big-data-cw-final

July 10, 2023

1 Big Data Coursework - Questions

1.1 Data Processing and Machine Learning in the Cloud

This is the **INM432 Big Data coursework 2023**. This coursework contains extended elements of **theory** and **practice**, mainly around parallelisation of tasks withSpark and a bit about parallel training using TensorFlow.

1.2 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.

1.3 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**.

1.4 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 Machine-Learning. 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**.

1.4.1 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).

1.4.2 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.

1.4.3 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**.

1.4.4 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.

1.4.5 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.

2 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.

2.0.1 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.

[30]:
```
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 2.12.0
```

2.0.2 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.

[31]:
```
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…
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/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.

[32]:
```
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.

[33]:
```
PROJECT = 'big-data-cw2-18002699'  ### USE YOUR GOOGLE CLOUD PROJECT ID HERE.␣
  ↪###
!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].
Updated property [compute/region].
Updated property [dataproc/region].
[component_manager]
disable_update_check = True
[compute]
region = us-central1
[core]
account = HellsingRed21@gmail.com
project = big-data-cw2-18002699
[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.**

[5]:
```
BUCKET = 'gs://{}-storage'.format(PROJECT)
!gsutil mb $BUCKET
```

```
Creating gs://big-data-cw2-18002699-storage/…
ServiceException: 409 A Cloud Storage bucket named 'big-data-
cw2-18002699-storage' already exists. Try another name. Bucket names must be
globally unique across all Google Cloud projects, including those outside of
your organization.
```

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.

[15]:
```
#@title Utility functions for image display **[RUN THIS TO ACTIVATE]** {␣
  ↪display-mode: "form" }
def display_9_images_from_dataset(dataset):
  plt.figure(figsize=(13,13))
```

```
  subplot=331
  for i, (image, label) in enumerate(dataset):
    plt.subplot(subplot)
    plt.axis('off')
    plt.imshow(image.numpy().astype(np.uint8))
    plt.title(str(label.numpy()), fontsize=16)
    # plt.title(label.numpy().decode(), fontsize=16)
    subplot += 1
    if i==8:
      break
  plt.tight_layout()
  plt.subplots_adjust(wspace=0.1, hspace=0.1)
  plt.show()

def display_training_curves(training, validation, title, subplot):
  if subplot%10==1: # set up the subplots on the first call
    plt.subplots(figsize=(10,10), facecolor='#F0F0F0')
    plt.tight_layout()
  ax = plt.subplot(subplot)
  ax.set_facecolor('#F8F8F8')
  ax.plot(training)
  ax.plot(validation)
  ax.set_title('model '+ title)
  ax.set_ylabel(title)
  ax.set_xlabel('epoch')
  ax.legend(['train', 'valid.'])

def dataset_to_numpy_util(dataset, N):
    dataset = dataset.batch(N)
    for images, labels in dataset:
        numpy_images = images.numpy()
        numpy_labels = labels.numpy()
        break;
    return numpy_images, numpy_labels

def title_from_label_and_target(label, correct_label):
  correct = (label == correct_label)
  return "{} [{}{}{}]".format(CLASSES[label], str(correct), ', shoud be ' if␣
 ↪not correct else '',
                              CLASSES[correct_label] if not correct else ''),␣
 ↪correct

def display_one_flower(image, title, subplot, red=False):
    plt.subplot(subplot)
    plt.axis('off')
    plt.imshow(image)
    plt.title(title, fontsize=16, color='red' if red else 'black')
```

```
    return subplot+1

def display_9_images_with_predictions(images, predictions, labels):
  subplot=331
  plt.figure(figsize=(13,13))
  classes = np.argmax(predictions, axis=-1)
  for i, image in enumerate(images):
    title, correct = title_from_label_and_target(classes[i], labels[i])
    subplot = display_one_flower(image, title, subplot, not correct)
    if i >= 8:
      break;

  plt.tight_layout()
  plt.subplots_adjust(wspace=0.1, hspace=0.1)
  plt.show()
```

2.0.3 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**.

[6]:
```
%cd
!apt-get update -qq
!apt-get install openjdk-8-jdk-headless -qq >> /dev/null # send any output to␣
 ↪null device
!tar -xzf "/content/drive/My Drive/Big_Data/data/spark/spark-3.2.0-bin-hadoop2.
 ↪7.tgz" # unpack

!pip install -q findspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/root/spark-3.2.0-bin-hadoop2.7"
import findspark
findspark.init()
import pyspark
print(pyspark.__version__)
sc = pyspark.SparkContext.getOrCreate()
print(sc)
```

```
/root
3.2.0
<SparkContext master=local[*] appName=pyspark-shell>
```

3 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.

3.1 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**.

We start by **setting some variables for the *Flowers* dataset**.

[7]:
```
GCS_PUBLIC_BUCKET = '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
```

We **read the image files** from the public GCS bucket that contains the *Flowers* dataset. **TensorFlow** has **functions** to execute glob patterns that we use to calculate the the number of images in total and per partition (rounded up as we cannont deal with parts of images).

[8]:
```
nb_images = len(tf.io.gfile.glob(GCS_PUBLIC_BUCKET)) # number of images
partition_size = math.ceil(1.0 * nb_images / PARTITIONS) # images per partition␣
 ↪(float)
print("GCS_PATTERN matches {} images, to be divided into {} partitions with up␣
 ↪to {} images each.".format(nb_images, PARTITIONS, partition_size))
```

```
GCS_PATTERN matches 3670 images, to be divided into 16 partitions with up to 230
images each.
```

3.1.1 Map functions

In order to read use the images for learning, they need to be **preprocessed** (decoded, resized, cropped, and potentially recompressed). Below are **map functions** for these steps. You **don't need to study** the **internals of these functions** in detail.

[9]:
```
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
```

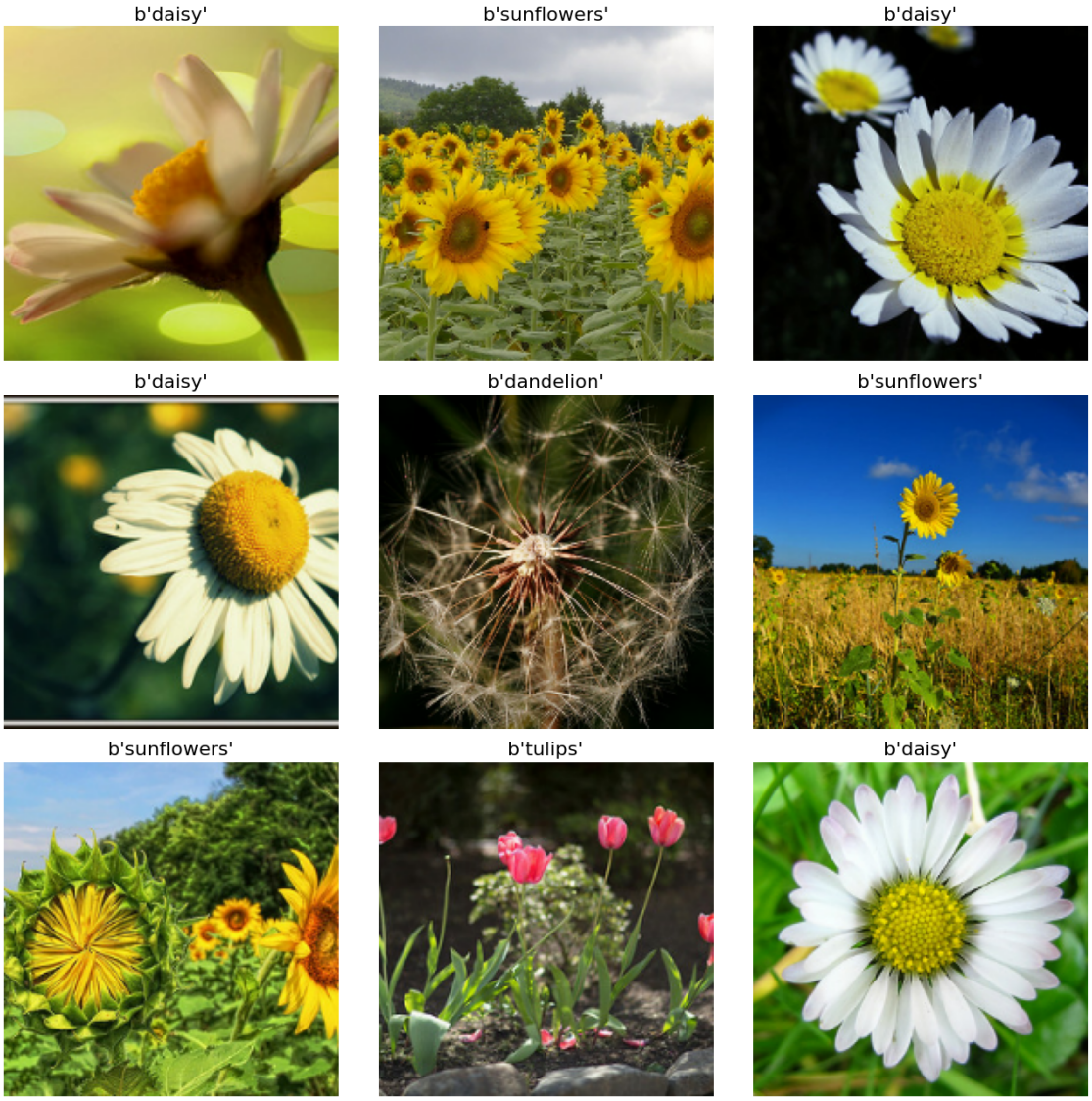
```
def resize_and_crop_image(image, label):
    # Resizes and cropd using "fill" algorithm:
    # always make sure the resulting image is cut out from the source image
    # so that it fills the TARGET_SIZE entirely with no black bars
    # and a preserved aspect ratio.
    w = tf.shape(image)[0]
    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
```

With `tf.data`, we can apply decoding and resizing as map functions.

[14]:
```
dsetFiles = tf.data.Dataset.list_files(GCS_PUBLIC_BUCKET) # This also shuffles␣
 ↪the images
dsetDecoded = dsetFiles.map(decode_jpeg_and_label)
dsetResized = dsetDecoded.map(resize_and_crop_image)
```

We can also look at some images using the image display function defined above (the one with the hidden code).

[16]:
```
display_9_images_from_dataset(dsetResized)
```



Now, let's test continuous reading from the dataset. We can see that reading the first 100 files already takes some time.

[17]:
```
sample_set = dsetResized.batch(10).take(10) # take 10 batches of 10 images for␣
 ↪testing
for image, label in sample_set:
    print("Image batch shape {}, {})".format(image.numpy().shape,
        [lbl.decode('utf8') for lbl in label.numpy()]))
```

