | --image-version |
                         --initialization-action-timeout |
                         --initialization-actions | --kerberos-config-file |
                         --kerberos-kms-key | --kerberos-kms-key-keyring |
                         --kerberos-kms-key-location |
                         --kerberos-kms-key-project |
                         --kerberos-root-principal-password-uri | --kms-key |
                         --kms-keyring | --kms-location | --kms-project |
                         --labels | --master-accelerator |
                         --master-boot-disk-size | --master-boot-disk-type |
                         --master-local-ssd-interface | --master-machine-type |
                         --master-min-cpu-platform | --max-age | --max-idle |
                         --metadata | --metric-overrides |
                         --metric-overrides-file | --metric-sources |
                         --min-num-workers | --min-secondary-worker-fraction |
                         --network | --node-group |
                         --num-driver-pool-local-ssds |
                         --num-master-local-ssds | --num-masters |
                         --num-secondary-worker-local-ssds |
                         --num-secondary-workers | --num-worker-local-ssds |
                         --num-workers | --optional-components |
                         --private-ipv6-google-access-type | --properties |
                         --public-ip-address | --region | --reservation |
                         --reservation-affinity | --scopes |
                         --secondary-worker-accelerator |
                         --secondary-worker-boot-disk-size |
                         --secondary-worker-boot-disk-type |
                         --secondary-worker-local-ssd-interface |
                         --secondary-worker-machine-types |
                         --secondary-worker-type |
                         --secure-multi-tenancy-user-mapping |
                         --service-account | --shielded-integrity-monitoring |
                         --shielded-secure-boot | --shielded-vtpm |
                         --single-node | --subnet | --tags | --temp-bucket |
                         --worker-accelerator | --worker-boot-disk-size |
                         --worker-boot-disk-type |
                         --worker-local-ssd-interface | --worker-machine-type |
                         --worker-min-cpu-platform | --zone

For detailed information on this command and its flags, run:
  gcloud dataproc clusters create --help
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 6.44 µs
```

```
1 # Description of the cluster
2 !gcloud dataproc clusters describe $CLUSTER
```

```
ERROR: (gcloud.dataproc.clusters.describe) argument (CLUSTER : --region=REGION): Must be specified.
Usage: gcloud dataproc clusters describe (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --help | --region

For detailed information on this command and its flags, run:
  gcloud dataproc clusters describe --help
```

```
1 # Submit the job
2 !gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1di.py
```

```
ERROR: (gcloud.dataproc.jobs.submit.pyspark) argument PY_FILE: Must be specified.
Usage: gcloud dataproc jobs submit pyspark PY_FILE (--cluster=CLUSTER | --cluster-labels=[KEY=VALUE,...]) [optional flags] [-- JOB_ARGS
  optional flags may be  --archives | --async | --bucket | --cluster |
                         --cluster-labels | --driver-log-levels |
                         --driver-required-memory-mb |
                         --driver-required-vcores | --files | --help | --jars |
                         --labels | --max-failures-per-hour |
                         --max-failures-total | --properties |
                         --properties-file | --py-files | --region

For detailed information on this command and its flags, run:
  gcloud dataproc jobs submit pyspark --help
```

```
                                ### CODING TASK ### ("Done & Working")
# I) improving parallelization, with second parameter
%%writefile task_1di.py
# Setting the working environment
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
#import scipy as sp scipy give error in this task.
#import scipy.stats
import time
import tensorflow as tf

#from matplotlib import pyplot as plt
from pyspark.sql import SQLContext
from pyspark.sql import Row

# define variables
PROJECT = 'big-data-cw-420116' # my project id
BUCKET = 'gs://{}-storage'.format(PROJECT) # my own bucket storage
CLUSTER = '{}-cluster'.format(PROJECT)
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

def recompress_image(image, label):
    # this reduces the amount of data, but takes some time
    image = tf.cast(image, tf.uint8)
    image = tf.image.encode_jpeg(image, optimize_size=True, chroma_downsampling=False)
    return image, label

# Retrieve file paths from GCS using a pattern
filenames = tf.io.gfile.glob(GCS_PATTERN)

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

### TASK 1d ###
filenames_rdd = sc.parallelize(filenames,16) # Adding a second parameter in the initiall call to parallelize.

# Sampling the rdd
```

```
sample_rdd = filenames_rdd.sample(False, 0.02)

# Adapting tensorflow to spark, rdd for decode jpeg and label, rdd for resize and crop image, rdd for recompress image
decode_jpeg_and_label_rdd = filenames_rdd.map(decode_jpeg_and_label)
resize_and_crop_image_rdd = decode_jpeg_and_label_rdd.map(lambda x: resize_and_crop_image(x[0],x[1]))
recompress_image_rdd = resize_and_crop_image_rdd.map(lambda x: recompress_image(x[0],x[1]))

# functions for writing TFRecord entries
# Feature values are always stored as lists, a single data element will be a list of size 1
def _bytestring_feature(list_of_bytestrings):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=list_of_bytestrings))

def _int_feature(list_of_ints): # int64
    return tf.train.Feature(int64_list=tf.train.Int64List(value=list_of_ints))

def to_tfrecord(tfrec_filewriter, img_bytes, label): # Create tf data records
    class_num = np.argmax(np.array(CLASSES)==label) # 'roses' => 2 (order defined in CLASSES)
    one_hot_class = np.eye(len(CLASSES))[class_num]     # [0, 0, 1, 0, 0] for class #2, roses
    feature = {
        "image": _bytestring_feature([img_bytes]), # one image in the list
        "class": _int_feature([class_num]) #,      # one class in the list
    }
    return tf.train.Example(features=tf.train.Features(feature=feature))

def write_tfrecords(index,partition):  # write the images to files.
   #tt0 = time.time()
  # good practice to have the number of records in the filename
  filename = GCS_OUTPUT + "{}.tfrec".format(index)
  # (CHANGED) You need to change GCS_OUTPUT to your own bucket to actually create new files
  with tf.io.TFRecordWriter(filename) as out_file:
    for element in partition:
      image=element[0]
      label=element[1]
      example = to_tfrecord(out_file,
                            image.numpy(), # re-compressed image: already a byte string
                            label.numpy() #
                            )
      out_file.write(example.SerializeToString())
              #print("Wrote file {} containing {} records".format(filename, partition_size))
    #print("Total time: "+str(time.time()-tt0))
  return (filename)

## Repartition RDD for parallel processing
partitions_rdd = recompress_image_rdd.repartition(PARTITIONS)
# Write TFRecords for each partition
filenames_rdd = recompress_image_rdd.mapPartitionsWithIndex(write_tfrecords)
```

```
Overwriting task_1di.py
```

```
#Was getting error to create the cluster so I deleted it to create it again.
!gcloud dataproc clusters delete big-data-cw-420116-cluster --region us-central1
```

```
ERROR: (gcloud.dataproc.clusters.delete) Error parsing [cluster].
The [cluster] resource is not properly specified.
Failed to find attribute [project]. The attribute can be set in the following ways:
- provide the argument `cluster` on the command line with a fully specified name
- provide the argument `--project` on the command line
- set the property `core/project`
```

```
# Creatng Dataproc cluster with 1 master machine
#CHECK WORKERS
import time
start_time = time.time()

!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --image-version 1.4-ubuntu18 \
    --master-machine-type n1-standard-1 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100\
    --num-workers 7 --worker-machine-type n1-standard-1 --worker-boot-disk-size 100\
    --max-idle 3600s \
    --initialization-actions gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh \
    --metadata PIP_PACKAGES=tensorflow==2.4.0
%time
end_time = time.time()
execution_time = end_time - start_time
```

```
print(f"Execution time: {execution_time} seconds")