```
Image batch shape (10, 192, 192, 3), ['sunflowers', 'tulips', 'daisy', 'tulips',
'roses', 'roses', 'dandelion', 'sunflowers', 'dandelion', 'roses'])
Image batch shape (10, 192, 192, 3), ['sunflowers', 'tulips', 'dandelion',
'dandelion', 'tulips', 'daisy', 'roses', 'dandelion', 'sunflowers', 'tulips'])
```

```
Image batch shape (10, 192, 192, 3), ['roses', 'sunflowers', 'roses', 'roses',
'tulips', 'daisy', 'daisy', 'sunflowers', 'daisy', 'dandelion'])
Image batch shape (10, 192, 192, 3), ['tulips', 'tulips', 'tulips', 'daisy',
'sunflowers', 'roses', 'roses', 'daisy', 'daisy', 'roses'])
Image batch shape (10, 192, 192, 3), ['sunflowers', 'roses', 'tulips',
'dandelion', 'dandelion', 'dandelion', 'roses', 'daisy', 'sunflowers',
'tulips'])
Image batch shape (10, 192, 192, 3), ['daisy', 'daisy', 'tulips', 'tulips',
'sunflowers', 'tulips', 'tulips', 'dandelion', 'dandelion', 'tulips'])
Image batch shape (10, 192, 192, 3), ['daisy', 'sunflowers', 'dandelion',
'dandelion', 'sunflowers', 'dandelion', 'tulips', 'daisy', 'daisy',
'dandelion'])
Image batch shape (10, 192, 192, 3), ['tulips', 'daisy', 'roses', 'dandelion',
'sunflowers', 'daisy', 'daisy', 'roses', 'sunflowers', 'tulips'])
Image batch shape (10, 192, 192, 3), ['daisy', 'roses', 'dandelion', 'tulips',
'dandelion', 'daisy', 'roses', 'tulips', 'tulips', 'dandelion'])
Image batch shape (10, 192, 192, 3), ['tulips', 'roses', 'sunflowers',
'dandelion', 'dandelion', 'dandelion', 'sunflowers', 'roses', 'tulips',
'dandelion'])
```

3.2 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**.

3.2.1 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**.

[18]:
```
# 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,), ['tulips', 'sunflowers', 'roses', 'sunflowers',
'tulips', 'tulips', 'daisy', 'tulips', 'sunflowers', 'tulips'])
Image batch shape (10,), ['tulips', 'roses', 'dandelion', 'daisy', 'sunflowers',
'tulips', 'sunflowers', 'sunflowers', 'roses', 'sunflowers'])
Image batch shape (10,), ['daisy', 'daisy', 'dandelion', 'tulips', 'daisy',
'dandelion', 'sunflowers', 'sunflowers', 'daisy', 'sunflowers'])
Image batch shape (10,), ['roses', 'tulips', 'tulips', 'tulips', 'dandelion',
'sunflowers', 'dandelion', 'sunflowers', 'tulips', 'dandelion'])
Image batch shape (10,), ['daisy', 'tulips', 'dandelion', 'dandelion',
'sunflowers', 'roses', 'dandelion', 'sunflowers', 'sunflowers', 'tulips'])
Image batch shape (10,), ['daisy', 'dandelion', 'daisy', 'dandelion',
```

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

3.2.2 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.

[28]:
```
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.

[20]:
```
# 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
```

3.2.3 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.

[21]:
```
GCS_PUBLIC = 'gs://flowers-public/tfrecords-jpeg-192x192-2/'
# remove the line above to use your own files that you generated above

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 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


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

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

[22]:
```
batched_dataset = datasetTfrec.batch(10)
sample_set = batched_dataset.take(10)
for image, label in sample_set:
    print("Image batch shape {}, {})".format(image.numpy().shape, \
                          [str(lbl) for lbl in label.numpy()]))
```

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

```
Image batch shape (10, 192, 192, 3), ['1', '1', '1', '1', '2', '2', '2', '2',
'2', '2'])
Image batch shape (6, 192, 192, 3), ['2', '2', '3', '3', '3', '3'])
```

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

3.3 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.

3.3.1 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%)

[26]:
```
### CODING TASK ###
##Task i)
def decode_jpeg_and_label(filepath):#This function passes the filepath,␣
 ↪extracts image data and creates a class label from the filepath
    # Extract flower name from directory
    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(input_layer):  #Replaced the two argument variables␣
 ↪with one input layer and then extracting the data accordingly in the␣
 ↪following line.
    image, label = input_layer
    # 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(input_layer): #Replaced the two input variable with one␣
 ↪input and then deriving the image and label the same way.
    image, label = input_layer
    # 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
```

[35]:
```
# Task ii)
FILEPATH = tf.io.gfile.glob(GCS_PUBLIC)  # Defined the file path variable
RDD = sc.parallelize(FILEPATH)  # Make an RDD

# Task iii)
SampledRDD = RDD.sample(False, 0.02)  # Sample the RDD

# Task iv)
RDD_decoded = SampledRDD.map(decode_jpeg_and_label)
RDD_resized = RDD_decoded.map(resize_and_crop_image)
RDD_recompressed = RDD_resized.map(recompress_image)

print(RDD_recompressed.take(2))
```

```
[]
```

[36]:
```
# Task v)
def write_tfrecords_te(index, partition):
    filename = GCS_OUTPUT + "{}.tfrec".format(index)  # Using the previous␣
 ↪write_tf_records function to generate this.
```

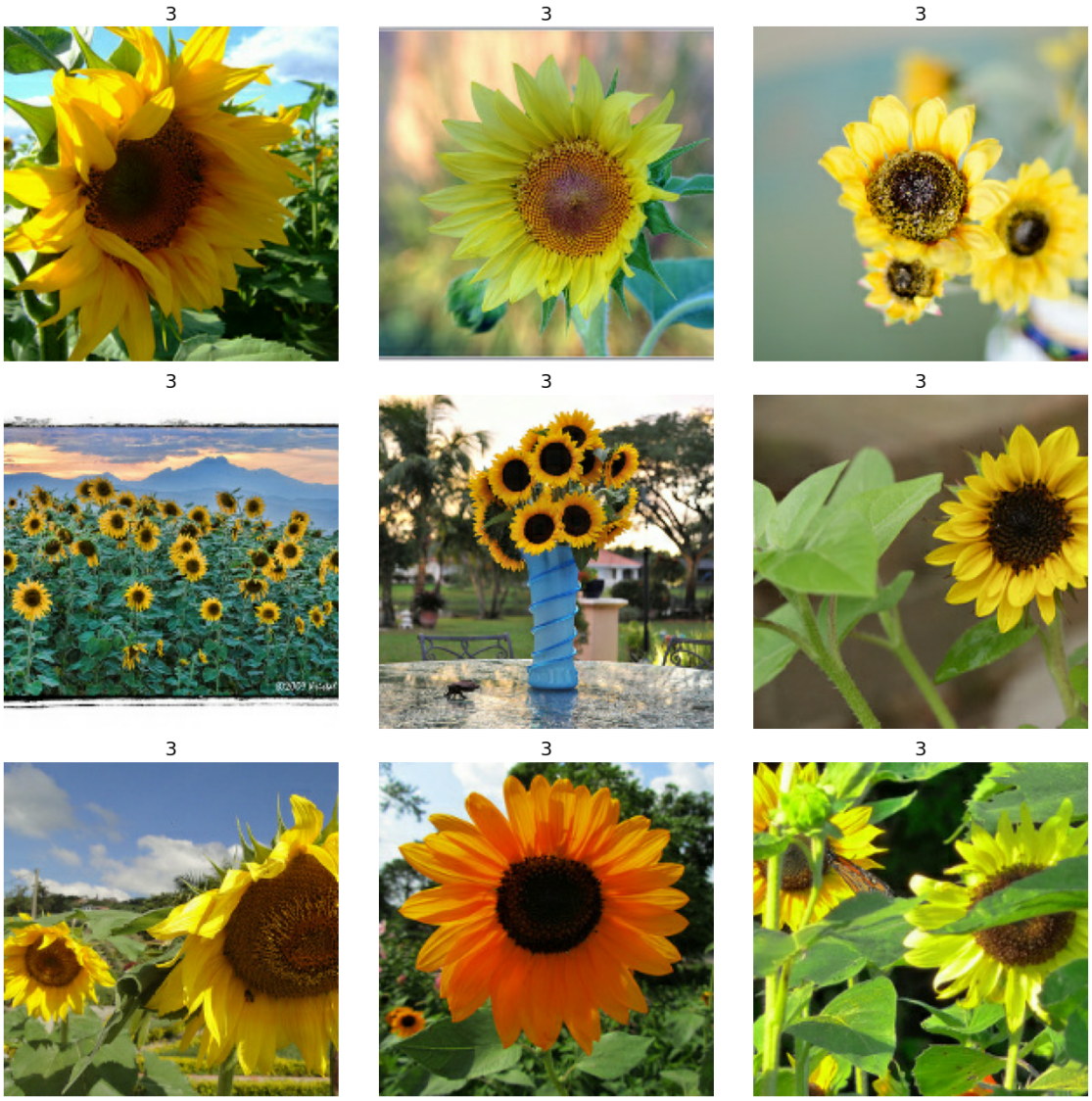
```
    with tf.io.TFRecordWriter(filename) as out_file:
        for element in partition:
            image = element[0]
            label = element[1]
            example = to_tfrecord(out_file,
                                  image.numpy(),  # Re-compressed image:␣
 ↪already a byte string
                                  label.numpy()  # , height.numpy()[i], width.
 ↪numpy()[i]
                                  )
            out_file.write(example.SerializeToString())
    yield filename


te = RDD_recompressed.mapPartitionsWithIndex(write_tfrecords_te)
print(te.collect())
```