#--initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
```

```
ERROR: (gcloud.dataproc.clusters.create) argument --bucket: expected one argument
Usage: gcloud dataproc clusters create (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --action-on-failed-primary-workers | --no-address |
                         --async | --autoscaling-policy | --bucket |
                         --confidential-compute | --dataproc-metastore |
                         --driver-pool-accelerator |
                         --driver-pool-boot-disk-size |
                         --driver-pool-boot-disk-type | --driver-pool-id |
                         --driver-pool-local-ssd-interface |
                         --driver-pool-machine-type |
                         --driver-pool-min-cpu-platform | --driver-pool-size |
                         --enable-component-gateway | --enable-kerberos |
                         --expiration-time | --gce-pd-kms-key |
                         --gce-pd-kms-key-keyring | --gce-pd-kms-key-location |
                         --gce-pd-kms-key-project | --help |
                         --identity-config-file | --image | --image-version |
                         --initialization-action-timeout |
                         --initialization-actions | --kerberos-config-file |
                         --kerberos-kms-key | --kerberos-kms-key-keyring |
                         --kerberos-kms-key-location |
                         --kerberos-kms-key-project |
                         --kerberos-root-principal-password-uri | --kms-key |
                         --kms-keyring | --kms-location | --kms-project |
                         --labels | --master-accelerator |
                         --master-boot-disk-size | --master-boot-disk-type |
                         --master-local-ssd-interface | --master-machine-type |
                         --master-min-cpu-platform | --max-age | --max-idle |
                         --metadata | --metric-overrides |
                         --metric-overrides-file | --metric-sources |
                         --min-num-workers | --min-secondary-worker-fraction |
                         --network | --node-group |
                         --num-driver-pool-local-ssds |
                         --num-master-local-ssds | --num-masters |
                         --num-secondary-worker-local-ssds |
                         --num-secondary-workers | --num-worker-local-ssds |
                         --num-workers | --optional-components |
                         --private-ipv6-google-access-type | --properties |
                         --public-ip-address | --region | --reservation |
                         --reservation-affinity | --scopes |
                         --secondary-worker-accelerator |
                         --secondary-worker-boot-disk-size |
                         --secondary-worker-boot-disk-type |
                         --secondary-worker-local-ssd-interface |
                         --secondary-worker-machine-types |
                         --secondary-worker-type |
                         --secure-multi-tenancy-user-mapping |
                         --service-account | --shielded-integrity-monitoring |
                         --shielded-secure-boot | --shielded-vtpm |
                         --single-node | --subnet | --tags | --temp-bucket |
                         --worker-accelerator | --worker-boot-disk-size |
                         --worker-boot-disk-type |
                         --worker-local-ssd-interface | --worker-machine-type |
                         --worker-min-cpu-platform | --zone

For detailed information on this command and its flags, run:
  gcloud dataproc clusters create --help
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 7.87 µs
```

```
# Description of the cluster
!gcloud dataproc clusters describe $CLUSTER
```

```
ERROR: (gcloud.dataproc.clusters.describe) argument (CLUSTER : --region=REGION): Must be specified.
Usage: gcloud dataproc clusters describe (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --help | --region

For detailed information on this command and its flags, run:
  gcloud dataproc clusters describe --help
```

```
# Submit the job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1di.py
```

```
ERROR: (gcloud.dataproc.jobs.submit.pyspark) argument PY_FILE: Must be specified.
Usage: gcloud dataproc jobs submit pyspark PY_FILE (--cluster=CLUSTER | --cluster-labels=[KEY=VALUE,...]) [optional flags] [-- JOB_ARGS
  optional flags may be  --archives | --async | --bucket | --cluster |
                         --cluster-labels | --driver-log-levels |
                         --driver-required-memory-mb |
                         --driver-required-vcores | --files | --help | --jars |
```

```
                              --labels | --max-failures-per-hour |
                              --max-failures-total | --properties |
                              --properties-file | --py-files | --region

    For detailed information on this command and its flags, run:
      gcloud dataproc jobs submit pyspark --help
```

```
# ii) Experiment with cluster configurations
```

```
#Was getting error to create the cluster so I deleted it to create it again.
!gcloud dataproc clusters delete big-data-cw-420116-cluster --region us-central1
```

```
ERROR: (gcloud.dataproc.clusters.delete) Error parsing [cluster].
The [cluster] resource is not properly specified.
Failed to find attribute [project]. The attribute can be set in the following ways:
- provide the argument `cluster` on the command line with a fully specified name
- provide the argument `--project` on the command line
- set the property `core/project`
```

```
# II) 4 machines with double the resources each (2 vCPUs, memory, disk) I need to change to 100 and take screenshot
import time
start_time = time.time()

REGION = 'us-central1'
!gcloud dataproc clusters create $CLUSTER \
  --bucket $PROJECT-storage \
  --image-version 1.4-ubuntu18 \
  --master-machine-type n1-standard-2 \
  --master-boot-disk-type pd-ssd --master-boot-disk-size 200 \
  --num-workers 3  --worker-machine-type n1-standard-2 --worker-boot-disk-size 200 \
  --max-idle 3600s \
  --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
  --metadata PIP_PACKAGES=tensorflow==2.1.0
%time

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")


  # Extra resources checked:
  # https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/init-actions
  # https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/cluster-properties
```

```
ERROR: (gcloud.dataproc.clusters.create) argument --bucket: expected one argument
Usage: gcloud dataproc clusters create (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --action-on-failed-primary-workers | --no-address |
                         --async | --autoscaling-policy | --bucket |
                         --confidential-compute | --dataproc-metastore |
                         --driver-pool-accelerator |
                         --driver-pool-boot-disk-size |
                         --driver-pool-boot-disk-type | --driver-pool-id |
                         --driver-pool-local-ssd-interface |
                         --driver-pool-machine-type |
                         --driver-pool-min-cpu-platform | --driver-pool-size |
                         --enable-component-gateway | --enable-kerberos |
                         --expiration-time | --gce-pd-kms-key |
                         --gce-pd-kms-key-keyring | --gce-pd-kms-key-location |
                         --gce-pd-kms-key-project | --help |
                         --identity-config-file | --image | --image-version |
                         --initialization-action-timeout |
                         --initialization-actions | --kerberos-config-file |
                         --kerberos-kms-key | --kerberos-kms-key-keyring |
                         --kerberos-kms-key-location |
                         --kerberos-kms-key-project |
                         --kerberos-root-principal-password-uri | --kms-key |
                         --kms-keyring | --kms-location | --kms-project |
                         --labels | --master-accelerator |
                         --master-boot-disk-size | --master-boot-disk-type |
                         --master-local-ssd-interface | --master-machine-type |
                         --master-min-cpu-platform | --max-age | --max-idle |
                         --metadata | --metric-overrides |
                         --metric-overrides-file | --metric-sources |
                         --min-num-workers | --min-secondary-worker-fraction |
                         --network | --node-group |
                         --num-driver-pool-local-ssds |
                         --num-master-local-ssds | --num-masters |
                         --num-secondary-worker-local-ssds |
```

```
                                --num-secondary-workers | --num-worker-local-ssds |
                                --num-workers | --optional-components |
                                --private-ipv6-google-access-type | --properties |
                                --public-ip-address | --region | --reservation |
                                --reservation-affinity | --scopes |
                                --secondary-worker-accelerator |
                                --secondary-worker-boot-disk-size |
                                --secondary-worker-boot-disk-type |
                                --secondary-worker-local-ssd-interface |
                                --secondary-worker-machine-types |
                                --secondary-worker-type |
                                --secure-multi-tenancy-user-mapping |
                                --service-account | --shielded-integrity-monitoring |
                                --shielded-secure-boot | --shielded-vtpm |
                                --single-node | --subnet | --tags | --temp-bucket |
                                --worker-accelerator | --worker-boot-disk-size |
                                --worker-boot-disk-type |
                                --worker-local-ssd-interface | --worker-machine-type |
                                --worker-min-cpu-platform | --zone

    For detailed information on this command and its flags, run:
      gcloud dataproc clusters create --help
    CPU times: user 3 µs, sys: 1e+03 ns, total: 4 µs
    Wall time: 6.68 µs
```

```
1 # Description of the cluster
2 !gcloud dataproc clusters describe $CLUSTER
```

```
    ERROR: (gcloud.dataproc.clusters.describe) argument (CLUSTER : --region=REGION): Must be specified.
    Usage: gcloud dataproc clusters describe (CLUSTER : --region=REGION) [optional flags]
      optional flags may be  --help | --region

    For detailed information on this command and its flags, run:
      gcloud dataproc clusters describe --help
```

```
1 # Submit the job
2 !gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1di.py
```

```
    ERROR: (gcloud.dataproc.jobs.submit.pyspark) argument PY_FILE: Must be specified.
    Usage: gcloud dataproc jobs submit pyspark PY_FILE (--cluster=CLUSTER | --cluster-labels=[KEY=VALUE,...]) [optional flags] [-- JOB_ARGS
      optional flags may be  --archives | --async | --bucket | --cluster |
                             --cluster-labels | --driver-log-levels |
                             --driver-required-memory-mb |
                             --driver-required-vcores | --files | --help | --jars |
                             --labels | --max-failures-per-hour |
                             --max-failures-total | --properties |
                             --properties-file | --py-files | --region

    For detailed information on this command and its flags, run:
      gcloud dataproc jobs submit pyspark --help
```

```
1 #Was getting error to create the cluster so I deleted it to create it again.
2 !gcloud dataproc clusters delete big-data-cw-420116-cluster --region us-central1
```

```
    ERROR: (gcloud.dataproc.clusters.delete) Error parsing [cluster].
    The [cluster] resource is not properly specified.
    Failed to find attribute [project]. The attribute can be set in the following ways:
    - provide the argument `cluster` on the command line with a fully specified name
    - provide the argument `--project` on the command line
    - set the property `core/project`
```

```
# 1 machine with eightfold resources, need to change the disk size to 100 after running.
import time
start_time = time.time()

!gcloud dataproc clusters create $CLUSTER \
  --bucket $PROJECT-storage \
  --image-version 1.4-ubuntu18 \
  --master-machine-type n1-standard-8 \
  --master-boot-disk-type pd-ssd --master-boot-disk-size 200 \
  --num-workers 0 \
  --max-idle 3600s \
  --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
  --metadata PIP_PACKAGES=tensorflow==2.1.0
%time

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")


  # Extra resources checked:
  # https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/init-actions
  # https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/cluster-properties
```

```
ERROR: (gcloud.dataproc.clusters.create) argument --bucket: expected one argument
Usage: gcloud dataproc clusters create (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --action-on-failed-primary-workers | --no-address |
                         --async | --autoscaling-policy | --bucket |
                         --confidential-compute | --dataproc-metastore |
                         --driver-pool-accelerator |
                         --driver-pool-boot-disk-size |
                         --driver-pool-boot-disk-type | --driver-pool-id |
                         --driver-pool-local-ssd-interface |
                         --driver-pool-machine-type |
                         --driver-pool-min-cpu-platform | --driver-pool-size |
                         --enable-component-gateway | --enable-kerberos |
                         --expiration-time | --gce-pd-kms-key |
                         --gce-pd-kms-key-keyring | --gce-pd-kms-key-location |
                         --gce-pd-kms-key-project | --help |
                         --identity-config-file | --image | --image-version |
                         --initialization-action-timeout |
                         --initialization-actions | --kerberos-config-file |
                         --kerberos-kms-key | --kerberos-kms-key-keyring |
                         --kerberos-kms-key-location |
                         --kerberos-kms-key-project |
                         --kerberos-root-principal-password-uri | --kms-key |
                         --kms-keyring | --kms-location | --kms-project |
                         --labels | --master-accelerator |
                         --master-boot-disk-size | --master-boot-disk-type |
                         --master-local-ssd-interface | --master-machine-type |
                         --master-min-cpu-platform | --max-age | --max-idle |
                         --metadata | --metric-overrides |
                         --metric-overrides-file | --metric-sources |
                         --min-num-workers | --min-secondary-worker-fraction |
                         --network | --node-group |
                         --num-driver-pool-local-ssds |
                         --num-master-local-ssds | --num-masters |
                         --num-secondary-worker-local-ssds |
                         --num-secondary-workers | --num-worker-local-ssds |
                         --num-workers | --optional-components |
                         --private-ipv6-google-access-type | --properties |
                         --public-ip-address | --region | --reservation |
                         --reservation-affinity | --scopes |
                         --secondary-worker-accelerator |
                         --secondary-worker-boot-disk-size |
                         --secondary-worker-boot-disk-type |
                         --secondary-worker-local-ssd-interface |
                         --secondary-worker-machine-types |
                         --secondary-worker-type |
                         --secure-multi-tenancy-user-mapping |
                         --service-account | --shielded-integrity-monitoring |
                         --shielded-secure-boot | --shielded-vtpm |
                         --single-node | --subnet | --tags | --temp-bucket |
                         --worker-accelerator | --worker-boot-disk-size |
                         --worker-boot-disk-type |
                         --worker-local-ssd-interface | --worker-machine-type |
                         --worker-min-cpu-platform | --zone

For detailed information on this command and its flags, run:
  gcloud dataproc clusters create --help
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 6.68 µs
```

```
# Description of the cluster
!gcloud dataproc clusters describe $CLUSTER
```

```
ERROR: (gcloud.dataproc.clusters.describe) argument (CLUSTER : --region=REGION): Must be specified.
Usage: gcloud dataproc clusters describe (CLUSTER : --region=REGION) [optional flags]
  optional flags may be  --help | --region

For detailed information on this command and its flags, run:
  gcloud dataproc clusters describe --help
```

```
# Submit the job
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \task_1di.py
```

```
ERROR: (gcloud.dataproc.jobs.submit.pyspark) argument PY_FILE: Must be specified.
Usage: gcloud dataproc jobs submit pyspark PY_FILE (--cluster=CLUSTER | --cluster-labels=[KEY=VALUE,...]) [optional flags] [-- JOB_ARGS
  optional flags may be  --archives | --async | --bucket | --cluster |
                         --cluster-labels | --driver-log-levels |
                         --driver-required-memory-mb |
                         --driver-required-vcores | --files | --help | --jars |
                         --labels | --max-failures-per-hour |
                         --max-failures-total | --properties |
                         --properties-file | --py-files | --region

For detailed information on this command and its flags, run:
  gcloud dataproc jobs submit pyspark --help
```

Section 2: Speed tests

We have seen that **reading from the pre-processed TFRecord files** is **faster** than reading individual image files and decoding on the fly. This task is about **measuring this effect** and **parallelizing the tests with PySpark**.

2.1 Speed test implementation

Here is **code for time measurement** to determine the **throughput in images per second**. It doesn't render the images but extracts and prints some basic information in order to make sure the image data are read. We write the information to the null device for longer measurements `null_file=open("/dev/null", mode='w')`. That way it will not clutter our cell output.

We use batches (`dset2 = dset1.batch(batch_size)`) and select a number of batches with (`dset3 = dset2.take(batch_number)`). Then we use the `time.time()` to take the **time measurement** and take it multiple times, reading from the same dataset to see if reading speed changes with mutiple readings.

We then **vary** the size of the batch (`batch_size`) and the number of batches (`batch_number`) and **store the results for different values**. Store also the **results for each repetition** over the same dataset (repeat 2 or 3 times).

The speed test should be combined in a **function** `time_configs()` that takes a configuration, i.e. a dataset and arrays of `batch_sizes`, `batch_numbers`, and `repetitions` (an array of integers starting from 1), as **arguments** and runs the time measurement for each combination of batch_size and batch_number for the requested number of repetitions.

```
### Checked ###
# Here are some useful values for testing your code, use higher values later for actually testing throughput
batch_sizes = [2,4]
batch_numbers = [3,6]
repetitions = [1]

def time_configs(dataset, batch_sizes, batch_numbers, repetitions):
    dims = [len(batch_sizes),len(batch_numbers),len(repetitions)]
    print(dims)
    results = np.zeros(dims)
    params = np.zeros(dims + [3])
    print( results.shape )
    with open("/dev/null",mode='w') as null_file: # for printing the output without showing it
        tt = time.time() # for overall time taking
        for bsi,bs in enumerate(batch_sizes):
            for dsi, ds in enumerate(batch_numbers):
                batched_dataset = dataset.batch(bs)
                timing_set = batched_dataset.take(ds)
                for ri,rep in enumerate(repetitions):
                    print("bs: {}, ds: {}, rep: {}".format(bs,ds,rep))
                    t0 = time.time()
                    for image, label in timing_set:
                        #print("Image batch shape {}".format(image.numpy().shape),
                        print("Image batch shape {}, {})".format(image.numpy().shape,
                            [str(lbl) for lbl in label.numpy()]), null_file)
                    td = time.time() - t0 # duration for reading images
                    results[bsi,dsi,ri] = ( bs * ds) / td
                    params[bsi,dsi,ri] = [ bs, ds, rep ]
    print("total time: "+str(time.time()-tt))
    return results, params
```

Let's try this function with a **small number** of configurations of batch_sizes batch_numbers and repetions, so that we get a set of parameter combinations and corresponding reading speeds. Try reading from the image fi**l**es (dataset4) and the TFRecord fi**l**es (datasetTfrec).