```
['gs://big-data-cw2-18002699-storage/tfrecords-jpeg-192x192-2/flowers0.tfrec',
'gs://big-data-cw2-18002699-storage/tfrecords-jpeg-192x192-2/flowers1.tfrec']
```

3.3.2 1b) Testing (3%)

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

[37]:
```
### CODING TASK ###
tf_records = tf.io.gfile.glob(GCS_OUTPUT + "*.tfrec")
data = load_dataset(tf_records)
display_9_images_from_dataset(data)
```



ii) Write your code above into a file using the *cell magic* `%%writefile spark_write_tfrec.py` at the beginning of the file. Then, run the file locally in Spark.

[40]:
```
%%writefile spark_write_tfrec.py
import numpy as np
import tensorflow as tf
import pyspark

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): #, height, width):
    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 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(input_layer):  #Replaced the two input variable with␣
 ↪one input and defined image and label underneath.
    image, label = input_layer
    # Resizes and cropd using "fill" algorithm:
    # alwimage, label = input_layerays 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(input_layer): #Replaced the two input variable with one␣
 ↪input and defined image and label underneath.
```

```
    image, label = input_layer
    # 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

def write_tfrecords_modified(index, data_partition):
  tfrecord_filename = GCS_OUTPUT + "{}.tfrec".format(index)
  with tf.io.TFRecordWriter(tfrecord_filename) as tfrecord_file:
    for data_element in data_partition:
      img = data_element[0]
      lbl = data_element[1]
      example = to_tfrecord(tfrecord_file,
                            img.numpy(), # re-compressed image: already a byte␣
 ↪string
                            lbl.numpy() #, height.numpy()[i], width.numpy()[i]
                            )
      tfrecord_file.write(example.SerializeToString())
  return [tfrecord_filename]

PROJECT = 'big-data-cw2-18002699'
GCS_PATTERN = 'gs://flowers-public/*/*.jpg' # glob  pattern for input files
PARTITIONS = 2 # 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 (folder names)
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'

#Task 1aii

# Make into RDD
sc = pyspark.SparkContext.getOrCreate()
file = tf.io.gfile.glob(GCS_PATTERN)
file_RDD = sc.parallelize(file)

#Task 1aiii

# Sample the RDD with a factor of 0.02 for short tests
RDD_Sampled = file_RDD.sample(False,0.02)

# Decode the RDD
Decoded_RDD = RDD_Sampled.map(decode_jpeg_and_label)

# Resize the RDD
Resized_RDD = Decoded_RDD.map(resize_and_crop_image)
```

```
# Recompress the RDD
Recompressed_RDD = Resized_RDD.map(recompress_image)

Partitioned_RDD = Recompressed_RDD.repartition(PARTITIONS)
tf_records_RDD = Partitioned_RDD.
  ↪mapPartitionsWithIndex(write_tfrecords_modified)
tf_records = tf_records_RDD.collect()
```

```
Overwriting spark_write_tfrec.py
```

3.3.3 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%)

[39]:
```
!gcloud dataproc clusters create $CLUSTER \
--image-version 1.4-ubuntu18 --single-node \
--master-machine-type n1-standard-8 \
--master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
--max-idle 3600s \
--initialization-actions gs://goog-dataproc-initialization-actions-$REGION/
  ↪python/pip-install.sh \
--metadata PIP_PACKAGES=tensorflow==2.4.0
```

```
Waiting on operation [projects/big-data-cw2-18002699/regions/us-
central1/operations/3da500de-5aff-39ef-95e1-9a5e73b72e00].

WARNING: Don't create production clusters that reference
initialization actions located in the gs://goog-dataproc-initialization-actions-
REGION public buckets. These scripts are provided as reference implementations,
and they are synchronized with ongoing GitHub repository changes-a new version
of a initialization action in public buckets may break your cluster creation.
Instead, copy the following initialization actions from public buckets into your
```

```
bucket : gs://goog-dataproc-initialization-actions-us-central1/python/pip-
install.sh
WARNING: Failed to validate permissions required for default service
account: '92774894602-compute@developer.gserviceaccount.com'. Cluster creation
could still be successful if required permissions have been granted to the
respective service accounts as mentioned in the document
https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/service-
accounts#dataproc_service_accounts_2. This could be due to Cloud Resource
Manager API hasn't been enabled in your project '92774894602' before or it is
disabled. Enable it by visiting 'https://console.developers.google.com/apis/api/
cloudresourcemanager.googleapis.com/overview?project=92774894602'.
WARNING: The firewall rules for specified network or subnetwork would
allow ingress traffic from 0.0.0.0/0, which could be a security risk.
Created [https://dataproc.googleapis.com/v1/projects/big-data-
cw2-18002699/regions/us-central1/clusters/big-data-cw2-18002699-cluster] Cluster
placed in zone [us-central1-a].
```

Run the script in the cloud and test the output.

[41]:
```
### CODING TASK ###
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_write_tfrec.py
```

```
Job [f6e5e0a303c944249852db604ffb118c] submitted.
Waiting for job output…
2023-07-10 14:18:47.616407: W
tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/usr/lib/hadoop/lib/native
2023-07-10 14:18:47.616444: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
23/07/10 14:18:50 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
23/07/10 14:18:50 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
23/07/10 14:18:50 INFO org.apache.spark.SparkEnv: Registering
OutputCommitCoordinator
23/07/10 14:18:50 INFO org.spark_project.jetty.util.log: Logging initialized
@5567ms to org.spark_project.jetty.util.log.Slf4jLog
23/07/10 14:18:51 INFO org.spark_project.jetty.server.Server:
jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown; jvm 1.8.0_312-b07
23/07/10 14:18:51 INFO org.spark_project.jetty.server.Server: Started @5685ms
23/07/10 14:18:51 INFO org.spark_project.jetty.server.AbstractConnector: Started
ServerConnector@2c627e8d{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
23/07/10 14:18:51 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair
Scheduler configuration file not found so jobs will be scheduled in FIFO order.
To use fair scheduling, configure pools in fairscheduler.xml or set
spark.scheduler.allocation.file to a file that contains the configuration.
23/07/10 14:18:52 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to
ResourceManager at big-data-cw2-18002699-cluster-m/10.128.0.10:8032
```

```
23/07/10 14:18:52 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to
Application History server at big-data-cw2-18002699-cluster-m/10.128.0.10:10200
23/07/10 14:18:54 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl:
Submitted application application_1688998473787_0001
23/07/10 14:19:03 WARN org.apache.spark.scheduler.TaskSetManager: Stage 0
contains a task of very large size (134 KB). The maximum recommended task size
is 100 KB.
23/07/10 14:19:13 INFO org.spark_project.jetty.server.AbstractConnector: Stopped
Spark@2c627e8d{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
Job [f6e5e0a303c944249852db604ffb118c] finished successfully.
done: true
driverControlFilesUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/f6e5e0a303c944249852db604ffb118c/
driverOutputResourceUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/f6e5e0a303c944249852db604ffb118c/driveroutput
jobUuid: eb50b6c4-4bcd-3d16-92bf-3858adde6558
placement:
  clusterName: big-data-cw2-18002699-cluster
  clusterUuid: 8e600862-d06f-44e9-b9ff-76c19049ad9e
pysparkJob:
  mainPythonFileUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/f6e5e0a303c944249852db604ffb118c/staging/spark_write_tfre
c.py
reference:
  jobId: f6e5e0a303c944249852db604ffb118c
  projectId: big-data-cw2-18002699
status:
  state: DONE
  stateStartTime: '2023-07-10T14:19:14.184721Z'
statusHistory:
- state: PENDING
  stateStartTime: '2023-07-10T14:18:43.627698Z'
- state: SETUP_DONE
  stateStartTime: '2023-07-10T14:18:43.675451Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2023-07-10T14:18:44.003522Z'
yarnApplications:
- name: spark_write_tfrec.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-
cw2-18002699-cluster-m:8088/proxy/application_1688998473787_0001/
```

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%)

[42]:
```
!gcloud dataproc clusters create $CLUSTER \
 --image-version 1.4-ubuntu18 \
 --num-masters 1 \
 --master-machine-type n1-standard-1 \
 --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
  --num-workers 7 \
  --worker-machine-type n1-standard-1 \
  --worker-boot-disk-size 285 \
  --max-idle 3600s \
  --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/
↪python/pip-install.sh \
  --metadata PIP_PACKAGES=tensorflow==2.4.0
```

```
ERROR: (gcloud.dataproc.clusters.create) INVALID_ARGUMENT:
Insufficient 'IN_USE_ADDRESSES' quota. Requested 8.0, available 7.0.
```

[43]:
```
### CODING TASK ###
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER \spark_write_tfrec.py
```

```
Job [4d840961385048e6b3e8320278a7a0c6] submitted.
Waiting for job output…
2023-07-10 15:17:24.534401: W
tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/usr/lib/hadoop/lib/native
2023-07-10 15:17:24.534455: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
23/07/10 15:17:27 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
23/07/10 15:17:27 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
23/07/10 15:17:27 INFO org.apache.spark.SparkEnv: Registering
OutputCommitCoordinator
23/07/10 15:17:27 INFO org.spark_project.jetty.util.log: Logging initialized
@4978ms to org.spark_project.jetty.util.log.Slf4jLog
```

```
23/07/10 15:17:27 INFO org.spark_project.jetty.server.Server:
jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown; jvm 1.8.0_312-b07
23/07/10 15:17:27 INFO org.spark_project.jetty.server.Server: Started @5103ms
23/07/10 15:17:27 INFO org.spark_project.jetty.server.AbstractConnector: Started
ServerConnector@7655c0b4{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
23/07/10 15:17:27 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair
Scheduler configuration file not found so jobs will be scheduled in FIFO order.
To use fair scheduling, configure pools in fairscheduler.xml or set
spark.scheduler.allocation.file to a file that contains the configuration.
23/07/10 15:17:28 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to
ResourceManager at big-data-cw2-18002699-cluster-m/10.128.0.10:8032
23/07/10 15:17:28 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to
Application History server at big-data-cw2-18002699-cluster-m/10.128.0.10:10200
23/07/10 15:17:30 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl:
Submitted application application_1688998473787_0002
23/07/10 15:17:38 WARN org.apache.spark.scheduler.TaskSetManager: Stage 0
contains a task of very large size (134 KB). The maximum recommended task size
is 100 KB.
23/07/10 15:17:50 INFO org.spark_project.jetty.server.AbstractConnector: Stopped
Spark@7655c0b4{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
Job [4d840961385048e6b3e8320278a7a0c6] finished successfully.
done: true
driverControlFilesUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/4d840961385048e6b3e8320278a7a0c6/
driverOutputResourceUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/4d840961385048e6b3e8320278a7a0c6/driveroutput
jobUuid: d0d8e44f-038b-3345-89a3-9b3543cc2842
placement:
  clusterName: big-data-cw2-18002699-cluster
  clusterUuid: 8e600862-d06f-44e9-b9ff-76c19049ad9e
pysparkJob:
  mainPythonFileUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/4d840961385048e6b3e8320278a7a0c6/staging/spark_write_tfre
c.py
reference:
  jobId: 4d840961385048e6b3e8320278a7a0c6
  projectId: big-data-cw2-18002699
status:
  state: DONE
  stateStartTime: '2023-07-10T15:17:51.364039Z'
statusHistory:
- state: PENDING
  stateStartTime: '2023-07-10T15:17:20.984766Z'
- state: SETUP_DONE
  stateStartTime: '2023-07-10T15:17:21.025256Z'
```

```
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2023-07-10T15:17:21.281772Z'
yarnApplications:
- name: spark_write_tfrec.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-
cw2-18002699-cluster-m:8088/proxy/application_1688998473787_0002/
```

3.3.4 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.

[44]:
```
%%writefile spark_write_tfrec_16_partitions.py
import numpy as np
import tensorflow as tf
import pyspark

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): #, height, width):
    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 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(input_layer):  #Replaced the two input variable with␣
 ↪one input and defined image and label underneath.
    image, label = input_layer
    # Resizes and cropd using "fill" algorithm:
    # alwimage, label = input_layerays 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(input_layer): #Replaced the two input variable with one␣
 ↪input and defined image and label underneath.
    image, label = input_layer
    # 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

def write_tfrecords_modified(index, data_partition):
  tfrecord_filename = GCS_OUTPUT + "{}.tfrec".format(index)
  with tf.io.TFRecordWriter(tfrecord_filename) as tfrecord_file:
    for data_element in data_partition:
      img = data_element[0]
      lbl = data_element[1]
      example = to_tfrecord(tfrecord_file,
                            img.numpy(), # re-compressed image: already a byte␣
 ↪string
                            lbl.numpy() #, height.numpy()[i], width.numpy()[i]
                            )
      tfrecord_file.write(example.SerializeToString())
  return [tfrecord_filename]

PROJECT = 'big-data-cw2-18002699'
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 (folder names)
BUCKET = 'gs://{}-storage'.format(PROJECT)
GCS_OUTPUT = BUCKET + '/tfrecords-jpeg-192x192-2/flowers'

#Task 1aii

# Make into RDD
sc = pyspark.SparkContext.getOrCreate()
file = tf.io.gfile.glob(GCS_PATTERN)
file_RDD = sc.parallelize(file,16)

#Task 1aiii

# Sample the RDD with a factor of 0.02 for short tests
RDD_Sampled = file_RDD.sample(False,0.02)

# Decode the RDD
Decoded_RDD = RDD_Sampled.map(decode_jpeg_and_label)

# Resize the RDD
Resized_RDD = Decoded_RDD.map(resize_and_crop_image)

# Recompress the RDD
Recompressed_RDD = Resized_RDD.map(recompress_image)
```

```
tf_records_RDD = Recompressed_RDD.
  ↪mapPartitionsWithIndex(write_tfrecords_modified)
tf_records = tf_records_RDD.collect()
```

```
Writing spark_write_tfrec_16_partitions.py
```

[45]:
```
!gcloud dataproc clusters create $CLUSTER \
--image-version 1.4-ubuntu18 \
--num-masters 1 \
--master-machine-type n1-standard-2 \
--master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
--num-workers 3 \
--worker-machine-type n1-standard-2 \
--worker-boot-disk-size 666 \
--max-idle 3600s\
--initialization-actions gs://goog-dataproc-initialization-actions-$REGION/
  ↪python/pip-install.sh \
--metadata PIP_PACKAGES=tensorflow==2.4.0
```

```
ERROR: (gcloud.dataproc.clusters.create) ALREADY_EXISTS: Already
exists: Failed to create cluster: Cluster projects/big-data-
cw2-18002699/regions/us-central1/clusters/big-data-cw2-18002699-cluster
```

[46]:
```
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER␣
  ↪\spark_write_tfrec_16_partitions.py
```

```
Job [d494871f09fd44e4bfb6c45c89aaa34f] submitted.
Waiting for job output…
2023-07-10 15:18:16.635236: W
tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/usr/lib/hadoop/lib/native
2023-07-10 15:18:16.635290: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
23/07/10 15:18:19 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
23/07/10 15:18:19 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
23/07/10 15:18:19 INFO org.apache.spark.SparkEnv: Registering
OutputCommitCoordinator
23/07/10 15:18:19 INFO org.spark_project.jetty.util.log: Logging initialized
@4936ms to org.spark_project.jetty.util.log.Slf4jLog
23/07/10 15:18:19 INFO org.spark_project.jetty.server.Server:
jetty-9.4.z-SNAPSHOT; built: unknown; git: unknown; jvm 1.8.0_312-b07
23/07/10 15:18:19 INFO org.spark_project.jetty.server.Server: Started @5048ms
23/07/10 15:18:19 INFO org.spark_project.jetty.server.AbstractConnector: Started
ServerConnector@30a43356{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
23/07/10 15:18:19 WARN org.apache.spark.scheduler.FairSchedulableBuilder: Fair
Scheduler configuration file not found so jobs will be scheduled in FIFO order.
```

```
To use fair scheduling, configure pools in fairscheduler.xml or set
spark.scheduler.allocation.file to a file that contains the configuration.
23/07/10 15:18:20 INFO org.apache.hadoop.yarn.client.RMProxy: Connecting to
ResourceManager at big-data-cw2-18002699-cluster-m/10.128.0.10:8032
23/07/10 15:18:20 INFO org.apache.hadoop.yarn.client.AHSProxy: Connecting to
Application History server at big-data-cw2-18002699-cluster-m/10.128.0.10:10200
23/07/10 15:18:22 INFO org.apache.hadoop.yarn.client.api.impl.YarnClientImpl:
Submitted application application_1688998473787_0003
23/07/10 15:18:40 INFO org.spark_project.jetty.server.AbstractConnector: Stopped
Spark@30a43356{HTTP/1.1, (http/1.1)}{0.0.0.0:4040}
Job [d494871f09fd44e4bfb6c45c89aaa34f] finished successfully.
done: true
driverControlFilesUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/d494871f09fd44e4bfb6c45c89aaa34f/
driverOutputResourceUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/d494871f09fd44e4bfb6c45c89aaa34f/driveroutput
jobUuid: 92d68f40-967d-3a34-9c7d-eea0f1467a08
placement:
  clusterName: big-data-cw2-18002699-cluster
  clusterUuid: 8e600862-d06f-44e9-b9ff-76c19049ad9e
pysparkJob:
  mainPythonFileUri: gs://dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/d494871f09fd44e4bfb6c45c89aaa34f/staging/spark_write_tfre
c_16_partitions.py
reference:
  jobId: d494871f09fd44e4bfb6c45c89aaa34f
  projectId: big-data-cw2-18002699
status:
  state: DONE
  stateStartTime: '2023-07-10T15:18:41.053705Z'
statusHistory:
- state: PENDING
  stateStartTime: '2023-07-10T15:18:13.378720Z'
- state: SETUP_DONE
  stateStartTime: '2023-07-10T15:18:13.428807Z'
- details: Agent reported job success
  state: RUNNING
  stateStartTime: '2023-07-10T15:18:13.633207Z'
yarnApplications:
- name: spark_write_tfrec_16_partitions.py
  progress: 1.0
  state: FINISHED
  trackingUrl: http://big-data-
cw2-18002699-cluster-m:8088/proxy/application_1688998473787_0003/
```

4 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**.

4.1 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.

[47]:
```
# 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 files (dataset4) and the TFRecord files (datasetTfrec).

[48]:
```
[res,par] = time_configs(dataset4, batch_sizes, batch_numbers, repetitions)
print(res)
print(par)

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

[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'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'>
Image batch shape (2,), ["b'dandelion'", "b'roses'"]) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
bs: 2, ds: 6, rep: 1
Image batch shape (2,), ["b'daisy'", "b'dandelion'"]) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'sunflowers'", "b'roses'"]) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'tulips'", "b'tulips'"]) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2,), ["b'daisy'", "b'dandelion'"]) <_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'dandelion'", "b'dandelion'"]) <_io.TextIOWrapper
```

```
name='/dev/null' mode='w' encoding='UTF-8'>
bs: 4, ds: 3, rep: 1
Image batch shape (4,), ["b'dandelion'", "b'sunflowers'", "b'tulips'",
"b'tulips'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'tulips'", "b'daisy'", "b'daisy'", "b'sunflowers'"])
<_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'roses'", "b'roses'", "b'roses'", "b'roses'"])
<_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
bs: 4, ds: 6, rep: 1
Image batch shape (4,), ["b'tulips'", "b'dandelion'", "b'tulips'", "b'tulips'"])
<_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'dandelion'", "b'sunflowers'", "b'dandelion'",
"b'daisy'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'sunflowers'", "b'sunflowers'", "b'dandelion'",
"b'tulips'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'sunflowers'", "b'dandelion'", "b'dandelion'",
"b'sunflowers'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'roses'", "b'roses'", "b'daisy'", "b'tulips'"])
<_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4,), ["b'sunflowers'", "b'tulips'", "b'dandelion'",
"b'roses'"]) <_io.TextIOWrapper name='/dev/null' mode='w' encoding='UTF-8'>
total time: 6.210814476013184
[[[ 5.09200804]
  [ 7.87737922]]

 [[ 8.73896171]
  [11.34693952]]]
[[[[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', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (2, 192, 192, 3), ['3', '3']) <_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', '3', '3', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
bs: 4, ds: 6, 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', '3', '3', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4, 192, 192, 3), ['3', '3', '3', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4, 192, 192, 3), ['3', '3', '3', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
Image batch shape (4, 192, 192, 3), ['3', '3', '3', '3']) <_io.TextIOWrapper
name='/dev/null' mode='w' encoding='UTF-8'>
total time: 1.2523729801177979
[[[34.43038695]
  [35.10636749]]

 [[36.81959356]
  [59.31684506]]]
[[[[2. 3. 1.]]

  [[2. 6. 1.]]]


 [[[4. 3. 1.]]

  [[4. 6. 1.]]]]
```

4.2 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).