```
### Checked ###

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

dataset4 = dsetResized.map(recompress_image)

#Time configurations for dataset4
[res,par] = time_configs(dataset4, batch_sizes, batch_numbers, repetitions)
print(res)
print(par)

print("=============")

# Time configurations for datasetTfrec
[res,par] = time_configs(datasetTfrec, batch_sizes, batch_numbers, repetitions)
print(res)
print(par)
```

```
[2, 2, 1]
(2, 2, 1)
bs: 2, ds: 3, rep: 1
Image batch shape (2,), ["b'roses'", "b'roses'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'dandelion'", "b'daisy'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'roses'", "b'tulips'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
bs: 2, ds: 6, rep: 1
Image batch shape (2,), ["b'daisy'", "b'roses'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'daisy'", "b'daisy'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'roses'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'daisy'", "b'tulips'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'dandelion'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'tulips'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
bs: 4, ds: 3, rep: 1
Image batch shape (4,), ["b'sunflowers'", "b'daisy'", "b'roses'", "b'tulips'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding
Image batch shape (4,), ["b'dandelion'", "b'sunflowers'", "b'sunflowers'", "b'sunflowers'"] <_io.TextIOWrapper name='/dev/null' mode
Image batch shape (4,), ["b'roses'", "b'tulips'", "b'dandelion'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encod
bs: 4, ds: 6, rep: 1
Image batch shape (4,), ["b'dandelion'", "b'sunflowers'", "b'daisy'", "b'tulips'"] <_io.TextIOWrapper name='/dev/null' mode='w' enco
Image batch shape (4,), ["b'tulips'", "b'dandelion'", "b'daisy'", "b'daisy'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding=
Image batch shape (4,), ["b'tulips'", "b'tulips'", "b'tulips'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encodin
Image batch shape (4,), ["b'daisy'", "b'tulips'", "b'daisy'", "b'roses'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF
Image batch shape (4,), ["b'daisy'", "b'daisy'", "b'tulips'", "b'roses'"] <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF
Image batch shape (4,), ["b'roses'", "b'roses'", "b'dandelion'", "b'dandelion'"] <_io.TextIOWrapper name='/dev/null' mode='w' encodi
total time: 4.946155309677124
[[[ 6.70955344]
  [11.65212834]]

 [[10.95695471]
  [12.6035298 ]]]
[[[[2. 3. 1.]]

  [[2. 6. 1.]]]


 [[[4. 3. 1.]]
```

```
  [[4. 6. 1.]]]]
=============
[2, 2, 1]
(2, 2, 1)
bs: 2, ds: 3, rep: 1
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
bs: 2, ds: 6, rep: 1
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
bs: 4, ds: 3, rep: 1
Image batch shape (4, 192, 192, 3), ['0', '0', '0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4, 192, 192, 3), ['0', '0', '0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4, 192, 192, 3), ['0', '0', '0', '0']) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
```

Task 2: Parallelising the speed test with Spark in the cloud. (36%)

As an exercise in **Spark programming and optimisation** as well as **performance analysis**, we will now implement the **speed test** with multiple parameters in parallel with Spark. Runing multiple tests in parallel would **not be a useful approach on a single machine, but it can be in the cloud** (you will be asked to reason about this later).

2a) Create the script (14%)

Your task is now to **port the speed test above to Spark** for running it in the cloud in Dataproc. **Adapt the speed testing** as a Spark program that performs the same actions as above, but **with Spark RDDs in a distributed way**. The distribution should be such that **each parameter combination (except repetition)** is processed in a separate Spark task.

More specifically:

- i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations in an RDD (2%)
- ii) get a Spark context and create the dataset and run timing test for each combination in parallel (2%)
- iii) transform the resulting RDD to the structure (parameter_combination, images_per_second) and save these values in an array (2%)
- iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the result for each parameter (2%)
- v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associativity in mind when implementing the average. (3%)
- vi) write the results to a pickle file in your bucket (2%)
- vii) Write your code it into a file using the *cell magic* `%%writefile spark_job.py` (1%)

Important: The task here is not to parallelize the pre-processing, but to run multiple speed tests in parallel using Spark.

```
                              ### CODING TASK  #     Checked
# Setting working environment
import pyspark

from pyspark.sql import Row
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext

# Pre-defined function. Function to parse the TFRecord
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

# Pre-defined function. Function to load the dataset
def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# Creating function for checking the performance
def time_configs_new(parameters_rdd):
  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]
  filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
  dset = load_dataset(filenames)

  batch = dset.batch(batch_size)
  sample_set = dset.take(batch_num)

  tt = []
  for rep in range(repetition):
    start = time.time()
    for picture in sample_set:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

  # Extra resources checked:
  # https://www.tensorflow.org/io/api_docs/python/tfio/IODataset
```

```
#  i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations in an RDD (2%)
# ii) get a Spark context and create the dataset and run timing test for each combination in parallel (2%)

batch_sizes = [6, 8, 10, 12]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []
for batch_size in batch_sizes:
  for batch in batch_numbers:
    for r in repetitions:
      parameter_list.append([batch_size,batch,r])

columns = ["batch_sizes", "batch_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
tfr_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
tfr_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
tfr_files = tfr_files_rdd.flatMap(time_configs_new)

# Construct dataframe (df) ffor TFR files
tfr_files_df = tfr_files.toDF(columns)

# Load and decode dataset
def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN)
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

# Function for image_files
def img_configs_new(parameters_rdd):
  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]

  dset = load_dataset_decoded()
  batch = dset.batch(batch_size)
  sample = dset.take(batch_num)
  tt = []
  for rep in range(repetition):
    start = time.time()
    for picture in sample:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
image_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
image_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
image_files = image_files_rdd.flatMap(img_configs_new)

# Create df for image files
image_files_df = image_files.toDF(columns)

```

```
# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these values in an array (2%)

tfr_array_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']),
                                                int(x['batch_nums']),
                                                int(x['repetitions']),
                                                int(x['datasetsize']),
                                                float(x['reading_speed']),
                                                float(x['throughput'])))

print("TFRecords Array RDD")
tfr_array_rdd.take(6)

# Extra resources checked:
# https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/
# https://sparkbyexamples.com/
# https://www.datacamp.com/tracks/big-data-with-pyspark
# https://pages.databricks.com/rs/094-YMS-629/images/LearningSpark2.0.pdf (This book in particular has been really helpful)
# https://sparkbyexamples.com/pyspark/pyspark-convert-dataframe-to-rdd/
```

```
TFRecords Array RDD
[(6, 6, 1, 36, 0.2102503776550293, 171.22442490480285),
 (6, 6, 2, 36, 0.17751073837280273, 202.80463215917607),
 (6, 6, 2, 36, 0.18248200416564941, 197.279727196117),
 (6, 6, 3, 36, 0.1756117343902588, 204.9976906440537),
 (6, 6, 3, 36, 0.17471814155578613, 206.04614769500327),
 (6, 6, 3, 36, 0.20185422897338867, 178.34652354371053)]
```

```
# RDD for image file parameter_combination

image_array_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_sizes']),
                                                    int(x['batch_nums']),
                                                    int(x['repetitions']),
                                                    int(x['datasetsize']),
                                                    float(x['reading_speed']),
                                                    float(x['throughput'])))

print("Image Array RDD")
image_array_rdd.take(6)
```

```
Image Array RDD
[(6, 6, 1, 36, 0.42624759674072266, 84.45795419205152),
 (6, 6, 2, 36, 0.4570634365081787, 78.76368382259739),
 (6, 6, 2, 36, 0.5044350624084473, 71.36696610286451),
 (6, 6, 3, 36, 0.5790822505950928, 62.167334541862864),
 (6, 6, 3, 36, 0.525456428527832, 68.5118652004334),
 (6, 6, 3, 36, 0.5065336227416992, 71.07129395506638)]
```

```
# iv) Create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the result for each parameter

# Create RDD for image files batch sieze  and their corresponding speeds
image_batchSizes_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
image_batchSizes_speed = image_batchSizes_speed_rdd.collect()  # collect batch size results

# Create RDD for image files batch numbers and their corresponding speeds
image_batchNums_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
image_batchNums_speed = image_batchNums_speed_rdd.collect()   # collect batch number results