4.2.1 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.

TFrec

[49]:
```
### CODING TASK
#Task 2ai
import pandas as pd
import pyspark
import pickle
from google.cloud import storage

GCS_PUBLIC_BUCKET = '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

def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "class": tf.io.FixedLenFeature([], tf.int64)
    }
    # 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 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

def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN) #Image Files
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(input_layer):
    image, label = input_layer
    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
```

```
batch_sizes = [2,4,6,8]
batch_numbers = [3,6,12,18]
repetitions=[1,2,3]

parameter_list = [[i,j,k] for i in batch_sizes for j in batch_numbers for k in␣
 ↪repetitions]

#Task 2aii
sc=pyspark.SparkContext.getOrCreate()
RDD_parameter = sc.parallelize(parameter_list, 16) #creates an RDD

def time_configs_combined(parameters_rdd):
  batch_size, batch_count, repetition_count = parameters_rdd

  filenames = tf.io.gfile.glob(GCS_PUBLIC_BUCKET + "*.tfrec")
  dataset = load_dataset(filenames)

  batched_dataset = dataset.batch(batch_size)
  sample_dataset = batched_dataset.take(batch_count)

  timings = []
  for rep in range(repetition_count):
    start_time = time.time()
    for picture in sample_dataset:
      print('string', file=open("/dev/null", mode='w'))
    end_time = time.time()
    reading_duration = end_time - start_time
    throughput = float((batch_size * batch_count) / reading_duration)
    dataset_size = batch_size * batch_count
    timings.append([batch_size, batch_count, repetition_count, dataset_size,␣
 ↪throughput])
  return timings

#Task 2aiii
tfrec_rdd = RDD_parameter.map(time_configs_combined)
### TASK 2c ###
tfrec_rdd.cache()

#Task 2aiv
# For TFRec dataset
tfrec_images_per_second = tfrec_rdd.flatMap(lambda x:␣
 ↪[(str(x[0][0])+'_'+str(x[0][1])+'_'+str(x[0][2]), x[0][4])])
### TASK 2c ###
tfrec_images_per_second.cache()
tfrec_results = tfrec_images_per_second.collect()
```

```
#Task 2av)
# For TFRec dataset
tfrec_images_per_second_avg = tfrec_images_per_second \
    .mapValues(lambda x: (x, 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: x[0] / x[1])
tfrec_avg_results = tfrec_images_per_second_avg.collect()


#Task 2avi)
# Define your bucket name and destination file name
PROJECT = 'big-data-cw2-18002699'
destination_blob_name = "average_results_tfrec.pkl"

def upload_to_bucket(BUCKET, destination_blob_name, data):
    storage_client = storage.Client()
    bucket = storage_client.get_bucket('{}-storage'.format(PROJECT))
    blob = bucket.blob(destination_blob_name)

    # Write the data to a temporary pickle file
    with blob.open('wb') as f:
        pickle.dump(data, f)


# Upload the results to the bucket
upload_to_bucket(PROJECT, destination_blob_name, tfrec_avg_results)
```

TFRec File

[50]:
```
#Task 2avii)

%%writefile spark_job_tfrec.py
### CODING TASK
#Task 2ai
import pandas as pd
import pyspark
import pickle
import tensorflow as tf
from google.cloud import storage
import time

GCS_PUBLIC_BUCKET = '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


def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "class": tf.io.FixedLenFeature([], tf.int64)
    }
    # 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 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

def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN) #Image Files
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(input_layer):
    image, label = input_layer
    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

batch_sizes = [2,4,6,8]
batch_numbers = [3,6,12,18]
repetitions=[1,2,3]

parameter_list = [[i,j,k] for i in batch_sizes for j in batch_numbers for k in␣
 ↪repetitions]

#Task 2aii
sc=pyspark.SparkContext.getOrCreate()
RDD_parameter = sc.parallelize(parameter_list, 16) #creates an RDD

def time_configs_combined(parameters_rdd):
  batch_size, batch_count, repetition_count = parameters_rdd

  filenames = tf.io.gfile.glob(GCS_PUBLIC_BUCKET + "*.tfrec")
  dataset = load_dataset(filenames)

  batched_dataset = dataset.batch(batch_size)
  sample_dataset = batched_dataset.take(batch_count)

  timings = []
  for rep in range(repetition_count):
    start_time = time.time()
    for picture in sample_dataset:
      print('string', file=open("/dev/null", mode='w'))
    end_time = time.time()
    reading_duration = end_time - start_time
    throughput = float((batch_size * batch_count) / reading_duration)
    dataset_size = batch_size * batch_count
    timings.append([batch_size, batch_count, repetition_count, dataset_size,␣
 ↪throughput])
  return timings

#Task 2aiii
tfrec_rdd = RDD_parameter.map(time_configs_combined)
### TASK 2c ###
```

```
tfrec_rdd.cache()

#Task 2aiv
# For TFRec dataset
tfrec_images_per_second = tfrec_rdd.flatMap(lambda x:␣
 ↪[(str(x[0][0])+'_'+str(x[0][1])+'_'+str(x[0][2]), x[0][4])])
### TASK 2c ###
tfrec_images_per_second.cache()
tfrec_results = tfrec_images_per_second.collect()


#Task 2av)
# For TFRec dataset
tfrec_images_per_second_avg = tfrec_images_per_second \
    .mapValues(lambda x: (x, 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: x[0] / x[1])
tfrec_avg_results = tfrec_images_per_second_avg.collect()


#Task 2avi)
# Define your bucket name and destination file name
PROJECT = 'big-data-cw2-18002699'
destination_blob_name = "average_results_tfrec.pkl"

def upload_to_bucket(BUCKET, destination_blob_name, data):
    storage_client = storage.Client()
    bucket = storage_client.get_bucket('{}-storage'.format(PROJECT))
    blob = bucket.blob(destination_blob_name)

    # Write the data to a temporary pickle file
    with blob.open('wb') as f:
        pickle.dump(data, f)


# Upload the results to the bucket
upload_to_bucket(PROJECT, destination_blob_name, tfrec_avg_results)
```

```
Writing spark_job_tfrec.py
```

Decoded

[51]:
```
### CODING TASK
#Task 2ai
import pandas as pd
import pyspark
import pickle
from google.cloud import storage
```

```
GCS_PUBLIC_BUCKET = 'gs://flowers-public/*/*.jpg' # glob  pattern for input␣
 ↪files

def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "class": tf.io.FixedLenFeature([], tf.int64)
    }
    # 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 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

def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN) #Image Files
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(input_layer):
    image, label = input_layer
    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

batch_sizes = [2,4,6,8]
batch_numbers = [3,6,12,18]
repetitions=[1,2,3]

parameter_list = [[i,j,k] for i in batch_sizes for j in batch_numbers for k in␣
 ↪repetitions]

#Task 2aii
sc=pyspark.SparkContext.getOrCreate()
RDD_parameter = sc.parallelize(parameter_list) #creates an RDD

def time_configs_combined(parameters):
  batch_size, batch_count, repetition_count = parameters
  filenames = tf.io.gfile.glob(GCS_PUBLIC_BUCKET + "*.tfrec")
  dataset = load_dataset(filenames)

  batched_dataset = dataset.batch(batch_size)
  sample_dataset = batched_dataset.take(batch_count)

  timings = []
  for rep in range(repetition_count):
    start_time = time.time()
    for picture in sample_dataset:
      print('string', file=open("/dev/null", mode='w'))
    end_time = time.time()
    reading_duration = end_time - start_time
    throughput = float((batch_size * batch_count) / reading_duration)
    dataset_size = batch_size * batch_count
    timings.append([batch_size, batch_count, repetition_count, dataset_size,␣
 ↪throughput])
  return timings

#Task 2aiii
tfrec_rdd = RDD_parameter.map(time_configs_combined)
### TASK 2c ###
```

```
tfrec_rdd.cache()

#Task 2aiv
# For TFRec dataset
tfrec_images_per_second = tfrec_rdd.flatMap(lambda x:␣
 ↪[(str(x[0][0])+'_'+str(x[0][1])+'_'+str(x[0][2]), x[0][4])])
### TASK 2c ###
tfrec_images_per_second.cache()
tfrec_results = tfrec_images_per_second.collect()


#Task 2av)
# For TFRec dataset
tfrec_images_per_second_avg = tfrec_images_per_second \
    .mapValues(lambda x: (x, 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: x[0] / x[1])
tfrec_avg_results = tfrec_images_per_second_avg.collect()


#Task 2avi)
# Define your bucket name and destination file name
PROJECT = 'big-data-cw2-18002699'
destination_blob_name = "average_results_decoded.pkl"

def upload_to_bucket(BUCKET, destination_blob_name, data):
    storage_client = storage.Client()
    bucket = storage_client.get_bucket('{}-storage'.format(PROJECT))
    blob = bucket.blob(destination_blob_name)

    # Write the data to a temporary pickle file
    with blob.open('wb') as f:
        pickle.dump(data, f)


# Upload the results to the bucket
upload_to_bucket(PROJECT, destination_blob_name, tfrec_avg_results)
```

Decoded File

[52]:
```
#Task 2avii)

%%writefile spark_job_decoded.py
### CODING TASK
#Task 2ai
import pandas as pd
import pyspark
```

```
import pickle
import tensorflow as tf
from google.cloud import storage
import time

GCS_PUBLIC_BUCKET = '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

def read_tfrecord(example):
    features = {
        "image": tf.io.FixedLenFeature([], tf.string),
        "class": tf.io.FixedLenFeature([], tf.int64)
    }
    # 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 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

def load_dataset_decoded():
  dataset_filename = tf.data.Dataset.list_files(GCS_PATTERN) #Image Files
  datasetDecoded = dataset_filename.map(decode_jpeg_and_label)
  datasetfn = datasetDecoded.map(resize_and_crop_image)
  return datasetfn

def decode_jpeg_and_label(filepath):
    bits = tf.io.read_file(filepath)
    image = tf.image.decode_jpeg(bits)
    label = tf.strings.split(tf.expand_dims(filepath, axis=-1), sep='/')
```

```
    label2 = label.values[-2]
    return image, label2

def resize_and_crop_image(input_layer):
    image, label = input_layer
    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

batch_sizes = [2,4,6,8]
batch_numbers = [3,6,12,18]
repetitions=[1,2,3]

parameter_list = [[i,j,k] for i in batch_sizes for j in batch_numbers for k in␣
 ↪repetitions]

#Task 2aii
sc=pyspark.SparkContext.getOrCreate()
RDD_parameter = sc.parallelize(parameter_list) #creates an RDD

def time_configs_combined(parameters):
  batch_size, batch_count, repetition_count = parameters
  filenames = tf.io.gfile.glob(GCS_PUBLIC_BUCKET + "*.tfrec")
  dataset = load_dataset(filenames)

  batched_dataset = dataset.batch(batch_size)
  sample_dataset = batched_dataset.take(batch_count)

  timings = []
  for rep in range(repetition_count):
    start_time = time.time()
    for picture in sample_dataset:
      print('string', file=open("/dev/null", mode='w'))
    end_time = time.time()
    reading_duration = end_time - start_time
    throughput = float((batch_size * batch_count) / reading_duration)
```

```
    dataset_size = batch_size * batch_count
    timings.append([batch_size, batch_count, repetition_count, dataset_size,␣
 ↪throughput])
  return timings

#Task 2aiii
tfrec_rdd = RDD_parameter.map(time_configs_combined)
### TASK 2c ###
tfrec_rdd.cache()

#Task 2aiv
# For TFRec dataset
tfrec_images_per_second = tfrec_rdd.flatMap(lambda x:␣
 ↪[(str(x[0][0])+'_'+str(x[0][1])+'_'+str(x[0][2]), x[0][4])])
### TASK 2c ###
tfrec_images_per_second.cache()
tfrec_results = tfrec_images_per_second.collect()


#Task 2av)
# For TFRec dataset
tfrec_images_per_second_avg = tfrec_images_per_second \
    .mapValues(lambda x: (x, 1)) \
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .mapValues(lambda x: x[0] / x[1])
tfrec_avg_results = tfrec_images_per_second_avg.collect()


#Task 2avi)
# Define your bucket name and destination file name
PROJECT = 'big-data-cw2-18002699'
destination_blob_name = "average_results_decoded.pkl"

def upload_to_bucket(BUCKET, destination_blob_name, data):
    storage_client = storage.Client()
    bucket = storage_client.get_bucket('{}-storage'.format(PROJECT))
    blob = bucket.blob(destination_blob_name)

    # Write the data to a temporary pickle file
    with blob.open('wb') as f:
        pickle.dump(data, f)


# Upload the results to the bucket
upload_to_bucket(PROJECT, destination_blob_name, tfrec_avg_results)
```

```
Writing spark_job_decoded.py
```

4.2.2 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.

[53]:
```
%run spark_job_tfrec.py
```

```
<Figure size 640x480 with 0 Axes>
```

[54]:
```
%run spark_job_decoded.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.

[55]:
```
### CODING TASK ###
CLUSTER = '{}-cluster'.format(PROJECT)
!gcloud dataproc clusters create $CLUSTER \
 --image-version 1.4-ubuntu18 \
 --num-masters 1 \
 --master-machine-type n1-standard-1 \
 --master-boot-disk-type pd-ssd --master-boot-disk-size 100 \
 --num-workers 7 \
 --worker-machine-type n1-standard-1 \
 --worker-boot-disk-size 285 \
 --max-idle 3600s \
 --initialization-actions gs://goog-dataproc-initialization-actions-$REGION/
 ↪python/pip-install.sh \
 --metadata PIP_PACKAGES="google-cloud-storage tensorflow==2.4.0"
```

```
ERROR: (gcloud.dataproc.clusters.create) INVALID_ARGUMENT:
Insufficient 'IN_USE_ADDRESSES' quota. Requested 8.0, available 7.0.
```

[56]:
```
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER spark_job_tfrec.py
```

```
Job [ebf93128690b4867bd812b915b4e01ba] submitted.
Waiting for job output…
2023-07-10 15:20:57.025385: W
tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/usr/lib/hadoop/lib/native
```

```
2023-07-10 15:20:57.025433: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Traceback (most recent call last):
  File "/tmp/ebf93128690b4867bd812b915b4e01ba/spark_job_tfrec.py", line 7, in
<module>
    from google.cloud import storage
ModuleNotFoundError: No module named 'google.cloud'
ERROR: (gcloud.dataproc.jobs.submit.pyspark) Job
[ebf93128690b4867bd812b915b4e01ba] failed with error:
Job failed with message [ModuleNotFoundError: No module named 'google.cloud'].
Additional details can be found at:
https://console.cloud.google.com/dataproc/jobs/ebf93128690b4867bd812b915b4e01ba?
project=big-data-cw2-18002699&region=us-central1
gcloud dataproc jobs wait 'ebf93128690b4867bd812b915b4e01ba' --region 'us-
central1' --project 'big-data-cw2-18002699'
https://console.cloud.google.com/storage/browser/dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
b9ff-76c19049ad9e/jobs/ebf93128690b4867bd812b915b4e01ba/
gs://dataproc-staging-us-central1-92774894602-jezqmjjv/google-cloud-dataproc-met
ainfo/8e600862-d06f-44e9-b9ff-76c19049ad9e/jobs/ebf93128690b4867bd812b915b4e01ba
/driveroutput
```

[57]:
```
!gcloud dataproc jobs submit pyspark --cluster $CLUSTER spark_job_decoded.py
```

```
Job [f7b0989475b9447daa4aa85f087b9773] submitted.
Waiting for job output…
2023-07-10 15:21:32.016419: W
tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/usr/lib/hadoop/lib/native
2023-07-10 15:21:32.016459: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
Traceback (most recent call last):
  File "/tmp/f7b0989475b9447daa4aa85f087b9773/spark_job_decoded.py", line 7, in
<module>
    from google.cloud import storage
ModuleNotFoundError: No module named 'google.cloud'
ERROR: (gcloud.dataproc.jobs.submit.pyspark) Job
[f7b0989475b9447daa4aa85f087b9773] failed with error:
Job failed with message [ModuleNotFoundError: No module named 'google.cloud'].
Additional details can be found at:
https://console.cloud.google.com/dataproc/jobs/f7b0989475b9447daa4aa85f087b9773?
project=big-data-cw2-18002699&region=us-central1
gcloud dataproc jobs wait 'f7b0989475b9447daa4aa85f087b9773' --region 'us-
central1' --project 'big-data-cw2-18002699'
https://console.cloud.google.com/storage/browser/dataproc-staging-us-
central1-92774894602-jezqmjjv/google-cloud-dataproc-metainfo/8e600862-d06f-44e9-
```