# Create RDD for image files repetitions and their corresponding speeds
image_repetitions_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
image_repetitions_speed = image_repetitions_speed_rdd.collect()  # collect repetitions results

# Create RDD for image files data set size and their corresponding speeds
image_dsSize_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
image_dsSize_speed = image_dsSize_speed_rdd.collect() # collect datasetsize results




# Create RDD for batch sizes and their corresponding speeds
tfr_batchSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
tfr_batchSize_speed = tfr_batchSize_speed_rdd.collect() # collect batch size results

# Create RDD for batch numbers and their corresponding speeds
tfr_batchNum_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
tfr_batchNum_speed = tfr_batchNum_speed_rdd.collect()  # collect batch number results

# Create RDD for repetitions and their corresponding speeds
tfr_repetitions_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
tfr_repetitions_speed = tfr_repetitions_speed_rdd.collect()  # collect repetitions results

# Create RDD for dataset size (dsSize) and their corresponding speeds
tfr_dsSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
tfr_dsSize_speed = tfr_dsSize_speed_rdd.collect() # collect datasetsize results


# Extra resources checked:
# https://spark.apache.org/docs/latest/rdd-programming-guide.html
# https://sparkbyexamples.com/pyspark-rdd/
# https://www.oreilly.com/library/view/learning-spark/9781449359034/ch04.html (best resource)
# https://stackoverflow.com/questions/73014570/best-way-to-join-2-pair-rdds-with-exactly-the-same-keys-and-where-all-keys-are-u
# https://towardsdatascience.com/the-ultimate-guide-to-functional-programming-for-big-data-1e57b0d225a3
# https://spark.apache.org/docs/latest/rdd-programming-guide.html#basics
# https://www.geeksforgeeks.org/pyspark-map-transformation/
# https://www.analyticsvidhya.com/blog/2016/10/using-pyspark-to-perform-transformations-and-actions-on-rdd/
```

```
# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associativity in mind when impl

# Create RDD for batch sizes and their corresponding avg speed
tfr_batchSize_avgSpeed_rdd = tfr_batchSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValue
tfr_batchSize_avgSpeed = tfr_batchSize_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for tfr batch numbers and their corresponding avg speed
tfr_batchNums_avgSpeed_rdd = tfr_batchNum_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues
tfr_batchNums_avgSpeed = tfr_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for tfr repetitions and their corresponding avg speed
tfr_repetitions_avgSpeed_rdd = tfr_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
tfr_repetitions_avgSpeed = tfr_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for tfr dataset size and their corresponding avg speed
tfr_dsSize_avgSpeed_rdd = tfr_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(lamb
tfr_dsSize_avgSpeed = tfr_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results



# Create RDD for image files batch size and their corresponding avg speed
image_batchSizes_avgSpeed_rdd = image_batchSizes_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).ma
image_batchSizes_avgSpeed = image_batchSizes_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for image batch number and their corresponding avg speed
image_batchNums_avgSpeed_rdd = image_batchNums_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
image_batchNums_avgSpeed = image_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for image repetitions and their corresponding avg speed
image_repetitions_avgSpeed_rdd = image_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).
image_repetitions_avgSpeed = image_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for image dataset size and their corresponding avg speed
image_dsSize_avgSpeed_rdd = image_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(
image_dsSize_avgSpeed = image_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results


# Some extra resources checked:
# https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.mapValues.html
#
```

```
# vi) write the results to a pickle file in your bucket (2%)

# Save object to local file using pickle, upload to GCS, and log status
def save(object,bucket,filename):
    with open(filename, mode='wb') as f:
      pickle.dump(object,f)
    print("Saving{} to {}".format(filename,bucket))
    import subprocess
    proc=subprocess.run(["gsutil","cp",filename,bucket],stderr=subprocess.PIPE)
    print("gstuil returned: " + str(proc.returncode))
    print(str(proc.stderr))

filename="2aVI_results.pkl"
with open(filename,mode='wb') as f:
      pickle.dump(tfr_batchSize_speed,f)
      pickle.dump(tfr_batchNum_speed,f)
      pickle.dump(tfr_repetitions_speed,f)
      pickle.dump(tfr_dsSize_speed,f)
      pickle.dump(image_batchSizes_speed,f)
      pickle.dump(image_batchNums_speed,f)
      pickle.dump(image_repetitions_speed,f)
      pickle.dump(image_dsSize_speed,f)
      pickle.dump(tfr_batchSize_avgSpeed,f)
      pickle.dump(tfr_batchNums_avgSpeed,f)
      pickle.dump(tfr_repetitions_avgSpeed,f)
      pickle.dump(tfr_dsSize_avgSpeed,f)
      pickle.dump(image_batchSizes_avgSpeed,f)
      pickle.dump(image_batchNums_avgSpeed,f)
      pickle.dump(image_repetitions_avgSpeed,f)
      pickle.dump(image_dsSize_avgSpeed,f)


print("Saving {} to {}".format(filename,BUCKET))
import subprocess
proc = subprocess.run(["gsutil", "cp",filename, BUCKET], stderr=subprocess.PIPE)
print("gstuil returned: " +str(proc.returncode))
print(str(proc.stderr))


# Extra resources consulted:
# https://www.geeksforgeeks.org/file-handling-python/
# https://docs.python.org/3/library/subprocess.html
# https://stackoverflow.com/questions/40982132/python-pickle-dump-wb-parameter
```

```
Saving 2aVI_results.pkl to gs://big-data-cw-420116-storage
gstuil returned: 0
b'Copying file://2aVI_results.pkl [Content-Type=application/octet-stream]...\n/ [0 files][    0.0 B/ 10.6 KiB]
```

```
# vii) Write your code it into a file using the cell magic %%writefile spark_job.py (1%)

%%writefile spark_job.py
# Setting the working environment
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
#import scipy as sp
#import scipy.stats
import time
import tensorflow as tf
print ("TensorFlow Version" == tf.__version__)

#from matplotlib import pyplot as plt
from pyspark.sql import Row
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext

PROJECT = 'big-data-cw-420116'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
          # labels for the data
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket
nb_images = len(tf.io.gfile.glob(GCS_PATTERN)) # number of images


# Pre-defined function. Function to parse the TFRecord
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

```

```
# Pre-defined function. Function to load the dataset
def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# Creating function for checking the performance
def time_configs_new(parameters_rdd):
  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]

  filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
  dset = load_dataset(filenames)

  batch = dset.batch(batch_size)
  sample_set = dset.take(batch_num)

  tt = []
  for rep in range(repetition):
    start = time.time()
    for picture in sample_set:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

#  i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations in an RDD (2%)
# ii) get a Spark context and create the dataset and run timing test for each combination in parallel (2%)

batch_sizes = [6, 8, 10, 12]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []
for batch_size in batch_sizes:
  for batch in batch_numbers:
    for r in repetitions:
      parameter_list.append([batch_size,batch,r])

columns = ["batch_sizes", "batch_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
tfr_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
tfr_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
tfr_files = tfr_files_rdd.flatMap(time_configs_new)

# Construct dataframe (df) for TFR files
tfr_files_df = tfr_files.toDF(columns)

# Load and decode dataset
def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN)
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

# create function for image files
def img_configs_new(parameters_rdd):
  dset = load_dataset_decoded()
```

```
  tt = []

  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]

  batch = dset.batch(batch_size)
  sample = dset.take(batch_num)

  for rep in range(repetition):
    start = time.time()
    for picture in sample:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
image_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
image_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
image_files = image_files_rdd.flatMap(img_configs_new)

# Create df for image files
image_files_df = image_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these values in an array (2%)
tfr_array_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']),
                                                int(x['batch_nums']),
                                                int(x['repetitions']),
                                                int(x['datasetsize']),
                                                float(x['reading_speed']),
                                                float(x['throughput'])))
tfr_array_rdd.take(6)



# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the result for each parameter

# Create RDD for batch sizes and their corresponding speeds
tfr_batchSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
tfr_batchSize_speed = tfr_batchSize_speed_rdd.collect() # collect batch size results

# Create RDD for batch numbers and their corresponding speeds
tfr_batchNum_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
tfr_batchNum_speed = tfr_batchNum_speed_rdd.collect()  # collect batch number results

# Create RDD for repetitions and their corresponding speeds
tfr_repetitions_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
tfr_repetitions_speed = tfr_repetitions_speed_rdd.collect()  # collect repetitions results