```
b9ff-76c19049ad9e/jobs/f7b0989475b9447daa4aa85f087b9773/
gs://dataproc-staging-us-central1-92774894602-jezqmjjv/google-cloud-dataproc-met
ainfo/8e600862-d06f-44e9-b9ff-76c19049ad9e/jobs/f7b0989475b9447daa4aa85f087b9773
/driveroutput
```

4.2.3 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**

4.2.4 2d) Retrieve, analyse and discuss the output (12%)

Run the tests over a wide range of different paramters and list the results in a table.

Perform a **linear regression** (e.g. using scikit-learn) over **the values for each parameter** and for the **two cases** (reading from image files/reading TFRecord files). List a **table** with the output and interpret the results in terms of the effects of overall.
Also, **plot** the output values, the averages per parameter value and the regression lines for each parameter and for the product of batch_size and batch_number

Discuss the **implications** of this result for **applications** like large-scale machine learning. Keep in mind that cloud data may be stored in distant physical locations. Use the numbers provided in the PDF latency-numbers document available on Moodle or here for your arguments.

How is the **observed** behaviour **similar or different** from what you'd expect from a **single machine**? Why would cloud providers tie throughput to capacity of disk resources?

By **parallelising** the speed test we are making **assumptions** about the limits of the bucket reading speeds. See here for more information. Discuss, **what we need to consider** in **speed tests** in parallel on the cloud, which bottlenecks we might be identifying, and how this relates to your results.

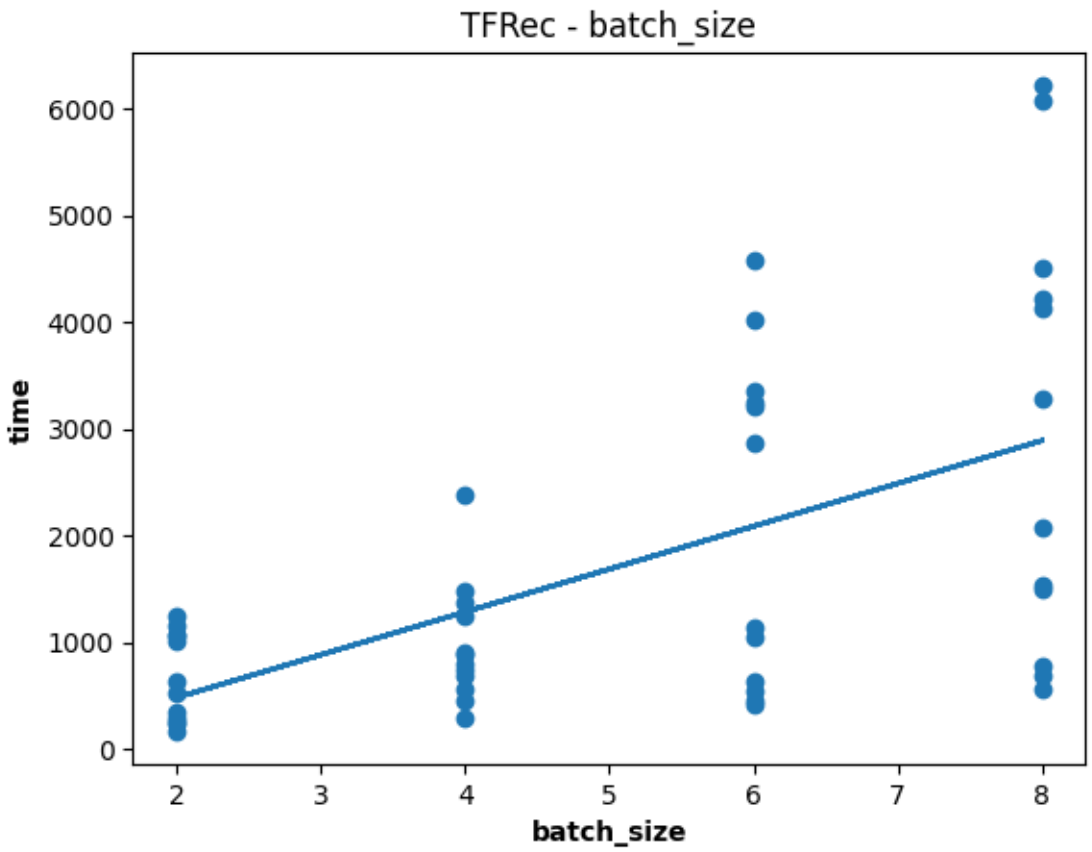
Discuss to what extent **linear modelling** reflects the **effects** we are observing. Discuss what could be expected from a theoretical perspective and what can be useful in practice.

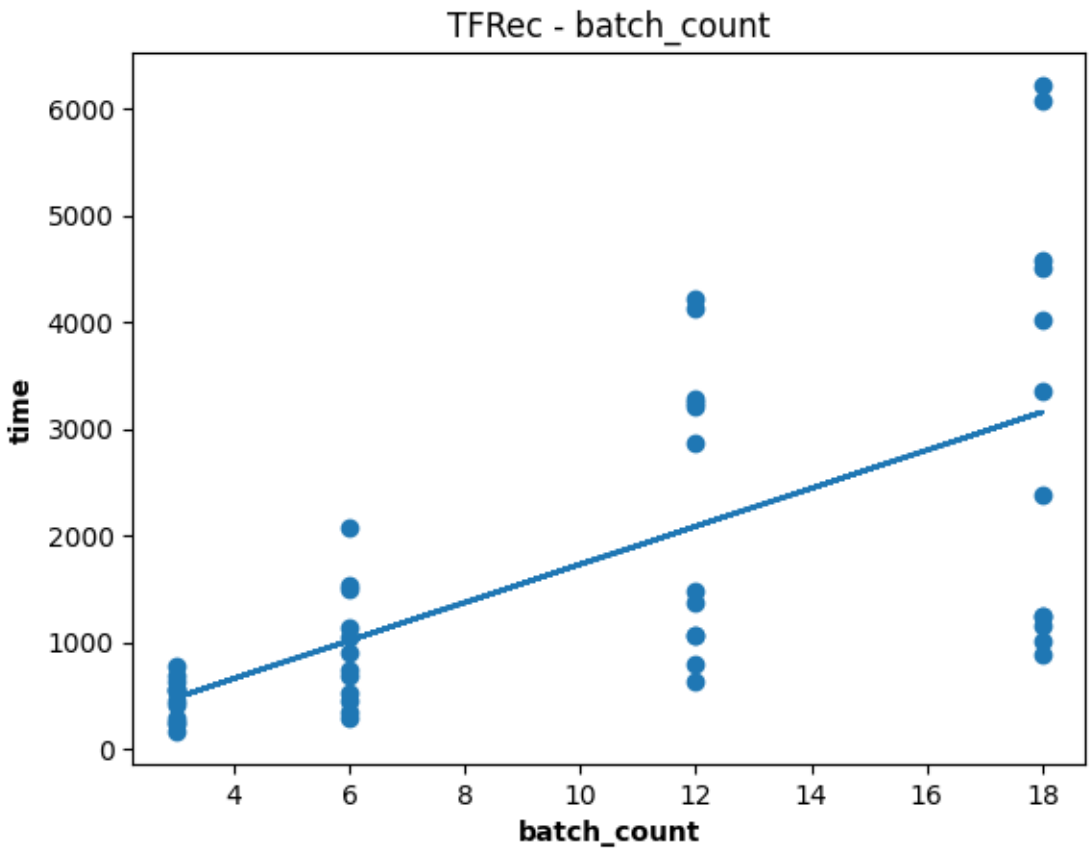
Write your **code below** and **include the output** in your submitted `ipynb` file. Provide the answer **text in your report**.

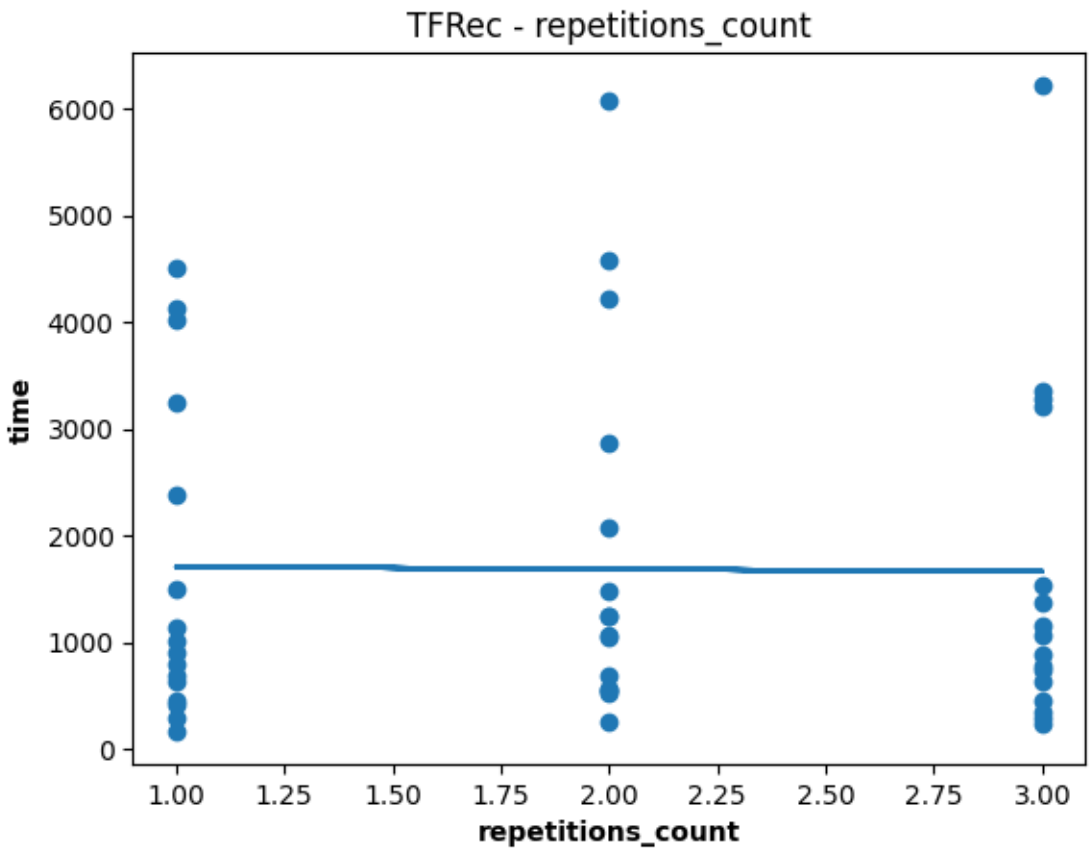
[59]:
```
### CODING TASK ###
from sklearn.linear_model import LinearRegression
import numpy as np
import scipy
```

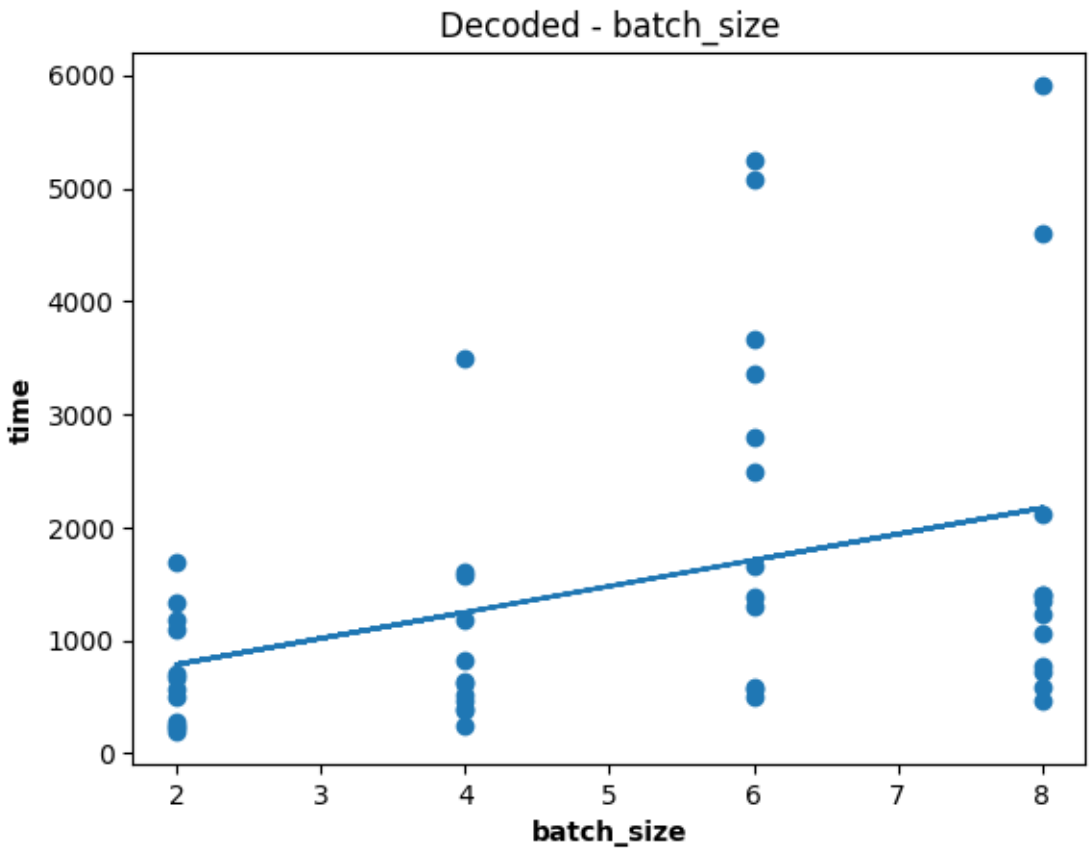
```
from matplotlib import pyplot as plt
from google.cloud import storage
import pickle