# Create RDD for dataset size (dsSize) and their corresponding speeds
tfr_dsSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
tfr_dsSize_speed = tfr_dsSize_speed_rdd.collect() # collect datasetsize results

# Create RDD for image files batch sieze  and their corresponding speeds
image_batchSizes_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
image_batchSizes_speed = image_batchSizes_speed_rdd.collect()  # collect batch size results

# Create RDD for image files batch numbers and their corresponding speeds
image_batchNums_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
image_batchNums_speed = image_batchNums_speed_rdd.collect()   # collect batch number results

# Create RDD for image files repetitions and their corresponding speeds
image_repetitions_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
image_repetitions_speed = image_repetitions_speed_rdd.collect()  # collect repetitions results

```

```
# Create RDD for image files data set size and their corresponding speeds
image_dsSize_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
image_dsSize_speed = image_dsSize_speed_rdd.collect() # collect datasetsize results



# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associativity in mind when impl

# Create RDD for batch sizes and their corresponding avg speed
tfr_batchSize_avgSpeed_rdd = tfr_batchSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValue
tfr_batchSize_avgSpeed = tfr_batchSize_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for tfr batch numbers and their corresponding avg speed
tfr_batchNums_avgSpeed_rdd = tfr_batchNum_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues
tfr_batchNums_avgSpeed = tfr_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for tfr repetitions and their corresponding avg speed
tfr_repetitions_avgSpeed_rdd = tfr_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
tfr_repetitions_avgSpeed = tfr_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for tfr dataset size and their corresponding avg speed
tfr_dsSize_avgSpeed_rdd = tfr_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(lamb
tfr_dsSize_avgSpeed = tfr_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results

# Create RDD for image files batch size and their corresponding avg speed
image_batchSizes_avgSpeed_rdd = image_batchSizes_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).ma
image_batchSizes_avgSpeed = image_batchSizes_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for image batch number and their corresponding avg speed
image_batchNums_avgSpeed_rdd = image_batchNums_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
image_batchNums_avgSpeed = image_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for image repetitions and their corresponding avg speed
image_repetitions_avgSpeed_rdd = image_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).
image_repetitions_avgSpeed = image_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for image dataset size and their corresponding avg speed
image_dsSize_avgSpeed_rdd = image_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(
image_dsSize_avgSpeed = image_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results



# vi) write the results to a pickle file in your bucket (2%)

#Save object to local file using pickle, upload to GCS, and log status
def save(object,bucket,filename):
    with open(filename, mode='wb') as f:
      pickle.dump(object,f)
    print("Saving{} to {}".format(filename,bucket))
    import subprocess
    proc=subprocess.run(["gsutil","cp",filename,bucket],stderr=subprocess.PIPE)
    print("gstuil returned: " + str(proc.returncode))
    print(str(proc.stderr))

filename="2aVI_results.pkl"
with open(filename,mode='wb') as f:
      pickle.dump(tfr_batchSize_speed,f)
      pickle.dump(tfr_batchNum_speed,f)
      pickle.dump(tfr_repetitions_speed,f)
      pickle.dump(tfr_dsSize_speed,f)
      pickle.dump(image_batchSizes_speed,f)
      pickle.dump(image_batchNums_speed,f)
      pickle.dump(image_repetitions_speed,f)
      pickle.dump(image_dsSize_speed,f)
      pickle.dump(tfr_batchSize_avgSpeed,f)
      pickle.dump(tfr_batchNums_avgSpeed,f)
      pickle.dump(tfr_repetitions_avgSpeed,f)
      pickle.dump(tfr_dsSize_avgSpeed,f)
      pickle.dump(image_batchSizes_avgSpeed,f)
      pickle.dump(image_batchNums_avgSpeed,f)
      pickle.dump(image_repetitions_avgSpeed,f)
      pickle.dump(image_dsSize_avgSpeed,f)

print("Saving {} to {}".format(filename,BUCKET))
import subprocess
```

```
proc = subprocess.run(["gsutil", "cp",filename, BUCKET], stderr=subprocess.PIPE)
print("gstuil returned: " +str(proc.returncode))
print(str(proc.stderr))
```

```
Writing spark_job.py
```

2b) Testing the code and collecting results (4%)

i) First, test locally with `%run` .

It is useful to create a **new filename argument**, so that old results don't get overwritten.

You can for instance use `datetime.datetime.now().strftime("%y%m%d-%H%M")` to get a string with the current date and time and use that in the file name.

```
### CODING TASK
%run ./spark_job.py
```

```
False
Saving 2aVI_results.pkl to gs://big-data-cw-420116-storage
gstuil returned: 0
b'Copying file://2aVI_results.pkl [Content-Type=application/octet-stream]...\n/ [0 files][    0.0 B/ 10.6 KiB]
<Figure size 640x480 with 0 Axes>
```

```
%%writefile spark_job2b.py
# Setting the working environment
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
import time
import tensorflow as tf
print ("TensorFlow Version" == tf.__version__)

from pyspark.sql import Row
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext

PROJECT = 'big-data-cw-420116'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
          # labels for the data
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket
nb_images = len(tf.io.gfile.glob(GCS_PATTERN)) # number of images

# Pre-defined function. Function to parse the TFRecord
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Pre-defined function. Function to load the dataset
def load_dataset(filenames):
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
```

```
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# Creating function for checking the performance
def time_configs_new(parameters_rdd):
  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]

  filenames = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
  dset = load_dataset(filenames)

  batch = dset.batch(batch_size)
  sample_set = dset.take(batch_num)

  tt = []
  for rep in range(repetition):
    start = time.time()
    for picture in sample_set:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

#  i) combine the previous cells to have the code to create a dataset and create a list of parameter combinations in an RDD (2%)
# ii) get a Spark context and create the dataset and run timing test for each combination in parallel (2%)

batch_sizes = [6, 8, 10, 12]
batch_numbers = [6, 9, 12, 15]
repetitions = [1, 2, 3]

parameter_list = []
for batch_size in batch_sizes:
  for batch in batch_numbers:
    for r in repetitions:
      parameter_list.append([batch_size,batch,r])

columns = ["batch_sizes", "batch_nums", "repetitions", "datasetsize", "reading_speed", "throughput"]

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
tfr_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
tfr_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
tfr_files = tfr_files_rdd.flatMap(time_configs_new)

# Construct dataframe (df) for TFR files
tfr_files_df = tfr_files.toDF(columns)

# Load and decode dataset
def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN)
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

# create function for image files
def img_configs_new(parameters_rdd):
  dset = load_dataset_decoded()
  tt = []

  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]

```

```
  batch = dset.batch(batch_size)
  sample = dset.take(batch_num)

  for rep in range(repetition):
    start = time.time()
    for picture in sample:
      print('string', file=open("/dev/null", mode='w'))
    end = time.time()
    reading_speed = end - start
    throughput = float((batch_size * batch_num) / (end - start))
    datasetsize = batch_size * batch_num
    tt.append([batch_size, batch_num, repetition, datasetsize, reading_speed, throughput])
  return tt

# Get spark context for RDDs
sc = pyspark.SparkContext.getOrCreate()

# Parallelize data into RDD
image_files_rdd = sc.parallelize(parameter_list)

# Initialize spark session for df conversion
image_files_sparkSession = SparkSession(sc)

# Flattening RDD into list of list
image_files = image_files_rdd.flatMap(img_configs_new)

# Create df for image files
image_files_df = image_files.toDF(columns)

# iii) transform the resulting RDD to the structure ( parameter_combination, images_per_second ) and save these values in an array (2%)
tfr_array_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']),int(x['batch_nums']),int(x['repetitions']),
                                                int(x['datasetsize']),float(x['reading_speed']), float(x['throughput'])))
tfr_array_rdd.take(6)


# iv) create an RDD with all results for each parameter as (parameter_value,images_per_second) and collect the result for each parameter
# Create RDD for batch sizes and their corresponding speeds
tfr_batchSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
tfr_batchSize_speed = tfr_batchSize_speed_rdd.collect() # collect batch size results

# Create RDD for batch numbers and their corresponding speeds
tfr_batchNum_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
tfr_batchNum_speed = tfr_batchNum_speed_rdd.collect()  # collect batch number results

# Create RDD for repetitions and their corresponding speeds
tfr_repetitions_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
tfr_repetitions_speed = tfr_repetitions_speed_rdd.collect()  # collect repetitions results

# Create RDD for dataset size (dsSize) and their corresponding speeds
tfr_dsSize_speed_rdd = tfr_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
tfr_dsSize_speed = tfr_dsSize_speed_rdd.collect() # collect datasetsize results

# Create RDD for image files batch sieze  and their corresponding speeds
image_batchSizes_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_sizes']), float(x['throughput'])))
image_batchSizes_speed = image_batchSizes_speed_rdd.collect()  # collect batch size results

# Create RDD for image files batch numbers and their corresponding speeds
image_batchNums_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['batch_nums']), float(x['throughput'])))
image_batchNums_speed = image_batchNums_speed_rdd.collect()   # collect batch number results

# Create RDD for image files repetitions and their corresponding speeds
image_repetitions_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['repetitions']), float(x['throughput'])))
image_repetitions_speed = image_repetitions_speed_rdd.collect()  # collect repetitions results

# Create RDD for image files data set size and their corresponding speeds
image_dsSize_speed_rdd = image_files_df.rdd.map(lambda x: (int(x['datasetsize']), float(x['throughput'])))
image_dsSize_speed = image_dsSize_speed_rdd.collect() # collect datasetsize results


# v) create an RDD with the average reading speeds for each parameter value and collect the results. Keep associativity in mind when impl

# Create RDD for batch sizes and their corresponding avg speed
tfr_batchSize_avgSpeed_rdd = tfr_batchSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValue
tfr_batchSize_avgSpeed = tfr_batchSize_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for tfr batch numbers and their corresponding avg speed
tfr_batchNums_avgSpeed_rdd = tfr_batchNum_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues
```

```
tfr_batchNums_avgSpeed = tfr_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for tfr repetitions and their corresponding avg speed
tfr_repetitions_avgSpeed_rdd = tfr_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
tfr_repetitions_avgSpeed = tfr_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for tfr dataset size and their corresponding avg speed
tfr_dsSize_avgSpeed_rdd = tfr_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(lamb
tfr_dsSize_avgSpeed = tfr_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results

# Create RDD for image files batch size and their corresponding avg speed
image_batchSizes_avgSpeed_rdd = image_batchSizes_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).ma
image_batchSizes_avgSpeed = image_batchSizes_avgSpeed_rdd.collect()  # collect batch size results

# Create RDD for image batch number and their corresponding avg speed
image_batchNums_avgSpeed_rdd = image_batchNums_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapV
image_batchNums_avgSpeed = image_batchNums_avgSpeed_rdd.collect()  # collect batch number results

# Create RDD for image repetitions and their corresponding avg speed
image_repetitions_avgSpeed_rdd = image_repetitions_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).
image_repetitions_avgSpeed = image_repetitions_avgSpeed_rdd.collect() # collect repetitions results

# Create RDD for image dataset size and their corresponding avg speed
image_dsSize_avgSpeed_rdd = image_dsSize_speed_rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(
image_dsSize_avgSpeed = image_dsSize_avgSpeed_rdd.collect()  # collect datasetsize results


# vi) write the results to a pickle file in your bucket (2%)

#Save object to local file using pickle, upload to GCS, and log status
def save(object,bucket,filename):
    with open(filename, mode='wb') as f:
      pickle.dump(object,f)
    print("Saving{} to {}".format(filename,bucket))
    import subprocess
    proc=subprocess.run(["gsutil","cp",filename,bucket],stderr=subprocess.PIPE)
    print("gstuil returned: " + str(proc.returncode))
    print(str(proc.stderr))

filename="2b_results.pkl"
with open(filename,mode='wb') as f:
      pickle.dump(tfr_batchSize_speed,f)
      pickle.dump(tfr_batchNum_speed,f)
      pickle.dump(tfr_repetitions_speed,f)
      pickle.dump(tfr_dsSize_speed,f)
      pickle.dump(image_batchSizes_speed,f)
      pickle.dump(image_batchNums_speed,f)
      pickle.dump(image_repetitions_speed,f)
      pickle.dump(image_dsSize_speed,f)
      pickle.dump(tfr_batchSize_avgSpeed,f)
      pickle.dump(tfr_batchNums_avgSpeed,f)
      pickle.dump(tfr_repetitions_avgSpeed,f)
      pickle.dump(tfr_dsSize_avgSpeed,f)
      pickle.dump(image_batchSizes_avgSpeed,f)
      pickle.dump(image_batchNums_avgSpeed,f)
      pickle.dump(image_repetitions_avgSpeed,f)
      pickle.dump(image_dsSize_avgSpeed,f)

print("Saving {} to {}".format(filename,BUCKET))
import subprocess
proc = subprocess.run(["gsutil", "cp",filename, BUCKET], stderr=subprocess.PIPE)
print("gstuil returned: " +str(proc.returncode))
print(str(proc.stderr))
```

```
Writing spark_job2b.py
```

ii) Cloud

If you have a cluster running, you can run the speed test job in the cloud.

While you run this job, switch to the Dataproc web page and take **screenshots of the CPU and network load** over time. They are displayed with some delay, so you may need to wait a little. These images will be useful in the next task. Again, don't use the SCREENSHOT function that Google provides, but just take a picture of the graphs you see for the VMs.

```
#Was getting error to create the cluster so I deleted it to create it again.
!gcloud dataproc clusters delete big-data-cw-420116-cluster
```

```
The cluster 'big-data-cw-420116-cluster' and all attached disks will be deleted.

Do you want to continue (Y/n)?  y

Waiting on operation [projects/big-data-cw-420116/regions/us-central1/operations/0338bdcb-b7ec-3f6b-90a6-270c2f41b32b].
Deleted [https://dataproc.googleapis.com/v1/projects/big-data-cw-420116/regions/us-central1/clusters/big-data-cw-420116-cluster].
```

```
### CODING TASK ###
# Set up a cluster with a single machine using the maximal SSD size (100) and  1 machine with eightfold resources.

import time
start_time = time.time()
#REGION = europe-west3
!gcloud dataproc clusters create $CLUSTER \
    --bucket $PROJECT-storage \
    --image-version 1.4-ubuntu18 \
    --master-machine-type n1-standard-8 \
    --master-boot-disk-type pd-ssd --master-boot-disk-size 100\
    --num-workers 0\
    --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/python/pip-install.sh \
    --metadata PIP_PACKAGES=tensorflow==2.4.0
%time

end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")


```

```
Waiting on operation [projects/big-data-cw-420116/regions/us-central1/operations/11a64304-c8fa-37e8-84e6-50a16bf977a2].

WARNING: Don't create production clusters that reference initialization actions located in the gs://goog-dataproc-initialization-actions
WARNING: The firewall rules for specified network or subnetwork would allow ingress traffic from 0.0.0.0/0, which could be a security ri
Created [https://dataproc.googleapis.com/v1/projects/big-data-cw-420116/regions/us-central1/clusters/big-data-cw-420116-cluster] Cluster
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 8.34 µs
Execution time: 153.5760896205902 seconds
```

```
# get information of the single machine cluster
!gcloud dataproc clusters describe $CLUSTER
```

```
clusterName: big-data-cw-420116-cluster
clusterUuid: 35f0437b-4971-4385-b85a-3808433b9719
config:
  configBucket: big-data-cw-420116-storage
  endpointConfig: {}
  gceClusterConfig:
    internalIpOnly: false
    metadata:
      PIP_PACKAGES: tensorflow==2.4.0
    networkUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/global/networks/default
    serviceAccountScopes:
    - https://www.googleapis.com/auth/bigquery
    - https://www.googleapis.com/auth/bigtable.admin.table
    - https://www.googleapis.com/auth/bigtable.data
    - https://www.googleapis.com/auth/cloud.useraccounts.readonly
    - https://www.googleapis.com/auth/devstorage.full_control
    - https://www.googleapis.com/auth/devstorage.read_write
    - https://www.googleapis.com/auth/logging.write
    zoneUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/zones/us-central1-c
  initializationActions:
  - executableFile: gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh
    executionTimeout: 600s
  masterConfig:
    diskConfig:
      bootDiskSizeGb: 100
      bootDiskType: pd-ssd
    imageUri: https://www.googleapis.com/compute/v1/projects/cloud-dataproc/global/images/dataproc-1-4-ubu18-20220125-170200-rc01
    instanceNames:
    - big-data-cw-420116-cluster-m
    machineTypeUri: https://www.googleapis.com/compute/v1/projects/big-data-cw-420116/zones/us-central1-c/machineTypes/n1-standard-8
    minCpuPlatform: AUTOMATIC
    numInstances: 1
    preemptibility: NON_PREEMPTIBLE
  softwareConfig:
    imageVersion: 1.4.80-ubuntu18
```