def load_pickle(filename):
    client = storage.Client()
    bucket_name = "{}-storage".format(PROJECT)
    bucket = client.get_bucket(bucket_name)
    blob = bucket.blob(filename)

    with blob.open("rb") as f:
        file = pickle.load(f)

    return file


# Convert the parameter file to numeric values
def get_numeric_params(pickle_entries):
    numeric_params = []
    for entry in pickle_entries:
      combination = entry[0].split("_")
      combination = [int(param_value) for param_value in combination]
      time = entry[1]
      numeric_params.append([combination, time])

    return numeric_params


# Prepare the data for the regression
tfrec_avg_results = load_pickle("average_results_tfrec.pkl")
decoded_avg_results = load_pickle("average_results_decoded.pkl")

tfrec_num = get_numeric_params(tfrec_avg_results)
decoded_num = get_numeric_params(decoded_avg_results)

def get_individual_parameter(array, name):
    values = []
    if name == "batch_size":
        values = [value[0][0] for value in array]
    elif name == "batch_count":
        values = [value[0][1] for value in array]
    elif name == "repetitions_count":
        values = [value[0][2] for value in array]

    time_values = [value[1] for value in array]

    return values, time_values
```

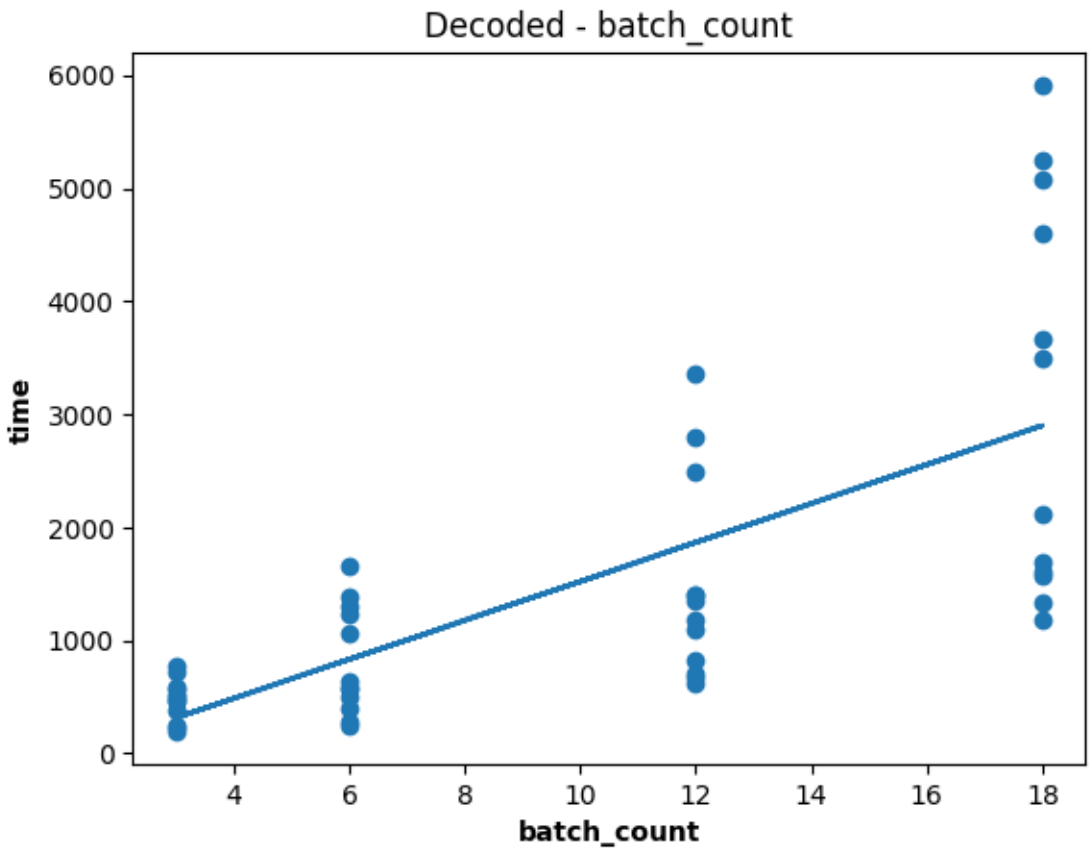
```
def plot_data(numeric_dataset, param_name, dataset_name):
  param, time = get_individual_parameter(numeric_dataset, param_name)

  plt.scatter(param, time)
  slope, intercept, _, _, _ = scipy.stats.linregress(np.array(param), np.
 ↪array(time))
  line = intercept + slope * np.array(param)
  plt.plot(param, line)
  plt.xlabel(param_name, fontweight="bold")
  plt.ylabel("time", fontweight="bold")
  plt.title(dataset_name)
  plt.show()



plot_data(tfrec_num, "batch_size", "TFRec - batch_size")
plot_data(tfrec_num, "batch_count", "TFRec - batch_count")
plot_data(tfrec_num, "repetitions_count", "TFRec - repetitions_count")

plot_data(decoded_num, "batch_size", "Decoded - batch_size")
plot_data(decoded_num, "batch_count", "Decoded - batch_count")
plot_data(decoded_num, "repetitions_count", "Decoded - repetitions_count")
```

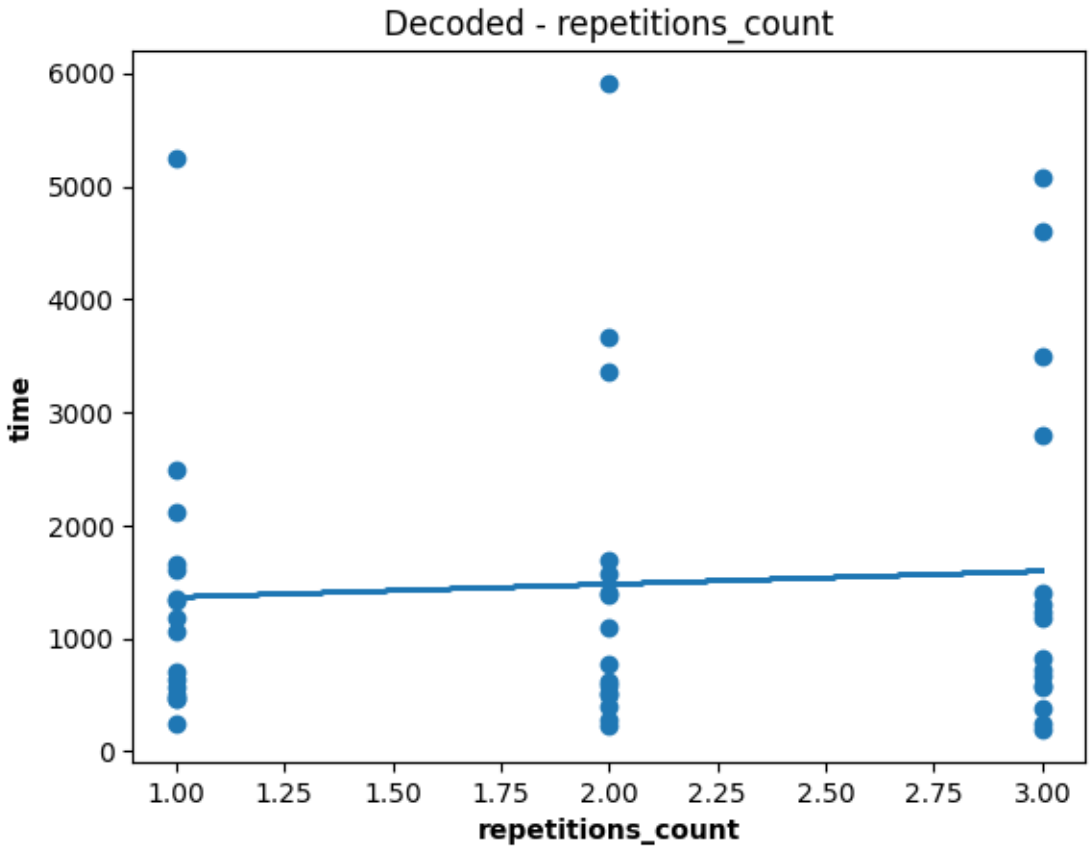

TFRec - batch_size
time
batch_size
0
1000
2000
3000
4000
5000
6000
2
3
4
5
6
7
8


TFRec - batch_count
time
batch_count
0
1000
2000
3000
4000
5000
6000
4
6
8
10
12
14
16
18


TFRec - repetitions_count
time
repetitions_count
0
1000
2000
3000
4000
5000
6000
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00


Decoded - batch_size
time
batch_size
0
1000
2000
3000
4000
5000
6000
2
3
4
5
6
7
8


Decoded - batch_count
time
batch_count
0
1000
2000
3000
4000
5000
6000
4
6
8
10
12
14
16
18



5 Section 3. Theoretical discussion

5.1 Task 3: Discussion in context. (24%)

In this task we refer an idea that is introduced in this paper: - Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., & Zhang, M. (2017). Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics.. In USENIX NSDI 17 (pp. 469-482).

Alipourfard et al (2017) introduce the prediction an optimal or near-optimal cloud configuration for a given compute task.

5.1.1 3a) Contextualise

Relate the previous tasks and the results to this concept. (It is not necessary to work through the full details of the paper, focus just on the main ideas). To what extent and under what conditions do the concepts and techniques in the paper apply to the task in this coursework? (12%)

5.1.2 3b) Strategise

Define - as far as possible - concrete strategies for different application scenarios (batch, stream) and discuss the general relationship with the concepts above. (12%)

Provide the answers to these questions in your report.

5.2 Final cleanup

Once you have finshed the work, you can delete the buckets, to stop incurring cost that depletes your credit.

[ ]:
```
!gsutil -m rm -r $BUCKET/* # Empty your bucket
!gsutil rb $BUCKET # delete the bucket
```