```
    properties:
      capacity-scheduler:yarn.scheduler.capacity.root.default.ordering-policy: fair
      core:fs.gs.block.size: '134217728'
      core:fs.gs.metadata.cache.enable: 'false'
      core:hadoop.ssl.enabled.protocols: TLSv1,TLSv1.1,TLSv1.2
      dataproc:dataproc.allow.zero.workers: 'true'
      distcp:mapreduce.map.java.opts: -Xmx768m
      distcp:mapreduce.map.memory.mb: '1024'
      distcp:mapreduce.reduce.java.opts: -Xmx768m
      distcp:mapreduce.reduce.memory.mb: '1024'
      hdfs:dfs.datanode.address: 0.0.0.0:9866
      hdfs:dfs.datanode.http.address: 0.0.0.0:9864
      hdfs:dfs.datanode.https.address: 0.0.0.0:9865
      hdfs:dfs.datanode.ipc.address: 0.0.0.0:9867
      hdfs:dfs.namenode.handler.count: '20'
      hdfs:dfs.namenode.http-address: 0.0.0.0:9870
      hdfs:dfs.namenode.https-address: 0.0.0.0:9871
      hdfs:dfs.namenode.lifeline.rpc-address: big-data-cw-420116-cluster-m:8050
      hdfs:dfs.namenode.secondary.http-address: 0.0.0.0:9868
      hdfs:dfs.namenode.secondary.https-address: 0.0.0.0:9869
      hdfs:dfs.namenode.service.handler.count: '10'
      hdfs:dfs.namenode.servicerpc-address: big-data-cw-420116-cluster-m:8051
      mapred-env:HADOOP_JOB_HISTORYSERVER_HEAPSIZE: '4000'
```

```
1 # submit the spark job
2 !gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_job2b.py
```

```
Job [cac7e1460f9c477cb944f548c1cdb5a5] submitted.
Waiting for job output...
2024-05-04 11:41:04.458709: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynamic library 'libcudart.so
2024-05-04 11:41:04.458757: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU se
False
24/05/04 11:41:07 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
24/05/04 11:41:07 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
24/05/04 11:41:07 INFO org.apache.spark.SparkEnv: Registering OutputCommitCoordinator
24/05/04 11:41:07 INFO org.spark_project.jetty.util.log: Logging initialized @6112ms to org.spark_project.jetty.util.log.Slf4jLog
24/05/04 11:41:07 INFO org.spark_project.jetty.server.Server: jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown; jvm 1.8.0_312-b07
24/05/04 11:41:07 INFO org.spark_project.jetty.server.Server: Started @6221ms
24/05/04 11:41:07 INFO org.spark_project.jetty.server.AbstractConnector: Started ServerConnector@69ddacff{HTTP/1.1, (http/1.1)}{0.0.0.0:
24/05/04 11:41:07 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair Scheduler configuration file not found so jobs will be sc
24/05/04 11:41:08 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to ResourceManager at big-data-cw-420116-cluster-m/10.128.0.14:
24/05/04 11:41:08 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to Application History server at big-data-cw-420116-cluster-m/
24/05/04 11:41:11 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl: Submitted application application_1714822762403_0001
Saving 2b_results.pkl to gs://big-data-cw-420116-storage
gstuil returned: 0
b'Copying file://2b_results.pkl [Content-Type=application/octet-stream]...\n/ [0 files][    0.0 B/ 11.3 KiB]
24/05/04 11:49:05 INFO org.spark_project.jetty.server.AbstractConnector: Stopped Spark@69ddacff{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
Job [cac7e1460f9c477cb944f548c1cdb5a5] finished successfully.
done: true
driverControlFilesUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/35f0437b-4971-4385-b85a-3808433b9719/jobs/cac7e146
driverOutputResourceUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/35f0437b-4971-4385-b85a-3808433b9719/jobs/cac7e1
jobUuid: 5ca9651a-fcc5-30b5-b77a-b0fca588db51
placement:
  clusterName: big-data-cw-420116-cluster
  clusterUuid: 35f0437b-4971-4385-b85a-3808433b9719
pysparkJob:
  mainPythonFileUri: gs://big-data-cw-420116-storage/google-cloud-dataproc-metainfo/35f0437b-4971-4385-b85a-3808433b9719/jobs/cac7e1460f
reference:
  jobId: cac7e1460f9c477cb944f548c1cdb5a5
  projectId: big-data-cw-420116
status:
  state: DONE
  stateStartTime: '2024-05-04T11:49:09.351993Z'
statusHistory:
- state: PENDING
  stateStartTime: '2024-05-04T11:40:59.944655Z'
- state: SETUP_DONE
  stateStartTime: '2024-05-04T11:40:59.975410Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2024-05-04T11:41:00.252670Z'
yarnApplications:
- name: spark_job2b.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-cw-420116-cluster-m:8088/proxy/application_1714822762403_0001/
```

2c) Improve efficiency (6%)

If you implemented a straightfoward version of 2a), you will **probably have an inefficiency** in your code.

Because we are reading multiple times from an RDD to read the values for the different parameters and their averages, caching existing results is important. Explain **where in the process caching can help**, and **add a call to** `RDD.cache()` to your code, if you haven't yet. Measure the the effect of using caching or not using it.

Make the **suitable change** in the code you have written above and mark them up in comments as `### TASK 2c ###`.

Explain in your report what the **reasons for this change** are and **demonstrate and interpret its effect**

```
### CODING TASK ###
%%writefile spark_job2c.py
# Setting the working environment
import datetime
import math
import numpy as np
import os
import pyspark
import pickle
import pandas as pd
import random
import string
import sys
#import scipy as sp
#import scipy.stats
import time
import tensorflow as tf
print ("TensorFlow Version" == tf.__version__)

#from matplotlib import pyplot as plt
from pyspark.sql import Row
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext

PROJECT = 'big-data-cw-420116'
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 16 # no of partitions we will use later
TARGET_SIZE = [192, 192] # target resolution for the images
CLASSES = [b'daisy', b'dandelion', b'roses', b'sunflowers', b'tulips']
          # labels for the data
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'  # prefix for output file names in my bucket
nb_images = len(tf.io.gfile.glob(GCS_PATTERN)) # number of images

# Pre-defined function. Function to parse the TFRecord
def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),  # tf.string = bytestring (not text string)
        "class": tf.io.FixedLenFeature([], tf.int64) #,   # shape [] means scalar
    }
    # decode the TFRecord
    example = tf.io.parse_single_example(example, features)
    image = tf.image.decode_jpeg(example['image'], channels=3)
    image = tf.reshape(image, [*TARGET_SIZE, 3])
    class_num = example['class']
    return image, class_num

def decode_jpeg_and_label(filepath):
    # extracts the image data and creates a class label, based on the filepath
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    # parse flower name from containing directory
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    h = tf.shape(image)[1]
    tw = TARGET_SIZE[1]
    th = TARGET_SIZE[0]
    resize_crit = (w * th) / (h * tw)
    image = tf.cond(resize_crit < 1,
                    lambda: tf.image.resize(image, [w*tw/w, h*tw/w]), # if true
                    lambda: tf.image.resize(image, [w*th/h, h*th/h])  # if false
                    )
    nw = tf.shape(image)[0]
    nh = tf.shape(image)[1]
    image = tf.image.crop_to_bounding_box(image, (nw - tw) // 2, (nh - th) // 2, tw, th)
    return image, label

# Pre-defined function. Function to load the dataset
def load_dataset(filenames):
```

```
    # read from TFRecords. For optimal performance, read from multiple
    # TFRecord files at once and set the option experimental_deterministic = False
    # to allow order-altering optimizations.
    option_no_order = tf.data.Options()
    option_no_order.experimental_deterministic = False

    dataset = tf.data.TFRecordDataset(filenames)
    dataset = dataset.with_options(option_no_order)
    dataset = dataset.map(read_tfrecord)
    return dataset

# Creating function for checking the performance
def time_configs_new(parameters_rdd):
  batch_size = parameters_rdd[0]
  batch_num = parameters_rdd[1]
  repetition = parameters_rdd[2]
```